AgentWhetters: Spatial Reasoning Skills for Cost-Effective Block-Building Agents

Whitten, Paul
pcw@case.edu

Chen, Li-Jen
chenlijen@gmail.com

Baddam, Sharath
sbaddam@alumni.purdue.edu

March 2026

Abstract

We present **AgentWhetters BWIM**, a purple agent for the AgentBeats *Build What I Mean* (BWIM) benchmark [2, 1] that constructs block structures on a $9 \times 5 \times 9$ grid from sometimes under-specified natural-language instructions. The system originated as an investigation into enabling small, cost-efficient language models, originally targeting edge deployment with limited compute, to perform reliable 3D spatial reasoning. It has since been adapted as a competition entry for the AgentX AgentBeats Phase 2 Sprint 1, where it is evaluated on structural accuracy and question efficiency against both a rational and an unreliable architect partner. Rather than relying on large frontier models, our approach augments GPT-4o-mini with a modular skills pipeline that offloads spatial computation to deterministic code, reducing the burden on the language model to tasks it handles well: instruction comprehension, ambiguity detection, and structured plan generation. The name *AgentWhetters* reflects our view that competitions such as AgentX AgentBeats offer a valuable opportunity to *whet* or *sharpen* our skills in developing, evaluating, and iterating on autonomous agents under realistic constraints.

1 Related Work

Large language models exhibit well-documented weaknesses in spatial and directional reasoning. Yamada et al. [3] benchmark LLMs on spatial tasks and find that chain-of-thought prompting and explicit coordinate tracking improve accuracy, while Bang et al. [4] document persistent failures in relative positioning and chain references for GPT-family models. These findings motivate our design: rather than relying on the LLM for coordinate computation, we confine it to instruction comprehension and plan generation, delegating all spatial arithmetic to deterministic code.

Our pipeline draws on several prior approaches. The plan-then-execute decomposition follows Wang et al. [9], who show that separating planning from execution improves zero-shot reasoning. The modular skill structure is informed by decomposed prompting [8], which splits complex tasks into sub-problems handled by specialized modules. The generate-verify-fix loop in our plan verifier parallels the self-refinement approach of Madaan et al. [6]. Our use of deterministic code for coordinate arithmetic reflects the tool-augmented approach explored by Schick et al. [7], and the overall split between language understanding and symbolic execution echoes the neuro-symbolic design of Yi et al. [10].

2 Architecture

The agent decomposes each building instruction through a five-stage pipeline: **parse**, **plan**, **detect under-specification**, **execute**, and **format**. If any stage fails, control falls back to a direct LLM call with an engineered system prompt.

1. **Instruction Parser.** Extracts the green agent's instruction text, separates the building directive from any existing grid state (START_STRUCTURE), and normalizes the representation for downstream stages.

2. **Structure Analyzer.** Before planning, the current grid state is analyzed to detect geometric primitives rows, stacks, L-shapes, T-shapes and produce a structured description. This pre-computed geometry

is injected into the planner prompt so the language model reasons over named shapes rather than raw coordinate lists.

3. **Build Planner (LLM).** The language model decomposes the instruction into a sequence of atomic build steps expressed as typed JSON actions (`stack`, `place_relative`, `extend_row`, etc.), each with a color, count, and position specification. Nine worked examples in the system prompt, following the chain-of-thought approach [5], cover chains, L-shapes, T-shapes, and edge placements to anchor the model's spatial reasoning within the coordinate system.

4. **Plan Verifier and Deterministic Corrections.** A rule-based post-planner validates the LLM-generated plan against the instruction text and the current grid geometry. Four deterministic correction passes run before execution:

   (a) *Direction consistency:* compares directional words in the instruction ("left," "right," "front," "behind") against the plan's `direction` fields and flips any contradiction.

   (b) *Each-end cap correction:* when the instruction contains "each end" or "both ends" and the plan extends a row, the system recomputes the new endpoints from the extension geometry and relocates any cap-placement steps to the correct positions.

   (c) *T-shape extend correction:* runs the structure analyzer on the starting grid to detect T-shaped geometry, computes the stem's axis and growth direction from block coordinates, and replaces invalid or incorrect extend directions (the LLM frequently generates `on_top`, which is not a valid horizontal direction).

   (d) *Stacking and count plausibility:* checks horizontal-versus-vertical intent and block-count consistency.

   These corrections operate on general geometric properties of the grid and instruction text. They do not reference specific stimuli or expected outputs and generalize to arbitrary instructions and grid configurations.

5. **Spatial Executor.** A deterministic engine executes each plan step on an in-memory grid model, resolving relative references ("leftmost," "the red one"), computing gravity-based y-coordinates, and chaining positional context across steps. No LLM call is involved at this stage; all coordinate arithmetic is exact. A same-color skip-forward rule in the `extend_row` handler detects when the specified start position is already occupied by a block of the same color and advances the start by one grid step in the extension direction before placing. This prevents vertical stacking when the LLM intends horizontal extension from an existing block.

   A key design choice is that the LLM planner operates in what is effectively a two-dimensional problem space. Each plan step specifies only a horizontal position (x, z) and an action type; the executor computes the vertical coordinate automatically by scanning the column for existing blocks and placing each new block at the next available height, much like dropping a piece in Connect 4. This means the language model never needs to perform y-coordinate arithmetic (ground level at $y = 50$, each block adding $+100$) or track how many blocks already occupy a column. In our initial approach, the LLM was asked to produce the complete `[BUILD];Color,x,y,z;...` response directly, which required it to enumerate every block's three-dimensional coordinates and to correctly chain height calculations across multiple stacking operations. That approach suffered from frequent off-by-one y-errors, duplicate blocks at the same position, and miscounted stack heights. The current executor eliminates these failure modes entirely by confining the LLM to the two-dimensional layout it can reason about reliably and delegating all vertical placement to deterministic code.

6. **Response Formatter.** Translates the final grid state into the protocol-required `[BUILD]` format and validates the output before transmission.

3 Underspecification Detection

A key challenge in the benchmark is that many instructions omit color or block count. Our agent employs a two-tiered detection strategy:

- **Missing color.** Heuristic analysis determines whether a color can be inferred from context (e.g., reusing the sole color mentioned) or is genuinely ambiguous. In the latter case, the agent issues an `[ASK]` query to the architect. Because a correct answer yields +10 while asking costs −5, the expected value of asking (+5) exceeds guessing (≈ 0). A multi-color disambiguation loop handles instructions that leave multiple colors unspecified.

- **Missing count.** Empirical analysis of the stimulus data revealed that heuristic count inference (copying an adjacent stack's height or defaulting to three) achieves only ∼65% accuracy, well below the 75% threshold at which guessing dominates asking under the scoring function. At $p = 0.65$, $\mathrm{EV}_{\mathrm{guess}} = 20(0.65) - 10 = +3.0$, while $\mathrm{EV}_{\mathrm{ask}} = +5.0$ per round, a gain of +2.0 per count-underspecified round.

  A complication is that the architect answering questions is itself an LLM (GPT-4o-mini) that receives only the target structure as coordinates, not the original instruction. When presented with a generic question such as "How many blocks should be in the unspecified stack?" the architect cannot determine which stack is being referenced and defaults to answering "3 blocks" regardless of the true count, producing a 23% error rate in our experiments. To address this, we generate *color-specific* questions that name the color of the uncounted phrase, for example "How many blue blocks should be in the blue stack?" This allows the architect to identify the correct stack by counting blocks of that color in its coordinate data. The answered count is then patched directly into the instruction text before the LLM planner runs, so the planner receives a fully specified instruction such as "build a stack of 3 blue blocks" rather than the ambiguous "build a blue stack."

  Because only one question is permitted per round, the agent prioritizes color over count. When multiple phrases lack counts, the first uncounted phrase with a known color is selected for the question and the remainder fall back to heuristic inference. The heuristic resolves unspecified counts through a three-level cascade: first, it copies the height of an adjacent stack at the target position; second, if no neighbor exists, it uses the tallest stack currently on the grid; third, it defaults to three, which matches the modal count in the stimulus data (53% of count-underspecified trials).

4 Adaptive Prompt Enrichment

We observe that certain spatial concepts such as "each end," chain references, L-shape extensions, T-shape junctions are disproportionately error-prone for small models. Our *adaptive prompt enrichment* system scans each incoming instruction against a library of 15 pattern-matching rules. When a rule fires, a short concept definition with a few-shot worked example (concrete coordinates showing correct and incorrect placements) is injected into the user prompt immediately before the LLM call. This places the relevant spatial rule and a grounding example in the model's local attention window, adjacent to the instruction it must interpret. Rules are independent and composable; multiple may fire on a single instruction without conflict. The enrichment library is validated against the full stimulus set to ensure zero false positives on fully-specified instructions.

To guard against teaching the test, all few-shot examples are constructed with different colors, counts, positions, and phrasing from the evaluation scenarios. We measure unigram+bigram cosine similarity between each example and every stimulus instruction. The highest similarity across all examples is 0.71, while real stimulus instructions reach pairwise similarities of up to 0.80 among themselves. Our examples therefore share less surface overlap with any individual test item than the test items share with each other.

5 Additional Design Considerations

- **Chain reference patching.** A dedicated pre-processing pass resolves chain references ("to the right of the green one") that span multiple build steps, ensuring positional context propagates correctly.

- **Ask-once guard.** The agent tracks whether it has already asked a question in the current round, preventing redundant queries that would incur additional point penalties.

- **Cost efficiency.** The entire pipeline uses GPT-4o-mini, one of the lowest-cost commercial models available. Deterministic execution of spatial computation means the LLM is called at most twice per round (planning plus at most one clarification re-plan), keeping token usage minimal.
- **Graceful degradation.** If the skills pipeline encounters an unrecoverable error at any stage, the agent falls back to a direct LLM call with a detailed system prompt containing coordinate examples and spatial rules, ensuring every round produces a valid `[BUILD]` response.

6 Results

Table 1 summarizes score progression across development iterations. In local evaluation against the benchmark's 160-round scenario (combining rational and unreliable architect partners), the agent achieves 93.8–95.6% structural accuracy across three consecutive runs with the final codebase, scoring +920 to +980 (mean +953, mean accuracy 94.8%). All three runs use GPT-4o-mini. A fourth run using GPT-4o scores +960 (95.0% accuracy), falling within the same range as the GPT-4o-mini results. This confirms that the deterministic pipeline rather than model size drives the performance gains.

Run	Changes	Score	Accuracy	T-shape	Each-end
Baseline	No count-ask	+780	81.9%	–	–
+Count-ask	Count questions	+720	87.5%	0/4	0/4
+Enrichment	Prompt enrichment	+780	88.1%	0/4	1/4
+Each-end fix	Deterministic cap fix	+820	89.4%	1/4	4/4
+Example rewrite	Decontaminated examples	+860	91.9%	2/4	2/4
+Extend skip	Executor same-color fix	+980	95.6%	4/4	4/4

Table 1: Score progression across development iterations (160 rounds each).

Adding count-specific `[ASK]` questions raised structural accuracy from 82% to 87% at the cost of a lower raw score due to question penalties. Adaptive prompt enrichment and deterministic post-plan corrections recovered the score while further improving accuracy. The each-end deterministic fix alone moved that sub-task from 1/4 to 4/4 correct, contributing +40 net points. A subsequent fix to the underspecification detector's question targeting (asking about the correct uncounted stack rather than a misidentified one) addresses 6 additional failure rounds in testing. Rewriting few-shot examples to avoid stimulus contamination and adding a fallback each-end correction path brought the overall accuracy to 91.9%.

The final improvement, a same-color skip-forward rule in the spatial executor, corrected a class of failures where `extend_row` stacked vertically on an existing block of the same color instead of extending horizontally past it. This fix raised the score from +860 to +980 and moved both T-shape (2/4 to 4/4) and each-end (2/4 to 4/4) sub-tasks to perfect accuracy.

Across the three final runs (7, 8, and 10 failures respectively), the failure causes are consistent. In the best run, 5 of 7 failures originate in the green agent (architect) providing incorrect color or count information. For example, the architect names a block color as Green when the target structure contains Blue, or reports a count of 4 for a stack that should have 3. These errors arise because the green agent is itself an LLM (GPT-4o-mini) that sometimes misreads its own coordinate data. The remaining 2 failures are LLM spatial reasoning errors in the purple agent: one incorrect chain coordinate reference and one L-shape with combined wrong-color and vertical stacking.

The dominant remaining error source is the green agent rather than the purple agent's pipeline. The purple agent cannot currently detect or compensate for architect errors because it has no independent access to the target structure. One possible mitigation would be a plausibility check that flags unlikely color or count values in the instruction, but the current scoring function (-5 per question) limits the value of additional verification queries.

7 Conclusion

We described AgentWhetters BWIM, a builder agent that pairs GPT-4o-mini with a deterministic spatial execution pipeline to achieve 93.8–95.6% accuracy (mean 94.8%) on the BWIM benchmark across three evaluation runs. A GPT-4o run scores 95.0%, falling within the same range, confirming that the deterministic pipeline rather than model capacity is the primary driver of performance. The core lesson is that small language models can perform reliably on spatial tasks when the problem is decomposed so that the model handles language understanding and a code layer handles coordinate arithmetic. Four rule-based correction passes fix recurring LLM errors without referencing expected outputs, and each generalizes to arbitrary grid configurations. The majority of remaining failures originate in the architect agent rather than our pipeline, suggesting that the primary bottleneck has shifted from builder reasoning to architect reliability.

References

[1] Berkeley RDI AgentX. *AgentBeats Competition*, 2026. `https://rdi.berkeley.edu/agentx-agentbeats.html`

[2] UvA LTL. *Build What I Mean*, 2026. `https://github.com/ltl-uva/build_what_i_mean`

[3] Y. Yamada, Y. Bao, A. K. Lampinen, J. Kasai, and I. Yildirim. Evaluating spatial understanding of large language models. *Transactions on Machine Learning Research*, 2024.

[4] Y. Bang et al. A multitask, multilingual, multimodal evaluation of ChatGPT on reasoning, hallucination, and interactivity. In *AACL*, 2023.

[5] J. Wei et al. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

[6] A. Madaan et al. Self-refine: Iterative refinement with self-feedback. In *NeurIPS*, 2023.

[7] T. Schick et al. Toolformer: Language models can teach themselves to use tools. *arXiv:2302.04761*, 2023.

[8] T. Khot et al. Decomposed prompting: A modular approach for solving complex tasks. In *ICLR*, 2023.

[9] L. Wang et al. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *ACL*, 2023.

[10] K. Yi et al. Neural-symbolic VQA: Disentangling reasoning from vision and language understanding. In *NeurIPS*, 2018.