EVM-QuestBench: An Execution-Grounded Benchmark for Natural-Language Transaction Code Generation

Pei Yang[1*] **Wanyi Chen**[2*] **Ke Wang**[1] **Lynn Ai**[1] **Eric Yang**[1] **Tianyu Shi**[1†]
[1]Gradient
[2]Soochow University

Abstract

Large language models are increasingly applied to various development scenarios. However, in on-chain transaction scenarios, even a minor error can cause irreversible loss for users. Existing evaluations often overlook execution accuracy and safety. We introduce EVM-QuestBench, an execution-grounded benchmark for natural-language transaction-script generation on EVM-compatible chains. The benchmark employs dynamic evaluation: instructions are sampled from template pools, numeric parameters are drawn from predefined intervals, and validators verify outcomes against these instantiated values. EVM-QuestBench contains 107 tasks (62 atomic, 45 composite). Its modular architecture enables rapid task development. The runner executes scripts on a forked EVM chain with snapshot isolation; composite tasks apply step-efficiency decay. We evaluate 20 models with 5 independent rounds each and find large performance gaps, with split scores revealing persistent asymmetry between single-action precision and multi-step workflow completion. Code: https://github.com/OpenEdgeHQ/EVM-quest-bench.

1 Introduction

Large language models (LLMs) are increasingly being used to control software and tools (OpenAI, 2023; Anil et al., 2023). Leveraging LLMs for code generation and blockchain transactions is becoming commonplace, but this introduces significant financial risks. Even a small error, such as an incorrect address, unit, or deadline, can result in irreversible losses.

Benchmarks for code understanding and generation need to cover a broad range of programming tasks and reasoning capabilities (Chen et al., 2021; Lu et al., 2021). Many evaluations still rely on lexical overlap metrics such as BLEU or CodeBLEU (Papineni et al., 2002; Ren et al., 2020). These metrics can reward outputs that appear similar to references but fail to run or fail to meet functional constraints. Benchmarks such as SWE-bench (Jimenez et al., 2024) focus on real-world software engineering tasks like bug fixing in Python repositories. Due to the difference in test domains, such benchmarks cannot directly reflect a model's ability to execute transactions in blockchain environments. Blockchain-specific benchmarks such as Solana Bench (Solana Foundation, 2025) explore the boundaries of LLM capabilities in Web3 but struggle to provide feedback on the accuracy of natural language understanding and the safety of transaction execution.

Transaction script generation presents a distinct set of failure modes. LLMs must first interpret diverse natural language instructions. The generated code must correctly construct calldata, account for chain-specific units and token decimals, adhere to protocol constraints, and manage dependencies across multiple transaction steps. Even minor deviations can lead to transaction reverts, partial execution, or incorrect state transitions. These characteristics make blockchain automation an ideal domain for execution-based evaluation.

We introduce EVM-QuestBench, an execution-grounded benchmark with two splits. **Atomic tasks** test single-action precision. **Composite tasks** test multi-step workflows that require planning, prerequisite handling, and parameter propagation. Composite scoring incorporates a step-efficiency factor that penalizes unnecessary steps. EVM-QuestBench contains 107 tasks with a maximum total score of 10,700. This design significantly simplifies benchmark development and maintenance. Creating an atomic task only requires defining the problem and developing a validator. Creating a new composite task only requires updating a JSON file.

*Equal contribution.
†Corresponding author: tianyu@gradient.network.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 35513–35529

We evaluate 20 models under a unified protocol with 5 independent rounds per model. Results show substantial variance. Split scores reveal a persistent capability asymmetry between single-action precision and workflow completion. Several models achieve strong Composite performance with weaker Atomic scores. Several models fail on Composite workflows despite non-trivial Atomic performance. These patterns motivate split reporting for diagnosis.

Our contributions are as follows:

- We release EVM-QuestBench, a benchmark for natural language to transaction script generation on EVM-compatible chains, with Atomic and Composite splits. Its declarative task design keeps evaluation costs low: each task round consumes a small number of tokens, enabling comprehensive model assessment at minimal token cost.
- We introduce an atomic/composite benchmark paradigm that significantly reduces development costs, especially when using LLMs to assist development.
- We provide an execution protocol with snapshot isolation, a fixed runner interface, and validator-based scoring over receipts and post-state constraints.
- We report results on 20 models with 5-round statistical analysis, including standard deviations, observed ranges, and rank-order consistency, that separates single-action precision from multi-step workflow completion.

2 Related Work

Execution-based evaluation for code generation. Reference-based code evaluation metrics such as BLEU and CodeBLEU (Papineni et al., 2002; Ren et al., 2020) share a fundamental flaw: they measure surface similarity rather than functional correctness, so code that resembles a reference but fails at runtime can still receive a high score. Execution-based benchmarks address this by running generated programs against tests, including HumanEval and MBPP for function synthesis (Chen et al., 2021; Austin et al., 2021), and extensions to library usage and multilingual settings (Wang et al., 2023; Khan et al., 2024; Yan et al., 2024). However, most prior benchmarks focus on stateless, sandboxed functions and do not model shared external state or irreversible actions.

Real-world software engineering, agent benchmarks, and blockchain constraints. SWE-bench evaluates issue resolution by applying patches to real repositories (Jimenez et al., 2024), and agent-style benchmarks study multi-step tool use in interactive environments (Qin et al., 2023; Liu et al., 2023). However, these benchmarks do not target the constraints that characterize blockchain interaction: shared mutable state, protocol prerequisites, strict unit and decimal handling, and irreversible revert risk. Prior blockchain benchmarks such as Solana-focused transaction evaluations (Solana Foundation, 2025) also do not disentangle single-transaction precision from multi-transaction workflow completion under a unified execution and validation interface. EVM-QuestBench fills this gap by evaluating natural-language-to-transaction-script generation on EVM-compatible chains with atomic/composite splits, using declaratively specified tasks and reusable validator components that keep development costs low while preserving execution-grounded evaluation.

3 Benchmark Tasks

3.1 Benchmark Overview

EVM-QuestBench evaluates whether a model can convert a natural language goal into an executable outcome on EVM-compatible chains. Each task specifies a target end state. For evaluation, the runner samples a template from a pre-built pool and instantiates numeric parameters within predefined ranges. The model outputs a TypeScript module that constructs transaction request objects. The runner executes the plan on a forked chain, and validators score post-execution constraints.

Figure 1 illustrates the layered architecture. The design is modular: adding a new atomic task requires only a JSON specification and a validator; adding a composite task requires only a JSON file. At evaluation time, the system flows through dynamic instantiation (template and parameter sampling), LLM interaction (code generation), code execution (sandboxed TypeScript runtime), transaction execution (signing and broadcasting on the fork), and validation (weighted scoring against dynamically sampled parameters).

The benchmark targets transaction script generation rather than contract synthesis. Failures commonly arise from incorrect calldata construction, unit conversion, or missing protocol prerequisites.

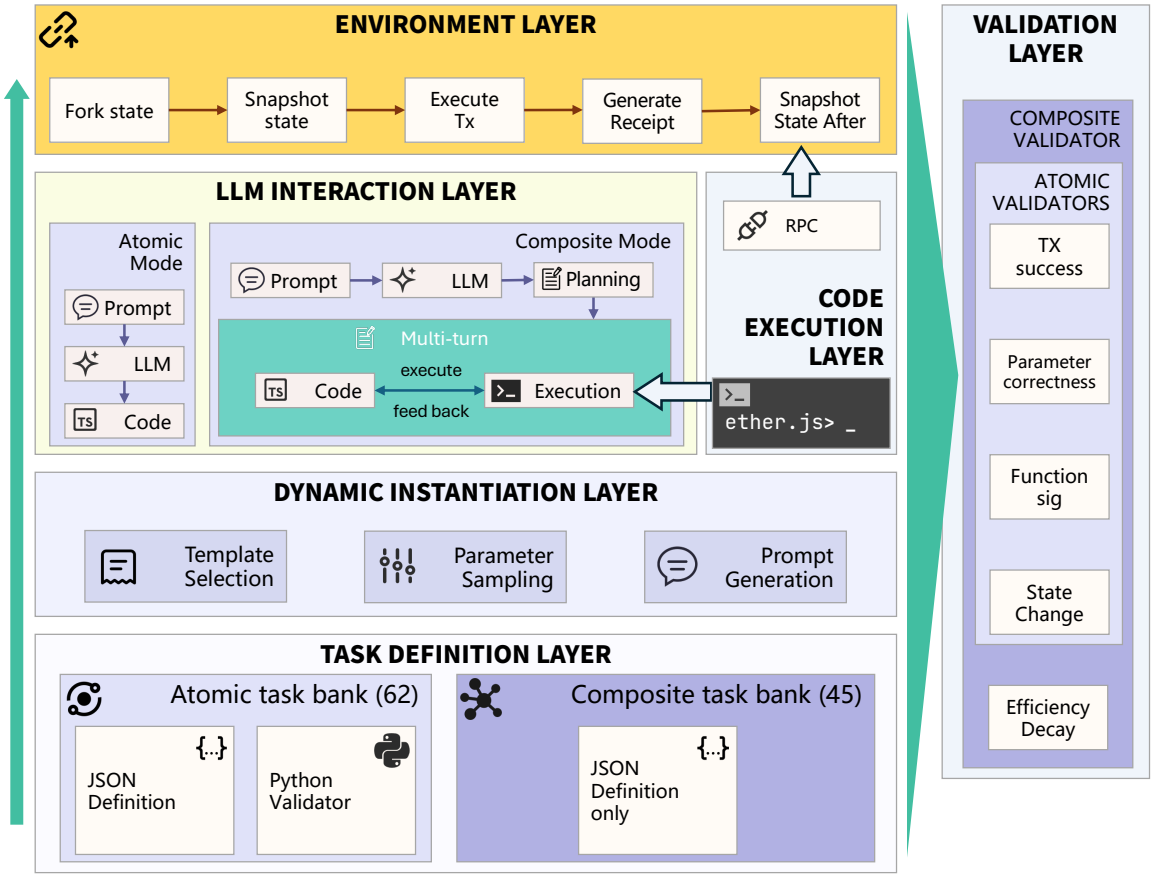

Figure 1: EVM-QuestBench evaluation architecture and end-to-end pipeline. A natural language instruction is sampled from a template pool with dynamic numeric parameters, passed to the LLM for TypeScript script generation, executed on a snapshot-isolated forked chain, and scored by task-specific validators against post-state constraints. Composite tasks additionally apply step-efficiency decay.

3.2 Task Specification and Data Format

Each task is stored as a JSON specification containing an identifier, metadata, natural language templates, parameter definitions, and a validation configuration. Atomic and composite tasks share a common surface structure (templates, parameters, and validation). Composite tasks additionally include an explicit workflow structure and a scoring strategy, enabling evaluation of multi-transaction dependencies.

Atomic task schema. An atomic task defines a single on-chain action and its expected effects. The specification includes: (i) natural language templates that render user instructions, (ii) typed parameters with default ranges, and (iii) a validator class with task-specific arguments (e.g., token address, recipient address, amount, decimals). At evaluation time, the runner instantiates the validator and computes a weighted score from post-execution checks.

Composite task schema. A composite task defines a multi-transaction workflow and an end state condition. Each composite specification includes a `composite_structure` field that names a workflow pattern, sets `optimal_steps`, and enumerates step-level atomic operations. Composite validation checks whether the final on-chain condition holds, then applies a step-efficiency decay described in Section 4. This design prioritizes end-to-end completion over intermediate surface matching.

Running example. A typical instance provides an instruction such as *"Swap 0.1 BNB to USDT"* together with an execution context (RPC endpoint, agent address, and a contract address map). The model returns a module whose `executeSkill` function emits a transaction request that calls the router with a concrete `to` address and `data` calldata. The validator then checks transaction success and verifies the expected balance change within the task tolerance (Section 3.6).

3.3 Benchmark Composition

EVM-QuestBench contains 107 tasks (62 atomic, 45 composite) with a maximum total score of 10,700. Figure 4 (Appendix B) shows the split sizes and Figure 5 (Appendix B) shows the composite workflow complexity.

Atomic tasks require one on-chain action. The task bank is organized into three categories: `basic_transactions` (40), `defi_operations` (19), and `advanced_features` (3). Basic transactions cover wallet and token operations (queries, native transfers, ERC-20 transfers, approvals). DeFi operations cover swaps, liquidity, and staking. Advanced features cover edge cases (fallback handling, `delegatecall`, flashloans). Atomic tasks stress parameter correctness, unit handling, and precise state change targets.

Composite tasks require multi-transaction workflows with prerequisite approvals, protocol-imposed intermediate steps, and consistent parameter propagation. Examples include approve→swap, approve→add liquidity, and add liquidity→stake. Workflows range from 2 to 6 optimal steps (mean 3.27, median 3), concentrated at 3 steps (53.3%). This split separates single-action precision from multi-step workflow completion.

3.4 Difficulty and Coverage

Each task is annotated with a difficulty label from {`easy`, `easy-medium`, `medium`, `hard`}. Atomic tasks are dominated by `medium` (56.5%), while composite tasks skew harder (`hard`: 24.4%), reflecting additional constraints from prerequisite handling and step ordering. Atomic tasks span subcategories including ERC-20 operations, native transfers, NFT operations, swaps, staking, and queries. Composite tasks diversify by workflow motifs: batch operations, swap-centric workflows, liquidity workflows, staking workflows, and query-and-verify patterns.

3.5 Instruction Templates and Parameterization

Each task provides multiple natural language templates (atomic: 3–5, mean 3.97; composite: 2–4, mean 2.82), generated by an LLM to ensure diverse and realistic phrasing. At evaluation time, the runner randomly selects one template and samples numeric parameter values uniformly within predefined ranges, injecting them into the selected template and passing the resolved parameters to validators. Dynamic sampling prevents trivial memorization, forces models to generalize across arbitrary values and unit conversions, and keeps per-run token consumption low by reusing the same 107 tasks with fresh parameters across rounds.

To control for wording bias, all 373 templates were rated on a 1–5 clarity scale by three SOTA models and averaged: 336 are *precise* (90.1%), 37 are *moderate* (9.9%), and none are *vague*, confirming the pool is consistently unambiguous. Full details and the difficulty selection API are in Appendix F.

3.6 Validators and Post-Execution Constraints

EVM-QuestBench uses validator-based scoring rather than reference code matching. Validators receive the same dynamically sampled parameters injected into the natural language instruction and verify that the model's output produces the expected on-chain effects for those specific values.

Each atomic task type has a dedicated validator that converts human-readable parameters to chain-native units (e.g., 0.1 tokens $\rightarrow 0.1 \times 10^{18}$ wei) and compares against actual on-chain state changes. A typical atomic validator computes a weighted score from four binary checks: transaction success (30 pts), contract address correctness (20 pts), function signature correctness (20 pts), and state change verification (30 pts). Tolerance thresholds are calibrated per task family: transfer validators apply a 0.1% relative tolerance, approval validators require an exact match, and swap/liquidity operations use a 5% slippage tolerance to account for AMM nondeterminism. Composite validators score the final end-state condition and apply step-efficiency decay (Section 4), reusing the corresponding atomic validator for consistency.

3.7 Score Reporting

We report three aggregate scores per model: Atomic score (sum over 62 tasks), Composite score (sum over 45 tasks), and Total score (sum over 107 tasks). We also report per task averages by normalizing each split sum by its task count. For summary statistics (e.g., pattern-level rates in Appendix B), we define a *soft pass* as scoring ≥ 60 points on a task. This differs from the stricter *Pass* counts in Table 1, which require all validator checks to be satisfied (`validation_passed=True`). On atomic tasks, soft pass can hold with partial credit below a perfect score; on composite tasks,

`validation_passed=True` can hold while the reported score falls below 60 after step-efficiency decay.

4 Evaluation Setup

We treat correctness as an end-to-end property: a submission must construct valid transaction request objects, execute successfully in a forked environment, and satisfy post-state constraints checked by validators.

4.1 Task Formalization

Atomic tasks. Let $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ denote the set of atomic tasks. Each atomic task a_i is defined as a tuple:

$$a_i = (T_i, P_i, V_i, S_i^{\max})$$

where T_i is the set of natural language templates (one is randomly selected at test time), P_i is the parameter space (numeric values are sampled uniformly within predefined intervals), V_i is the validator that checks post-execution state changes, and $S_i^{\max} = 100$ is the maximum score.

Composite tasks. A composite task C is constructed from an ordered sequence of atomic operations:

$$C = \left(a_{i_1} \to a_{i_2} \to \cdots \to a_{i_m},\ K^{\text{opt}},\ V_C\right)$$

where $a_{i_j} \in \mathcal{A}$ is the atomic operation at step j, K^{opt} is the optimal number of steps specified by the task definition, and V_C is the composite validator that checks the final state condition. The arrow notation $\to$ indicates execution order and potential data dependencies (e.g., an approval must precede a swap that requires that allowance).

4.2 Execution Environment

Evaluation runs on an Anvil fork of BSC mainnet (chain ID 56), an EVM-compatible chain. We use BSC as the instantiation; the benchmark design generalizes to other EVM chains. The runner specifies the upstream RPC endpoint. The fork block height is not pinned; each evaluation run forks at the latest block. Since all 107 tasks within a run share the same fork origin and snapshot isolation (Section 4.3) prevents cross-task state leakage, intra-run comparability is fully preserved.

Each run creates a fresh test account. The private key is generated at runtime and is never exposed to the model. Transaction signing happens in the evaluation process, separating generation from authorization and preventing the model from directly controlling keys.

Before tasks start, the account is funded with 100 BNB. The environment provisions task assets, including ERC20 tokens, LP tokens, and NFT holdings. The runner also deploys a small set of auxiliary contracts at runtime for testing.

Execution backend. All experiments use **ethers.js v6**, the de facto standard for EVM client scripting, chosen for its broad representation in LLM training corpora and as the most equitable cross-model baseline. A **viem v2** backend is also supported; switching requires only a different system configuration file with no changes to validators or scoring. Full backend details are in Appendix C.

4.3 Isolation

After environment initialization, we create a snapshot of the forked chain state. Before executing each task, the runner restores the snapshot, yielding a consistent initial state per task. This prevents cross-task interference and makes task scores comparable under identical starting conditions.

4.4 Inference and Subtask Planning

Unless otherwise specified, we use single-shot generation for atomic tasks: each model is called once per task instance to produce a complete TypeScript module. We set the decoding temperature to 0.7.

For composite tasks, we employ a multi-turn interaction protocol with explicit subtask planning. The LLM first enters a **planning phase**, where it analyzes the natural language instruction and decomposes the task into an ordered sequence of subtasks. Formally, the model outputs a plan:

$$\Pi = \langle \pi_1, \pi_2, \ldots, \pi_k \rangle$$

where each subtask $\pi_j = (t_j, c_j, p_j)$ specifies an action type t_j (e.g., `approve`, `swap`, `stake`, `query`), target contract c_j, and parameters p_j. The planning objective is to minimize the total number of steps while satisfying task constraints:

$$\min |\Pi| \quad \text{subject to} \quad \Pi \models C$$

where $\Pi \models C$ denotes that executing Π achieves the goal state specified by composite task C.

After planning, the LLM enters the **execution phase**, iterating through the planned subtasks. For

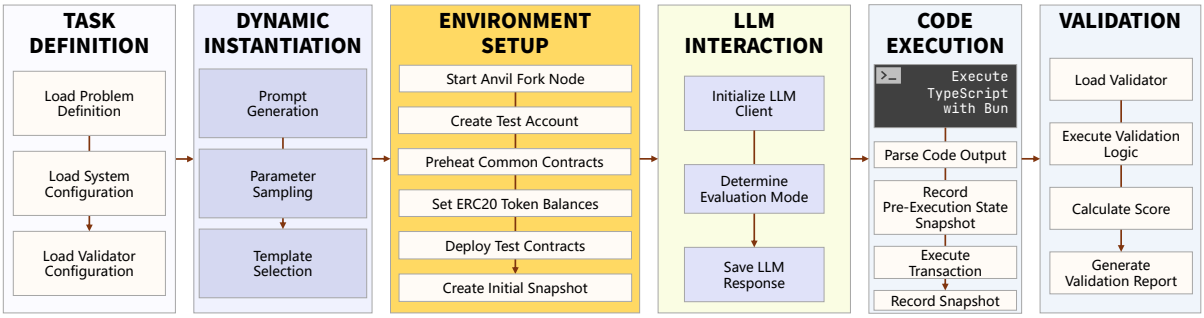

Figure 2: End-to-end evaluation pipeline of EVM-QuestBench. A natural language instruction is instantiated from a template with dynamically sampled parameters, fed to the LLM for TypeScript script generation, executed by the runner on a snapshot-isolated forked chain, and finally scored by task-specific validators against post-execution on-chain state.

each subtask, the model generates a TypeScript code block that returns a transaction request object. The runner executes the transaction on the forked chain and returns the result (success/failure, receipt, state changes) to the LLM. The model can also perform `query` actions to inspect on-chain state (balances, allowances) before executing transactions. This closed-loop interaction continues until the model signals task completion or exhausts the round budget.

4.5 Scoring

Each task has a maximum score of $S^{\max} = 100$.

Atomic scoring. Atomic tasks use task-specific validators. Each validator computes a weighted score from a set of post-execution checks $\mathcal{C}$:

$$S_{\text{atomic}} = \sum_{c \in \mathcal{C}} w_c \cdot \mathbf{1}\left[f_c(s_{\text{pre}}, s_{\text{post}})\right]$$

where w_c is the weight assigned to check c, f_c is the check function comparing pre-execution state s_{pre} and post-execution state s_{post}, and $\mathbf{1}[\cdot]$ is the indicator function. The weights satisfy $\sum_{c \in \mathcal{C}} w_c = 100$. Typical checks include transaction success (30 points), target address correctness (20 points), function signature correctness (20 points), and state-change verification through balance or allowance deltas (30 points). Where required by on-chain mechanics, validators apply tolerances to account for rounding or protocol-side price impact.

Composite scoring. Composite tasks use outcome-based scoring with step-efficiency decay. Let K^{act} denote the actual number of execution rounds (including failed attempts and retries up to the retry budget). The final score is:

$$S = S_{\text{base}} \cdot \min\left(1, \frac{K^{\text{opt}}}{K^{\text{act}}}\right)$$

where $S_{\text{base}} \in \{0, 100\}$ depends on whether the final state condition holds. This formulation rewards efficient execution: if $K^{\text{act}} \leq K^{\text{opt}}$, the model receives full credit; if $K^{\text{act}} > K^{\text{opt}}$, the score decays proportionally to the ratio of optimal to actual steps.

Aggregate scoring. Let $\mathcal{A}_{\text{bench}}$ denote the set of 62 atomic tasks and $\mathcal{C}_{\text{bench}}$ denote the set of 45 composite tasks. We report:

$$S_{\text{Atomic}} = \sum_{a \in \mathcal{A}_{\text{bench}}} S_a,$$

$$S_{\text{Composite}} = \sum_{c \in \mathcal{C}_{\text{bench}}} S_c,$$

$$S_{\text{Total}} = S_{\text{Atomic}} + S_{\text{Composite}}$$

The maximum possible scores are $S^{\max}_{\text{Atomic}} = 6{,}200$, $S^{\max}_{\text{Composite}} = 4{,}500$, and $S^{\max}_{\text{Total}} = 10{,}700$.

4.6 Artifacts

We release the benchmark codebase, task definitions, validators, and the runner. We release aggregate model scores. We do not release full per-task outputs, JSON logs, or execution traces at this time.

5 Results and Analysis

5.1 Leaderboard

We evaluate 20 models on EVM-QuestBench with 5 independent evaluation rounds per model (totaling $5 \times 107 \times 20 = 10{,}700$ task executions).

Each round uses freshly sampled numeric parameters, independent LLM calls (temperature $= 0.7$), and clean snapshot-isolated environments. Table 1 reports the mean Atomic, Composite, and Total scores across 5 rounds, along with standard deviation (SD), observed range (min, max) over 5 rounds, and coefficient of variation (CV%).

The top three models exceed 7,700 mean total points. Claude-Sonnet-4.5 achieves the highest mean total (8,236) with low variance (CV $= 2.1\%$). 14 out of 20 models exhibit CV% below 9%, demonstrating that evaluation scores are robust to the combined variance from dynamic parameter sampling, temperature-based generation, and RPC execution.

5.2 Atomic versus Composite Capability Separation

Figure 6 (Appendix B) visualizes the Atomic–Composite plane (a full ranking bar chart is in Appendix H, Figure 8). Models spread widely, indicating that single-step correctness and multi-step completion are only partially coupled.

We observe two characteristic asymmetries. Workflow-oriented models (DeepSeek-V3.2, Gemini-2.5-Flash, GPT-5.1-Mini) achieve high Composite despite weaker Atomic, suggesting stronger sequencing and end-state targeting. Precision-oriented models (Claude-Haiku-4.5, Devstral-2512) achieve high Atomic but lag on Composite, consistent with weaknesses in multi-step dependency tracking. Several code-specialized models (Qwen3-Coder-30B, Devstral-2512, Qwen3-Coder) score near zero on Composite due to repeated interface failures in multi-step workflows.

5.3 Composite Execution Quality

Table 1 (right block) reports fine-grained composite metrics averaged over 5 rounds. We analyze step efficiency ($\bar{K}_{\text{act}}$, Eff%) as primary dimensions.

Step efficiency reflects planning quality. Claude-Sonnet-4.5 leads with 88.0% efficiency and the highest composite score (3,958 avg), completing an average of 41 out of 45 tasks. DeepSeek-V3.2 achieves the lowest $\bar{K}_{\text{act}}$ (3.5) with 84.5% efficiency, indicating consistent per-task optimization. Lower-tier models average $\bar{K}_{\text{act}} > 5.9$, which compounds both failure risk and score decay.

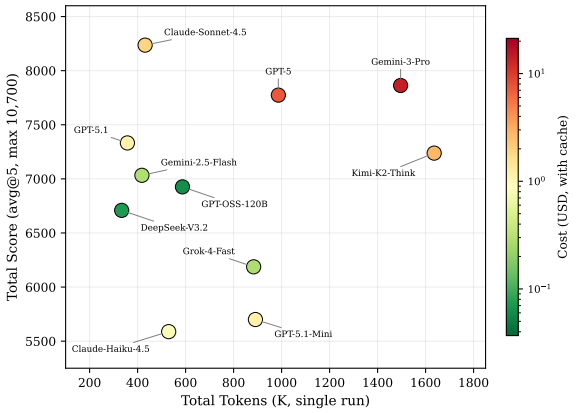

Figure 3: Total score (avg@5) versus total token usage (single run, 107 tasks). Color encodes API cost (USD, with prompt caching). Models in the upper-left quadrant achieve high scores with low token budgets.

Pass count separates tiers. Top models pass 33–41 of 45 composite tasks on average across 5 rounds. Claude-Haiku-4.5 passes only 13 tasks despite reasonable atomic performance, confirming that multi-step workflow completion requires capabilities beyond single-action precision. Four models (Devstral-2512, Qwen3-Coder-30B, Qwen3-Coder-Flash, Qwen3-Coder) pass fewer than 1 task on average, reflecting systematic interface or planning failures in multi-step workflows.

Interface compliance. All 20 models produce syntactically valid TypeScript modules that the runner can execute on atomic tasks. However, three code-specialized models (Qwen3-Coder-30B, Devstral-2512, Qwen3-Coder) score near zero on composite tasks due to repeated schema errors such as missing `ethers` imports or incorrect module structure in multi-step contexts. We retain these models in the leaderboard to document the full spectrum of interface-level and capability-level failures.

5.4 Token Usage and Cost Efficiency

Figure 3 plots total token consumption (single run, 107 tasks) against benchmark score, with color encoding per-run API cost under OpenRouter cache-read pricing.

A full 107-task run costs as little as \$0.06–\$0.29 for standard models (e.g., GPT-OSS-120B at \$0.06, Gemini-2.5-Flash at \$0.29), consuming under 420K tokens. Thinking-enabled models are substantially more expensive (up to \$14.16 for Gemini-3-Pro) due to chain-of-thought token overhead, yet do not consistently outperform efficient

#	Model	Atom	Comp	Total	SD	CV%	Range [min, max]	Pass_a	Pass_c	$\bar{K}$	Eff%
1	Claude-Sonnet-4.5	4278	3958	8236	174	2.1	[8017, 8403]	35.0	41.0	3.8	88.0
2	Gemini-3-Pro	4302	3561	7863	267	3.4	[7487, 8195]	36.6	37.6	4.3	79.7
3	GPT-5	4128	3646	7774	407	5.2	[7241, 8121]	31.6	38.0	3.6	83.8
4	GPT-5.1	3715	3617	7332	144	2.0	[7199, 7542]	28.0	38.0	4.1	80.4
5	Kimi-K2-Thinking	3605	3633	7238	207	2.9	[6928, 7498]	29.2	33.6	3.6	84.2
6	Gemini-2.5-Flash	3265	3768	7033	127	1.8	[6837, 7176]	22.0	38.6	4.1	83.7
7	GPT-OSS-120B	3458	3468	6926	436	6.3	[6430, 7633]	26.0	35.2	3.9	82.6
8	DeepSeek-V3.2	3071	3638	6709	384	5.7	[6074, 7088]	21.6	38.2	3.5	84.5
9	Grok-4-Fast	2980	3207	6187	224	3.6	[5913, 6384]	20.8	32.0	4.4	76.4
10	Qwen3-235B	3609	2115	5724	630	11.0	[4735, 6300]	28.0	11.0	5.8	62.6
11	GPT-5.1-Mini	2510	3190	5700	510	8.9	[5139, 6298]	19.4	31.0	4.7	73.6
12	Qwen3-Max	2775	2883	5658	229	4.1	[5388, 5956]	18.8	25.8	4.9	71.5
13	Claude-Haiku-4.5	3221	2367	5588	316	5.7	[5208, 5899]	24.8	13.0	5.9	60.7
14	Mimo-V2-Flash	3189	1723	4912	218	4.4	[4659, 5160]	25.0	9.0	6.2	56.9
15	GLM-4.6	2881	1971	4852	295	6.1	[4522, 5289]	21.2	10.2	6.0	58.3
16	Qwen3-30B	1554	2011	3565	398	11.2	[3212, 4214]	10.8	8.4	6.1	58.8
17	Devstral-2512	2892	22	2914	303	10.4	[2460, 3267]	20.6	0.0	6.8	50.8
18	Qwen3-Coder	1801	160	1961	291	14.8	[1600, 2265]	13.2	0.4	6.7	53.3
19	Qwen3-Coder-Flash	223	37	260	212	81.4	[40, 538]	1.6	0.2	6.8	51.1
20	Qwen3-Coder-30B	140	5	145	61	42.0	[70, 240]	0.6	0.0	6.8	50.6

Table 1: Leaderboard and execution quality (avg@5). Left block: Atom = Atomic (62 tasks, max 6,200), Comp = Composite (45 tasks, max 4,500), Total = 107 tasks (max 10,700); SD and CV% are over Total; Range = [min, max] over 5 rounds. Spearman's $\rho = 0.960$ avg across round pairs (all $p < 0.001$, range: 0.940–0.983). Right block: Pass_a = atomic tasks passed (of 62), Pass_c = composite tasks passed (of 45), $\bar{K}$ = mean K_{act}, Eff% = step efficiency. Pass is defined as `validation_passed=True` (all checks satisfied). Atom, Comp, and Total are rounded independently after averaging; Comp is adjusted by ± 1 for three models so that Atom + Comp = Total.

non-thinking models. Detailed per-model cost and caching breakdowns are in Appendix L. Workflow failure analysis by pattern is in Appendix H.

6 Conclusion

We presented EVM-QuestBench, an execution-grounded benchmark for natural language transaction script generation on EVM-compatible chains, instantiated on BNB Smart Chain (chain ID 56). The benchmark contains 107 tasks with Atomic and Composite splits. Model-generated TypeScript scripts are executed in a snapshot-isolated forked environment and scored by validators against post-state constraints; composite tasks additionally apply a step-efficiency factor.

We evaluated 20 models under a unified runner with 5 independent rounds each. Results reveal a persistent capability gap between single-transaction precision and multi-step workflow completion. Thanks to its dynamic parameterization design, a full evaluation run requires as few as 334K tokens (as low as $0.06), making large-scale, multi-round model comparison practically affordable.

EVM-QuestBench offers a standardized protocol for studying execution-grounded behavior in on-chain automation. We have already ported the same architecture to Solana, demonstrating the portability of the atomic/composite paradigm across heterogeneous blockchain ecosystems. Future work will expand task coverage, incorporate richer security checks for transaction intent and side effects, and evaluate LLMs' ability to generate task definitions autonomously.

7 Limitations

Execution stability. Because EVM-QuestBench is execution-grounded, scores inherit potential instability from RPC connectivity, fork performance, and provider availability. Although this effect was negligible in our experiments, snapshot isolation and composite retries reduce but cannot fully eliminate such instability. Because the fork block height is not pinned across runs, cross-run comparability may be affected by evolving on-chain state; we therefore report multi-round averages to mitigate this effect.

Number of evaluation rounds. We conduct 5 independent evaluation rounds per model and report mean scores with standard deviations and observed ranges (Table 1). While 5 rounds provide reasonable statistical power for tier-level separation, some adjacent-rank differences remain difficult to distinguish. Additional rounds would provide more stable variance estimates and enable finer-grained ranking distinctions.

Task coverage. The current benchmark contains 107 tasks, which covers a representative range of on-chain operations but remains limited in scale. We encourage community contributions to expand task coverage across protocols, chains, and operation types, and plan to maintain EVM-QuestBench as a living benchmark with ongoing community participation.

8 Ethics Statement

All experiments run on a locally forked copy of the blockchain; no real funds are transferred and no live network state is modified. The benchmark does not collect or process personal data. Model evaluation results are reported objectively without selective omission.

References

Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Noam Shazeer, and Malte Taropa. 2023. PaLM 2 technical report. *arXiv preprint arXiv:2305.10403*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, and Quoc Le. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, and Jared Kaplan. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

Mohammad Abdullah Matin Khan, M Saiful Bari, Do Xuan Long, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of ACL*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, and 3 others. 2023. AgentBench: Evaluating LLMs as agents. *arXiv preprint arXiv:2308.03688*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

OpenAI. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of ACL*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *arXiv preprint arXiv:2307.16789*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Solana Foundation. 2025. Introducing solana bench: How well can LLMs build complex transactions? Solana News.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023. Execution-based evaluation for open-domain code generation. In *Findings of EMNLP*.

Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Hari Sundaram, and Shuiguang Deng. 2024. CodeScope: An execution-based multilingual multitask multidimensional benchmark for evaluating LLMs on code understanding and generation. In *Proceedings of ACL*.

A Background: EVM Concepts and What EVM-QuestBench Measures

This section provides a self-contained introduction to key concepts from the EVM ecosystem for readers unfamiliar with blockchain technology, and explains concretely what EVM-QuestBench is designed to evaluate.

A.1 Core EVM Terminology

Ethereum Virtual Machine (EVM). The EVM is a sandboxed, stack-based virtual machine that executes smart contract bytecode on a distributed network of nodes. Every EVM-compatible chain—including Ethereum mainnet, BNB Smart Chain (BSC), Polygon, and Avalanche—runs the same instruction set, so programs written for one chain can generally be ported to others with minimal changes. EVM-QuestBench is instantiated on BSC (chain ID 56) but the design generalizes to any EVM-compatible chain.

Smart Contract. A smart contract is a piece of code deployed at a fixed address on the blockchain. Once deployed, it can be called by anyone by sending a transaction that encodes the function name and its arguments as raw bytes (*calldata*). The contract's logic executes deterministically on every node, and the resulting state changes (e.g., token balances, ownership records) are permanently recorded on-chain. Errors in calldata construction or parameter encoding commonly cause transactions to revert.

Externally Owned Account (EOA). An EOA is a user-controlled wallet identified by a public/private key pair. All transactions must originate from an EOA, which signs the transaction with its private key to authorize it. EVM-QuestBench creates a fresh test EOA for each evaluation run; the model generates unsigned transaction payloads that the runner signs and submits.

Transaction and Gas. A transaction is an authenticated message that transfers value or invokes a smart contract. Each transaction consumes *gas*, a unit of computational cost paid by the sender in native tokens (BNB on BSC). Gas limits prevent infinite loops and prioritize efficient code. Incorrectly estimated gas limits can cause transactions to revert or fail silently.

BNB Smart Chain (BSC). BSC is an EVM-compatible blockchain operated by Binance with fast block times ($\sim$3s) and low fees. It supports the same tooling as Ethereum. EVM-QuestBench forks BSC mainnet via Anvil to provide a realistic yet isolated on-chain environment.

Token Standards.

- **ERC-20**: Fungible token standard. Each contract maintains a balance mapping; transfers are authorized via `approve`/`transferFrom` or direct `transfer`. Amounts must be expressed in the token's native unit (e.g., 10^{18} for 18-decimal tokens).
- **ERC-721**: Non-fungible token (NFT) standard. Each token ID is unique and owned by exactly one address.
- **ERC-1155**: Multi-token standard that supports both fungible and non-fungible assets in a single contract.

DeFi, DEX, and AMM. *Decentralized Finance* (DeFi) refers to financial services built as smart contracts. A *Decentralized Exchange* (DEX) allows token swaps without a centralized intermediary. Most DEXes use an *Automated Market Maker* (AMM) model, where liquidity providers deposit token pairs into pools, and prices are determined by the ratio of reserves. PancakeSwap—the primary DEX used in EVM-QuestBench—is an AMM DEX on BSC. Swap transactions must encode the trade path, deadline, and minimum output amount; incorrect parameters cause reverts or financial loss.

Liquidity Provision and Staking. Liquidity providers deposit two tokens into an AMM pool and receive *LP tokens* representing their share. LP tokens can be deposited into staking contracts to earn rewards. These workflows require multiple sequential transactions (approve $\rightarrow$ add liquidity $\rightarrow$ stake), making them natural candidates for composite task evaluation.

A.2 What EVM-QuestBench Measures

EVM-QuestBench evaluates whether a language model can translate a natural language instruction (e.g., "Swap 0.1 BNB for USDT and stake the LP tokens") into a correct, executable client-side TypeScript module. The module must:

1. **Correctly identify the target contract and function.** The model must select the right protocol (e.g., PancakeSwap Router vs. staking pool), the right function signature, and encode calldata that satisfies the ABI.
2. **Handle chain-specific units.** Token amounts must be converted from human-readable form to on-chain representation (e.g., 0.1×10^{18} wei for 18-decimal tokens). Swaps require slippage-tolerant minimum output values.
3. **Satisfy protocol prerequisites.** Many operations require a prior approval transaction (ERC-20 `approve`) before the main action can execute. The model must identify and include these dependencies.

4. **Propagate parameters across steps.** In multi-step workflows, outputs from earlier steps (e.g., LP token amounts received from liquidity addition) feed into subsequent steps (e.g., staking). The model must track and propagate these values correctly.

The benchmark does *not* evaluate contract deployment or Solidity code generation. It specifically targets the client-side scripting layer: producing unsigned transaction payloads that a standardized runner can sign and broadcast. Validators then check the resulting on-chain state—not the code itself—ensuring that evaluation measures functional correctness under execution.

B Figures

This section contains benchmark composition and result figures referenced from the main text.

B.1 Benchmark Composition

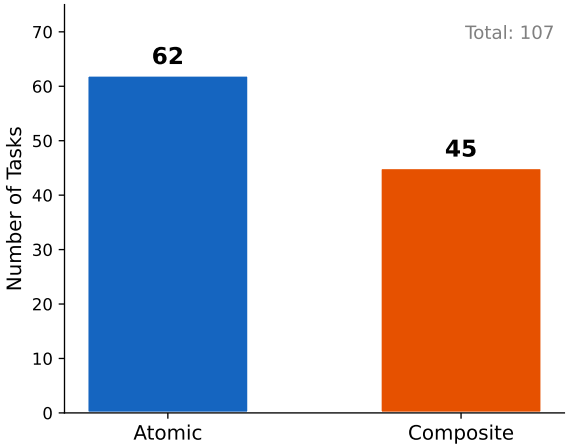

Figure 4: Task split in EVM-QuestBench: 62 atomic tasks and 45 composite tasks (107 total).

B.2 Results Figures

C Reproducibility

This appendix summarizes the setup required to reproduce EVM-QuestBench runs and the experimental settings that affect run-to-run variance. For strict reproducibility, record the fork block height, model sampling parameters, and the task parameter random seed.

C.1 Environment and Dependencies

Experiments require a local EVM mainnet fork and a TypeScript runner.

- Python 3.10 or newer

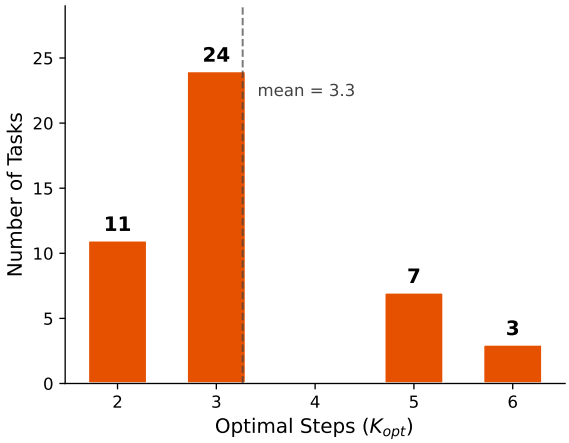

Figure 5: Distribution of optimal steps (K_{opt}) for composite tasks. Most workflows require 3 steps (53.3%), with a mean of 3.27.

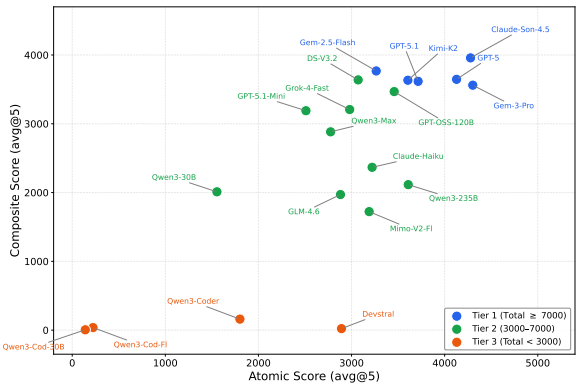

Figure 6: Atomic score versus Composite score (avg@5). Each point is a model.

- Node.js 18 or newer (Bun 1.0 or newer is also supported)
- Foundry with Anvil available in PATH

C.2 Key Experimental Settings

Table 2 lists the settings that most often explain score variance across reruns.

Item	Value
Chain	EVM-compatible mainnet fork
Runtime	TypeScript runner using ethers.js v6 (primary; viem v2 also supported)
Atomic output	One module exporting `executeSkill` that returns one `TransactionRequest`
Composite output	Iterative calls; each round returns either one tx module or a control JSON (query, error, submit)
Composite rounds	Bounded by task `optimal_steps` and a multiplier such as `max_rounds_multiplier`
Non determinism	Fork block height, model sampling parameters, random parameter sampling when seed is not fixed

Table 2: Reproducibility relevant settings.

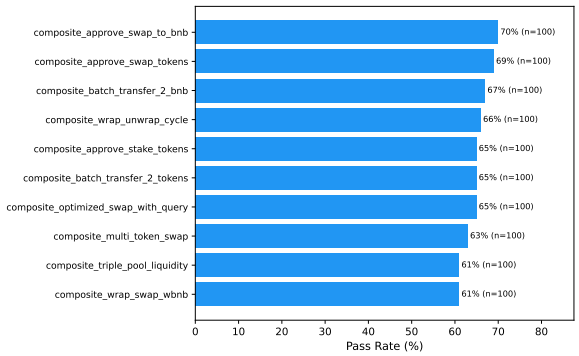

Figure 7: Composite workflow difficulty by pattern. Soft pass is defined as score ≥ 60 (Section 3.7; stricter Pass in Table 1).

D Prompt and Interface Specification

This section documents the runner interface and response validity rules used by the evaluator. The goal is to make the execution contract between model and runner explicit.

D.1 Runner Interface

Each model outputs a TypeScript module exporting an entry function. The runner provides `providerUrl`, the agent EOA address, and a contract address map for the local fork.

```
export async function executeSkill(
  providerUrl: string,
  agentAddress: string,
  deployedContracts: Record<string, string>
): Promise<Record<string, unknown>> {
  const tx: Record<string, unknown> = {
    to: "0x...",
    data: "0x..."
  };
  return tx;
}
```

D.2 Atomic and Composite Prompt Roles (Schematic)

Atomic tasks require a single transaction that satisfies post-execution checks. Composite tasks allow multi-round interaction and apply step-efficiency decay based on the number of rounds consumed.

```
Atomic role (schematic)
Produce TypeScript in a code block.
Return exactly one transaction request object.
Use task provided parameters and addresses.

Composite role (schematic)
You may query chain state before executing.
Each round returns either one tx module or a
    JSON control message.
Completion is signaled with {"submit": true}.
Fewer rounds yield higher score via step-
    efficiency decay.
```

D.3 Schema Invalid Rules

A response is marked `schema_invalid` if it cannot be executed under the runner contract.

1. Missing exported `executeSkill`
2. Function signature mismatch
3. Return value is not a transaction-like object
4. Missing required `to` field
5. Serialization failure under ethers.js
6. No valid TypeScript code block when code is required
7. Control JSON is not parseable in composite control rounds

E Task Definition Schema

This section summarizes task fields that are most relevant for reproduction and error diagnosis.

E.1 Atomic Task Fields

Atomic tasks specify one on-chain action and are validated by post-execution constraints.

Field	Type	Description
`id`	string	Unique task identifier
`category`, `subcategory`	string	Task family tags for coverage analysis
`difficulty`	string	easy, easy-medium, medium, hard
`natural_language_templates`	string[]	Instruction templates for prompts
`parameters`	object	Typed params with sampling ranges
`validation`	object	Post-execution checks and weights

Table 3: Atomic task schema summary.

E.2 Composite Task Fields

Composite tasks add a workflow template and a scoring strategy that emphasizes end-to-end completion.

Field	Type	Description
`composite_structure`	object	Workflow motif and step sequence
`atomic_operations`	array	Ordered atomic steps by `atomic_id`
`optimal_steps`	int	K_{opt} for step-efficiency decay
`max_rounds_multiplier`	int	Upper bound on interaction rounds
`scoring_strategy`	object	Validator config for end-state

Table 4: Composite task schema additions.

F NL Template Difficulty Scoring

To control for potential template-wording bias, we implemented a difficulty scoring system for all 373 natural language templates. Three SOTA models (Claude Opus 4.6, GPT-5.4, and Gemini 3.1 Pro) independently rated each template on a 1–5 clarity scale (1 = maximal precision, 5 = maximal ambiguity), and ratings were averaged. Templates are

classified into three tiers: *precise* (mean ≤ 2.0), *moderate* (> 2.0 and ≤ 3.5), and *vague* (> 3.5). Scores are stored in `nl_template_scores.json`.

The distribution is: 336 precise (90.1%), 37 moderate (9.9%), and 0 vague. This confirms that the template pool is consistently unambiguous across all task types.

At evaluation time, the `-nl-difficulty` flag supports four selection modes: `random` (default, no filtering), `precise`, `moderate`, and `vague`. All experiments in this paper use `random` mode. The difficulty API is available for controlled ablation studies on instruction clarity.

G Validators and Scoring

EVM-QuestBench uses validator-based post-execution scoring rather than reference code matching. Validators check receipts and post state signals such as balance deltas, allowances, and protocol-specific outcomes.

G.1 Validator Families

Table 5 summarizes common validator families and the constraints they evaluate.

G.2 Composite Step Efficiency Decay

Composite tasks apply an outcome-based base score and multiply it by a step-efficiency factor.

$$\text{Score}_{\text{final}} = \text{Score}_{\text{base}} \times \min\left(1.0, \frac{K_{\text{opt}}}{K_{\text{act}}}\right). \quad (1)$$

Here K_{opt} is the task-defined optimal step count and K_{act} is the number of executed rounds, including retries and query rounds.

G.3 Validator Tolerance Sensitivity

To assess whether the tolerance thresholds influence evaluation outcomes, we analyzed the deviation distribution across 5 rounds $\times$ 20 models $\times$ 107 tasks (10,700 question instances). Table 7 reports the results.

We varied the tolerance multiplier from $0.5\times$ to $3\times$ and re-scored all instances. Under all multiplier settings, all 20 models produce identical scores and rankings (Spearman's $\rho = 1.000$, $\Delta = 0\%$). This confirms that tolerances function as safety margins and do not influence reported scores or rankings. The bimodal nature of model outputs — either exactly correct or catastrophically wrong (transaction revert, schema error) — means no question instances fall in the "close but off by a few percent" zone where tolerance thresholds would matter.

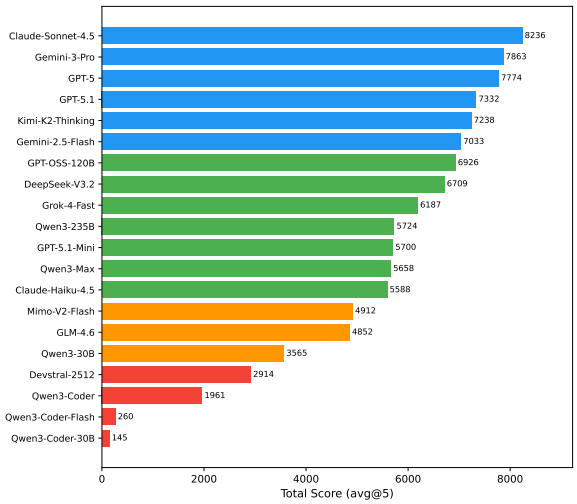

Figure 8: Model ranking by total score (avg@5, 20 models).

H Additional Analysis Figures

This section provides supporting plots used to interpret difficulty and failure modes.

H.1 Total Score Ranking

Figure 8 shows the total score ranking across all 20 models (avg@5).

H.2 Composite Pattern Difficulty

Figure 9 reports soft-pass rates by composite workflow pattern. The hardest patterns are multi-stage DeFi workflows combining liquidity and staking, as well as batch approval patterns. These concentrate three failure surfaces: (i) prerequisite correctness—the model must issue all required approvals before a protocol action; (ii) cross-step parameter consistency—output values from earlier steps (e.g., LP tokens received) must be correctly propagated to later steps; and (iii) execution robustness under multiple sequential transactions, where any single revert terminates the workflow and yields a zero base score. Lower pass rates indicate higher coordination burden across steps, including prerequisite approvals and parameter propagation.

H.3 Step Overhead Distribution

Figure 10 shows the distribution of step overhead $\Delta = K_{\text{act}} - K_{\text{opt}}$ aggregated across composite runs. Positive overhead indicates extra rounds consumed

Family	Typical tolerance	Examples of checks
Native and ERC20 transfer	0.1% relative	receipt success, recipient, amount, balance delta
Approval	exact match (==)	spender address, allowance target
DEX swap	slippage tolerance	router, path, amountIn and minOut, output delta
Liquidity	small relative tolerance	token amounts, LP minted, pool state updates
Staking and farming	small relative tolerance	pool identifier, staked balance delta, reward signals
Queries	numeric tolerance	returned value accuracy, output format correctness

Table 5: Validator families summary.

K_{opt}	K_{act}	Decay	Base score 100
3	2	1.00	100
3	3	1.00	100
3	4	0.75	75
3	6	0.50	50

Table 6: Step efficiency decay examples.

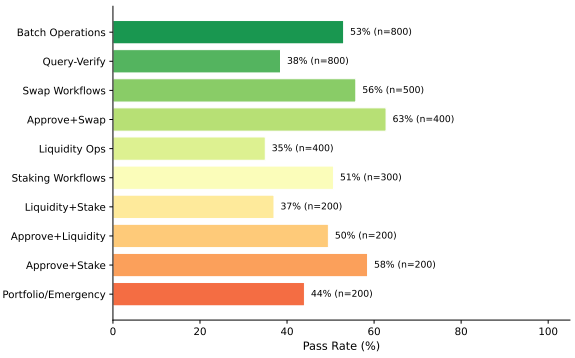

Figure 9: Soft-pass rate by composite workflow pattern for the ten most frequent patterns (Section 3.7).

by redundant queries, retries, or incorrect intermediate actions.

H.4 Atomic Subcategory Difficulty

Figure 11 summarizes atomic soft-pass rates by subcategory. This view localizes which single transaction primitives are brittle, such as query-heavy tasks or protocol-specific calls with strict calldata requirements.

I Full Evaluation Tables

This section lists full model coverage and step overhead statistics used in the main analysis.

I.1 Quadrant Summary

Table 8 summarizes the four capability profiles based on AtomicAvg and CompositeAvg medians (50.5 and 67.5 respectively).

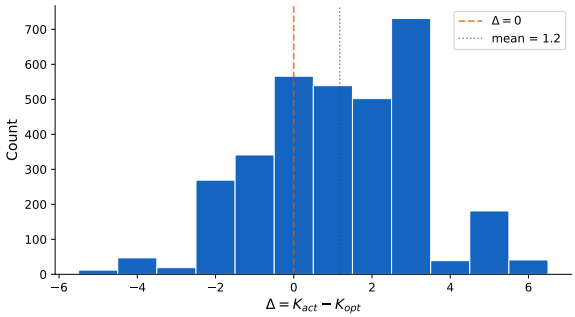

Figure 10: Distribution of step overhead $\Delta = K_{\text{act}} - K_{\text{opt}}$.

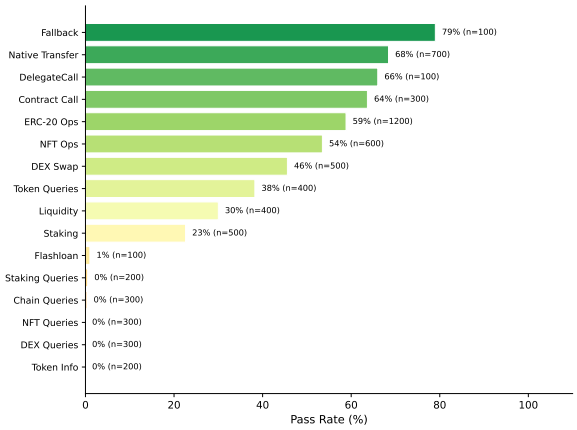

Figure 11: Soft-pass rate by atomic subcategory (Section 3.7).

I.2 Model Coverage and Quadrant Classification

Table 9 reports coverage, normalized averages (avg@5), total score, and the quadrant label based on AtomicAvg and CompositeAvg medians.

I.3 Step Overhead and Decay Impact Summary

Table 10 reports overhead statistics (avg@5) for models with non-trivial composite scores. The overhead statistic is $\Delta = K_{\text{act}} - K_{\text{opt}}$.

Outcome	Count	%
Score > 0, deviation $= 0\%$ (exact match)	7,013	65.5%
Score $= 0$, execution failure / schema error	1,931	18.0%
Score $= 0$, transaction revert	493	4.6%
Score $= 0$, other failure	1,191	11.1%
Score > 0, deviation $> 0\%$ (float precision)	72	0.7%

Table 7: Actual deviation distribution across all 10,700 evaluated question instances (20 models $\times$ 5 rounds $\times$ 107 tasks). Non-zero deviations are all $\sim 10^{-11}\%$ (floating-point precision), approximately $10^7\times$ smaller than the strictest tolerance.

Profile	Count	Interpretation
High-High	7	strong precision and workflows
High-Low	3	strong precision, weaker workflows
Low-High	3	weaker precision, stronger workflows
Low-Low	3	weaker on both splits
Atomic-Only	4	composite split near zero

Table 8: Quadrant summary based on AtomicAvg and CompositeAvg medians.

J End-to-End Walkthrough

We illustrate the complete evaluation pipeline with one atomic task: `bnb_transfer_percentage`.

Step 1: Parameter Instantiation. The runner selects a natural language template and samples numeric parameters:

- **Template selected:** "Transfer {percentage}% of my BNB balance to {recipient}."
- **Parameters sampled:** `percentage` $= 15$, `recipient` $=$ `0xA1b2...C3d4` (randomly generated).
- **Final instruction:** "Transfer 15% of my BNB balance to 0xA1b2...C3d4."

Step 2: Script Generation. The model receives the instruction along with system context (chain ID, available libraries, account address, contract ABIs). It generates a TypeScript module that:

1. Queries the current BNB balance via `provider.getBalance(account)`.
2. Computes `amount = balance * 15n / 100n`.
3. Returns a transaction request: `{to: recipient, value: amount}`.

Step 3: Execution. The runner signs and submits the transaction on the forked chain. The transaction executes successfully (receipt status $= 1$). The runner records pre-execution and post-execution balances for both the sender and recipient.

Step 4: Validation. The validator for this task (`bnb_transfer_percentage`) performs four checks:

1. **Transaction Success** (30 pts): receipt status $= 1$. ✓
2. **Recipient Correct** (20 pts): `tx.to` matches the sampled recipient. ✓
3. **Transfer Amount** (20 pts): actual transferred amount within 0.1% of expected (balance $\times$ 15%). ✓
4. **Balance Change** (30 pts): sender balance decreased by amount + gas; recipient balance increased by amount, both within 0.1% tolerance. ✓

Final score: $30 + 20 + 20 + 30 = 100$ out of 100.

Key design points. The expected transfer amount is computed dynamically from the fork state (not hardcoded), so the ground truth is always consistent with the execution environment. The tolerance on balance change (0.1%) absorbs floating-point precision differences without admitting materially incorrect transfers.

K Case Studies

This section provides representative workflows that illustrate common failure modes. The goal is to highlight failure points such as parameter propagation, prerequisite handling, and protocol sequencing.

K.1 Case 1: Complex Multi-step DeFi Workflow

Task: `composite_complete_swap_stake_workflow`.
Workflow: query, approve, swap, add liquidity, stake, verify.
Optimal steps: 6. **Pass rate:** 59%.

This task combines prerequisite checks with multiple protocol interactions. Failures often arise from incorrect swap path or slippage, mismatched token ratios during liquidity provision, or staking to an

Model	Atomic	Composite	AtomicAvg	CompAvg	Total	Quadrant
Claude-Sonnet-4.5	62/62	45/45	69.0	88.0	8235.8	High-High
Gemini-3-Pro	62/62	45/45	69.4	79.1	7863.0	High-High
GPT-5	62/62	45/45	66.6	81.0	7773.7	High-High
GPT-5.1	62/62	45/45	59.9	80.4	7331.7	High-High
Kimi-K2-Thinking	62/62	45/45	58.1	80.7	7238.3	High-High
Gemini-2.5-Flash	62/62	45/45	52.7	83.7	7033.4	High-High
GPT-OSS-120B	62/62	45/45	55.8	77.1	6925.7	High-High
Qwen3-235B	62/62	45/45	58.2	47.0	5724.5	High-Low
Claude-Haiku-4.5	62/62	45/45	52.0	52.6	5588.4	High-Low
Mimo-V2-Flash	62/62	45/45	51.4	38.3	4912.3	High-Low
DeepSeek-V3.2	62/62	45/45	49.5	80.9	6709.2	Low-High
Grok-4-Fast	62/62	45/45	48.1	71.3	6187.0	Low-High
GPT-5.1-Mini	62/62	45/45	40.5	70.9	5700.0	Low-High
Qwen3-Max	62/62	45/45	44.8	64.1	5658.5	Low-Low
GLM-4.6	62/62	45/45	46.5	43.8	4852.0	Low-Low
Qwen3-30B	62/62	45/45	25.1	44.7	3565.5	Low-Low
Devstral-2512	62/62	45/45	46.7	0.5	2913.5	Atomic-Only
Qwen3-Coder	62/62	45/45	29.0	3.6	1961.0	Atomic-Only
Qwen3-Coder-Flash	62/62	45/45	3.6	0.8	260.5	Atomic-Only
Qwen3-Coder-30B	62/62	45/45	2.3	0.1	145.0	Atomic-Only

Table 9: Model coverage and quadrant classification (avg@5). Averages are normalized to the 0–100 scale. Medians: AtomicAvg = 50.5, CompositeAvg = 67.5.

Model	Mean Δ	Over-opt (%)	P90 Δ	Comp. score
DeepSeek-V3.2	+0.05	44	3	3638.4
Kimi-K2-Thinking	+0.17	42	3	3633.3
GPT-5	+0.19	48	3	3645.7
Claude-Sonnet-4.5	+0.34	36	2	3958.0
GPT-OSS-120B	+0.41	47	3	3468.1
Gemini-2.5-Flash	+0.67	45	3	3768.0
GPT-5.1	+0.70	56	3	3617.1
Gemini-3-Pro	+0.83	59	3	3561.0
Grok-4-Fast	+1.02	62	3	3206.8
GPT-5.1-Mini	+1.24	67	3	3189.8
Qwen3-Max	+1.35	63	3	2883.5
Qwen3-Coder	+2.36	86	4	160.0
Qwen3-235B	+2.47	84	5	2115.9
Claude-Haiku-4.5	+2.51	88	5	2367.4
GLM-4.6	+2.55	89	5	1971.0
Mimo-V2-Flash	+2.76	92	5	1723.1
Qwen3-30B	+2.88	92	5	2011.5

Table 10: Step overhead and decay impact summary (avg@5). Models with near-zero composite scores (Devstral-2512, Qwen3-Coder-Flash, Qwen3-Coder-30B) are omitted.

incorrect pool. Models that succeed typically keep parameters consistent across all steps and avoid redundant query rounds that increase K_{act}.

K.2 Case 2: Batch Approval Workflow

Task: `composite_batch_approve_2_tokens`.
Workflow: approve token A, approve token B, execute the downstream action.
Optimal steps: 3. **Pass rate:** 56%.

This pattern stresses multi-transaction sequencing and nonce handling. Common failures include nonce conflicts under rapid submissions, gas estimation errors for back-to-back approvals, and incomplete approvals that allow only one token to proceed.

K.3 Case 3: Natural Language Misinterpretation

Task: `bnb_transfer_percentage`.
Instruction: "Transfer 15% of my BNB balance to 0xA1b2...C3d4."
Failure mode: The model interprets "15%" as a fixed amount of 15 BNB rather than 15% of the current balance. This produces a correct transaction that executes successfully but transfers the wrong amount — the validator detects a deviation exceeding the 0.1% tolerance on the Transfer Amount check.
Affected models: Observed in lower-tier models that lack numeric reasoning about percentage-based instructions.

K.4 Case 4: Planning-Level Failure

Task: `composite_approve_and_swap`.
Instruction: "Swap 0.05 BNB for USDT on PancakeSwap."
Failure mode: The model directly calls the PancakeSwap Router's `swapExactETHForTokens` without first checking whether the WBNB→USDT path exists or setting appropriate slippage. The transaction reverts due to insufficient output amount. Some models omit the prerequisite approval step entirely when the swap requires an ERC-20 input token.
Impact: The model scores 0 on this task. Step-efficiency decay is irrelevant since the base score is 0.

K.5 Case 5: Partial Execution in Multi-Step Workflows

Task: `composite_complete_swap_stake_workflow`.
Instruction: "Swap BNB for CAKE, add CAKE-BNB liquidity, then stake the LP tokens."
Failure mode: The model successfully completes the swap and liquidity addition but fails on the staking step due to an incorrect staking pool address or mismatched LP token. The validator checks the final state (LP tokens staked) and assigns a base score of 0 despite partial progress.
Observation: This pattern accounts for the majority of composite failures among mid-tier models, where individual actions succeed but end-to-end workflow orchestration breaks down.

L Token Usage and Cost Analysis

Figure 3 (Section 5) plots total token consumption against benchmark score for a single 107-task run. This appendix provides the detailed breakdown.

Token-efficient models. Token budgets vary by over $4\times$ across the top-11 models. Claude-Sonnet-4.5 achieves the highest score (8,236) while consuming only 431K tokens per run, making it both the most accurate and one of the most token-efficient models. Other efficient models include DeepSeek-V3.2 (334K tokens, \$0.07), GPT-5.1 (358K, \$1.15), and Gemini-2.5-Flash (418K, \$0.29), all scoring above 6,700 with fewer than 420K tokens per run.

Thinking-enabled models. Thinking-enabled models allocate substantially more output tokens to chain-of-thought reasoning: Kimi-K2-Thinking (1,636K, \$2.74), Gemini-3-Pro (1,496K, \$14.16), and GPT-5 (986K, \$7.48) each exceed 900K tokens, with output tokens dominating their budgets (82–93% of total).

Cost range and caching. Cost spans three orders of magnitude. GPT-OSS-120B achieves rank 7 at just \$0.06 per run, making it $230\times$ cheaper than Gemini-3-Pro (rank 2, \$14.16) despite only a 12% score gap. Claude-Sonnet-4.5 incurs \$1.91 per run due to its premium output pricing (\$15/M), yet its low token count keeps it far below the thinking models in absolute cost. Prompt caching reduces input costs by up to 30% for models with high cache-read ratios (Claude, Gemini), though output-heavy thinking models benefit minimally (2–4% savings) since caching applies only to input tokens.