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

Anonymous ACL submission

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 and find large performance gaps, with split scores revealing persistent asymmetry between single-action precision and multi-step workflow completion. Code: `https://anonymous.4open.science/r/bsc_quest_bench-A9CF/`.

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 Web2/Web3 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 study natural language transaction script generation on EVM-compatible chains. The target output is a client-side TypeScript module that composes calls to deployed contracts. The module returns unsigned transaction payloads. A standardized runner signs and executes the requests on a forked chain. Validators check receipts and post-state constraints. This design measures end-to-end correctness under execution.

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.

We evaluate 20 models under a unified protocol. 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.
- 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 split-level analysis that separates single-action precision from multi-step workflow completion.

2 Related Work

Execution-based evaluation for code generation. A major limitation of reference-based code evaluation is that surface overlap can reward outputs that resemble references but fail functionally. 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 and multi-step interaction. SWE-bench evaluates issue resolution by applying patches to real repositories and validating with test suites (Jimenez et al., 2024), emphasizing repository-scale context and cross-file edits. Separately, agent-style benchmarks study multi-step tool use and action sequencing in interactive environments (Qin et al., 2023; Liu et al., 2023a). These lines are complementary, but they typically do not target blockchain-specific constraints such as transaction construction, revert conditions, strict unit handling, or protocol-imposed prerequisites.

Blockchain-specific evaluation and our positioning. Blockchain interaction introduces shared mutable state, protocol prerequisites, and revert risk that are absent in standard code tasks. Prior blockchain benchmarks (e.g., Solana-focused transaction evaluations) (Solana Foundation, 2025) do not explicitly 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. Tasks are specified declaratively as JSON with reusable validator components, enabling new tasks to be added with minimal engineering effort while preserving comparable 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).

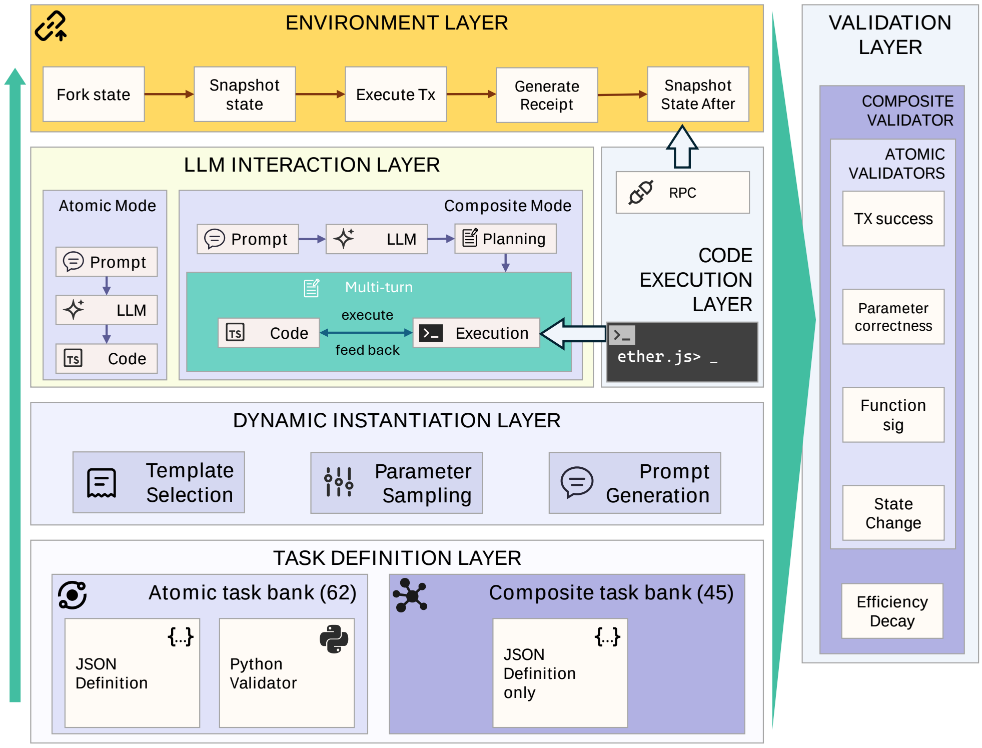

Figure 1: EVM-QuestBench evaluation architecture. Tasks are defined as JSON specs, dynamically instantiated into concrete instances, executed through a fixed TypeScript interface, and scored by validator-driven post-state checks. Composite tasks additionally apply step-efficiency decay.

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

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 ETH 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).

3.3 Benchmark Composition

EVM-QuestBench contains 107 tasks (62 atomic, 45 composite) with a maximum total score of 10,700. Figure 2 shows split sizes and compos-

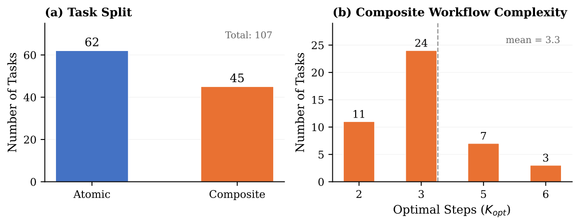

Figure 2: Task split and composite workflow complexity in EVM-QuestBench. (a) Atomic and composite task counts. (b) Distribution of optimal steps for composite tasks.

ite 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, ERC20 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`, `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 ERC20 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 that render a user goal as an instruction, together with a parameter schema that binds slots in the template (e.g., token symbols, contract keys, recipient addresses, and amounts). These templates are generated by an LLM during benchmark development to ensure diverse and realistic phrasing. Atomic tasks include 3–5 templates (mean 3.97, median 4), while composite tasks include 2–4 templates (mean 2.82, median 3).

At evaluation time, the runner randomly selects one template from the candidate pool and samples numeric parameter values uniformly within predefined ranges specified in the task configuration. For both atomic and composite tasks, all numeric values (e.g., transfer amounts, token quantities, percentages) are dynamically generated within configured intervals rather than fixed. The runner injects these sampled values into the selected template and passes the resolved parameters to validators.

Benefits of dynamic parameterization. Dynamic numeric sampling provides several evaluation advantages over fixed test cases. First, it prevents trivial memorization: models cannot succeed by caching specific amount patterns from training data. Second, it tests numeric reasoning robustness: models must correctly parse arbitrary values from natural language (e.g., "transfer 0.37 tokens" vs. "transfer 100 tokens") and propagate them through unit conversions and calldata encoding. Third, it exposes edge-case failures: boundary values and unusual magnitudes reveal precision errors, overflow handling issues, and decimal truncation bugs that fixed test cases might miss. Fourth, it enables statistical significance: multiple runs with different parameter samples yield confidence intervals rather than single-point estimates. This setup ensures that models must generalize across different numeric inputs rather than pattern-match against known examples. Correctness depends on consistent use of task-provided context, including the contract address map, token decimals, and protocol constraints.

3.6 Validators and Post Execution Constraints

EVM-QuestBench uses validator-based scoring rather than reference code matching. This design enables evaluation of functional correctness under dynamic parameterization: validators receive the same randomly sampled parameters that were injected into the natural language instruction, and verify that the model's output produces the expected on-chain effects for those specific values.

Validator architecture. Each atomic task type has a dedicated validator class (e.g., `ERC20TransferValidator`, `SwapExactETHForTokensValidator`). At instantiation, validators receive the dynamically sampled parameters from the runner, including target addresses, token amounts, and decimal configurations. Composite tasks use a `CompositeValidator` that loads and delegates to the appropriate atomic validator based on the key operation in the workflow.

Validation against dynamic parameters. The validator converts human-readable parameters to chain-native units (e.g., 0.1 tokens $\rightarrow 0.1 \times 10^{18}$ wei for an 18-decimal token) and compares against actual on-chain state changes. For a transfer task, the validator checks: (i) the sender's balance decreased by the expected amount, and (ii) the receiver's balance increased by the same amount. This approach ensures that correctness is measured against the specific numeric values presented in the instruction, not against a fixed reference output.

Weighted scoring components. Validators implement protocol and task-specific checks over receipts and post-state. A typical atomic validator (e.g., ERC20 transfer) computes a weighted score from four check categories: transaction success (30 points), contract address correctness (20 points), function signature correctness (20 points), and state change verification (30 points). Each check is binary pass/fail with its assigned weight. The final score is the sum of passed check weights.

Tolerance handling. Many tasks allow small tolerances to account for AMM slippage or rounding effects. Transfer validators typically allow a 1 to 2% relative tolerance on amounts. Swap and liquidity operations use a slippage-style tolerance (commonly 5%). Approval tasks require an exact allowance match. These tolerances are applied during the state change verification step, ensuring that minor deviations from expected values (due to protocol mechanics) do not penalize otherwise correct solutions.

Composite validation. Composite validators compute a base success score from the final end-state condition, then apply step efficiency decay as described in Section 4. The validator loads the composite task definition, identifies the key operation, and reuses the corresponding atomic validator to score the final transaction. This modular design eliminates code duplication and ensures scoring consistency between atomic and composite evaluations of the same operation type.

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, we use a pass threshold of 60 points per task.

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:

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

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 $\rightarrow$ 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 uses the latest block at fork time. Within an evaluation run, we enforce comparability across tasks via snapshot isolation (Section 4.3).

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.

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 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

#	Model	Atom	Comp	Total	Avg
1	Gemini-3-Pro	4460	3757	8217	76.8
2	Kimi-K2	4400	3505	7905	73.9
3	Claude-Sonnet-4.5	4088	3781	7869	73.5
4	GPT-5.1	4145	3497	7642	71.4
5	DeepSeek-V3.2	2915	4088	7003	65.5
6	GPT-OSS-120B	2531	3682	6213	58.1
7	Grok-4-Fast	2853	3358	6211	58.0
8	Claude-Haiku-4.5	3816	2215	6031	56.4
9	Gemini-2.5-Flash	2265	3716	5981	55.9
10	GPT-5	3516	2443	5959	55.7
11	Qwen3-Max	2775	2698	5473	51.2
12	GPT-5.1-Codex-Mini	1573	3286	5286	49.4
13	Qwen3-235B	3580	1691	5271	49.3
14	Mimo-V2-Flash	2740	2209	4949	46.3
15	Qwen3-30B	2845	1887	4732	44.2
16	GLM-4.6	2771	1145	3916	36.6
17	Devstral-2512	2905	0	2905	27.2
18	Qwen3-Coder-30B	3015	0	3015	28.2
19	Qwen3-Coder	2825	0	2825	26.4
20	Qwen3-Coder-Flash	290	38	328	3.1

Table 1: Leaderboard. Atom=Atomic (62 tasks), Comp=Composite (45 tasks), Total=107 tasks. Avg=Total/107.

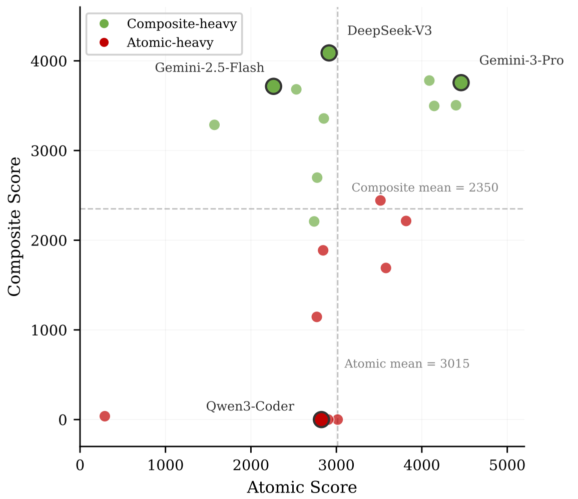

Figure 3: Atomic score versus Composite score. Each point is a model.

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. Table 1 reports Atomic, Composite, and Total scores, where Atomic sums 62 tasks, Composite sums 45 tasks, and Total sums 107 tasks (max 10,700). Avg is computed as `Total/107`.

The top three models exceed 7,800 total points. DeepSeek V3.2 achieves the strongest Composite score (4,088.3) while ranking mid range on Atomic (2,915), suggesting partial separability between workflow completion and single step exactness.

5.2 Atomic versus Composite Capability Separation

Figure 3 visualizes Atomic versus Composite scores. Models spread widely across the plane, 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) achieve high Composite despite weaker Atomic, suggesting stronger sequencing and end state targeting. Precision oriented models (Claude Haiku) achieve high Atomic but lag on Composite, consistent with weaknesses in multi step dependency tracking. Full quadrant assignments are provided in Appendix F.

5.3 Composite Execution Quality

Table 2 reports fine grained composite metrics for 13 models with complete composite evaluation logs. We analyze three dimensions: code quality (1st%), step efficiency (Eff%), and environment understanding (ethers%).

First try pass rate separates top models. GPT-5 achieves 86.7% first-try pass rate with only 5 total retries, indicating high code generation quality. Gemini-3-Pro follows at 71.1%. In contrast, models with 0% first-try pass rate (Claude Haiku, GLM) require retries on every task they pass.

Step efficiency reflects planning quality. Gemini-3-Pro leads with 97.3% efficiency despite having the same mean K_{act} as GPT-5 (2.2), indicating more consistent per-task optimization.

Model	Score	Pass	K	Eff	1st	eth
DeepSeek-V3.2	4088	42	3.0	90.6	60.0	26.7
GPT-5	3962	40	2.2	94.9	86.7	0.0
Gemini-2.5-Flash	3876	39	3.3	85.6	57.8	0.0
Gemini-3-Pro	3703	39	2.2	97.3	71.1	4.4
GPT-OSS-120B	3682	39	3.8	85.9	51.1	4.4
Kimi-K2	3605	37	3.5	87.0	46.7	42.2
Grok-4-Fast	3358	35	4.5	76.9	24.4	60.0
GPT-5.1-Mini	3183	30	4.7	75.9	26.7	51.1
Qwen3-Max	2668	24	5.2	72.2	17.8	73.3
Claude-Haiku	2238	15	6.1	64.8	0.0	100
GLM-4.6	1145	4	6.4	62.0	0.0	100
Qwen3-Coder-Fl	38	0	6.3	56.2	0.0	86.7
Qwen3-Coder-30B	0	0	6.6	58.4	0.0	100

Table 2: Composite execution quality (45 tasks). K=K_{act}, Eff=Eff%, 1st=1st%, eth=ethers%.

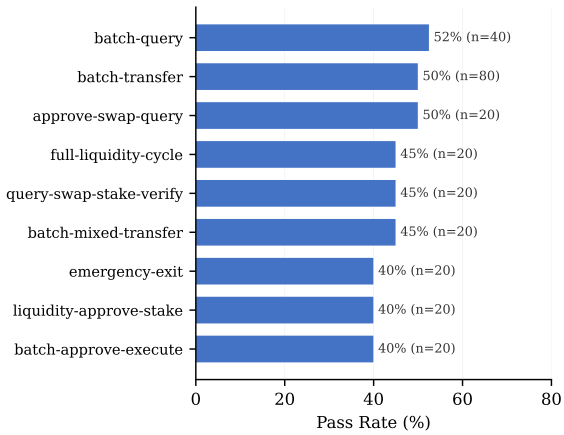

Figure 4: Composite workflow difficulty by pattern. Pass is defined as score ≥ 60.

Lower tier models average $K_{\text{act}} > 6$, which compounds both failure risk and score decay.

ethers% reveals environment misunderstanding. The "ethers is not defined" failure mode indicates models that do not use the pre-provisioned `ethers` library correctly. Models with 0% ethers errors (GPT-5, Gemini-2.5-Flash) consistently align with top tier composite performance, while models with near-ubiquitous ethers failures (Claude Haiku, GLM) show a fundamental mismatch to the execution interface.

Several code-specialized models (e.g., qwen3-coder, devstral) report Composite scores of 0 due to early termination triggered by repeated schema/interface failures. We treat these cases as coverage artifacts and exclude them from workflow failure analysis.

5.4 Where Workflows Fail

Figure 4 reports pass rates by 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, (ii) cross step parameter consistency, and (iii) execution robustness under multiple transactions. Additional analysis is in Appendix E.

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. Evaluation executes model generated TypeScript scripts in a forked environment with snapshot isolation. Validators score outcomes after execution. Composite tasks apply a step efficiency factor.

We evaluated 20 models under a unified runner. Performance varies substantially across models. Split scores expose a persistent gap between single transaction precision and workflow completion. Several models achieve strong Composite scores with weaker Atomic scores. Several models fail on Composite tasks despite non zero Atomic performance.

EVM-QuestBench offers a standardized protocol for studying execution grounded behavior in on-chain automation. Future work can expand task coverage, extend the benchmark to additional EVM chains, and strengthen analysis using per task traces. In fact, we have already leveraged the same architecture to develop and establish a benchmark on Solana, demonstrating the portability of our atomic/composite paradigm across heterogeneous blockchain ecosystems. We also plan to incorporate richer security checks for transaction intent and side effects. Additionally, we intend to evaluate LLMs' success rates in generating atomic and composite task definitions, which would further assess their understanding of on-chain transactions and smart contract semantics.

7 Limitations

Execution stability. Because EVM-QuestBench is execution grounded, scores inherit instability from RPC connectivity, fork performance, and provider availability. Snapshot isolation and composite retries reduce, but do not eliminate, this effect.

Reproducibility across forks. We do not pin a fixed fork block height. Snapshot restore ensures comparability within a run, but cross run results can drift with mainnet state, especially for price sensitive swaps and liquidity operations. Pinning the fork height and recording the fork block number in metadata would improve reproducibility.

Scoring and interface effects. Composite scoring applies a step efficiency decay factor $\min(1, K_{\text{opt}}/K_{\text{act}})$, where K_{act} counts execution rounds that may include retries; this can conflate planning inefficiency with execution instability. Validators also use tolerances for some tasks, which may admit occasional false positives or over penalize benign deviations. Finally, providers differ in default output budgets and formatting behavior, which can affect schema validity; prior work has noted that automatic metrics and LLM as a judge can introduce additional biases (Zheng et al., 2023; Liu et al., 2023b).

Single-run evaluation. Due to time and computational cost constraints, we evaluated each model only once rather than conducting multiple runs. This single-run protocol limits our ability to report confidence intervals or assess variance in model performance. Stochastic factors such as temperature-induced generation variability and non-deterministic execution timing may affect individual scores. Future work should incorporate multiple runs per model to provide more statistically robust comparisons.

References

Rohan Anil and 1 others. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*.

Jacob Austin, Augustus Odena, Maxwell Nye, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Mark Chen, Jerry Tworek, Heewoo Jun, and 1 others. 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, and 1 others. 2023a. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*.

Yang Liu and 1 others. 2023b. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of EMNLP*.

Shuai Lu, Daya Guo, Shuo Ren, and 1 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, and 1 others. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Shuo Ren and 1 others. 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 and 1 others. 2024. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. In *Proceedings of ACL*.

Lianmin Zheng and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*.

A 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.

A.1 Environment and Dependencies

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

- Python 3.10 or newer
- Node.js 18 or newer (Bun 1.0 or newer is also supported)
- Foundry with Anvil available in PATH

Item	Value
Chain	EVM-compatible mainnet fork
Runtime	TypeScript runner using ethers.js v6
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 3: Reproducibility relevant settings.

A.2 Execution Commands

The following commands run the benchmark for a given model identifier.

```
# Run all 107 tasks
python run_quest_bench.py --model <MODEL_NAME>

# Split runs
python run_quest_bench.py --model <MODEL_NAME>
    --type atomic
python run_quest_bench.py --model <MODEL_NAME>
    --type composite

# Naive mode (adds hints in task context)
python run_quest_bench.py --model <MODEL_NAME>
    --naive-mode
```

A.3 Key Experimental Settings

Table 3 lists the settings that most often explain score variance across reruns.

B 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.

B.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;
}
```

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 4: Atomic task schema summary.

B.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.
```

B.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

C Task Definition Schema

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

C.1 Atomic Task Fields

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

C.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 5: Composite task schema additions.

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 7: Step efficiency decay examples.

Family	Typical tolerance	Examples of checks
Native and ERC20 transfer	small relative tolerance	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 6: Validator families summary.

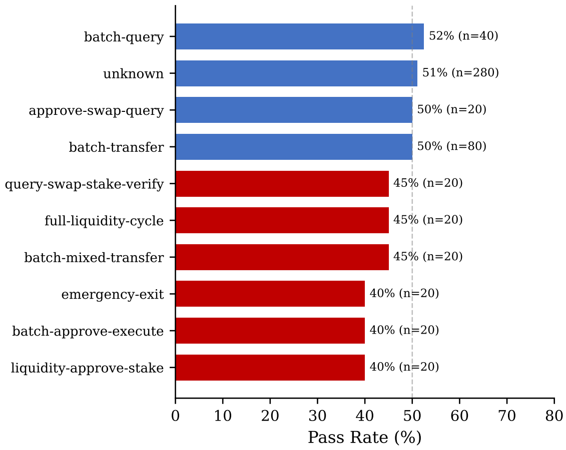

Figure 5: Pass rate by composite workflow pattern for the ten most frequent patterns.

D 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.

D.1 Validator Families

Table 6 summarizes common validator families and the constraints they evaluate.

D.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.

E Additional Analysis Figures

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

E.1 Composite Pattern Difficulty

Figure 5 reports pass rates for frequent composite workflow patterns. Lower pass rates indicate higher coordination burden across steps, including prerequisite approvals and parameter propagation.

E.2 Step Overhead Distribution

Figure 6 shows the distribution of step overhead $\Delta = K_{\text{act}} - K_{\text{opt}}$ aggregated across composite runs. Positive overhead indicates extra rounds consumed by redundant queries, retries, or incorrect intermediate actions.

E.3 Atomic Subcategory Difficulty

Figure 7 summarizes atomic 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.

F Full Evaluation Tables

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

F.1 Quadrant Summary

Table 8 summarizes the four capability profiles based on AtomicAvg and CompositeAvg medians (46.5 and 73.0 respectively).

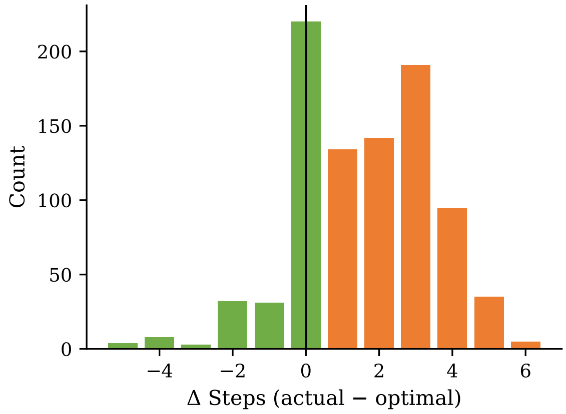

Figure 6: Distribution of step overhead $\Delta = K_{\text{act}} - K_{\text{opt}}$.

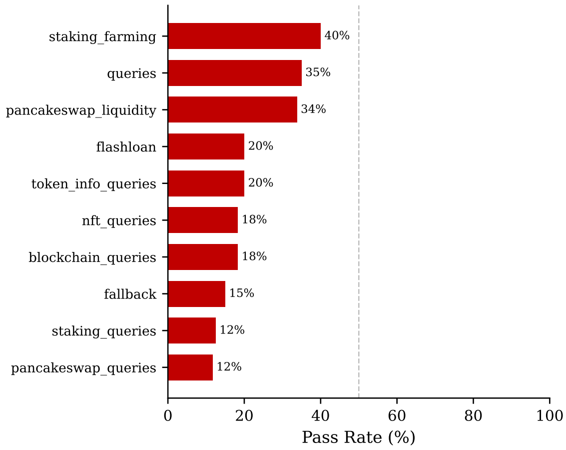

Figure 7: Pass rate by atomic subcategory.

F.2 Model Coverage and Quadrant Classification

Table 10 reports coverage, normalized averages, total score, and the quadrant label based on AtomicAvg and CompositeAvg medians.

F.3 Step Overhead and Decay Impact Summary

Table 9 reports overhead statistics for selected models and the composite score under decay. The overhead statistic is $\Delta = K_{\text{act}} - K_{\text{opt}}$.

G Case Studies

This section provides representative composite workflows that stress end to end execution. The goal is to highlight failure points such as parameter propagation, prerequisite handling, and protocol sequencing.

Profile	**Count**	**Interpretation**
High High	5	strong precision and workflows
High Low	3	strong precision, weaker workflows
Low High	4	weaker precision, stronger workflows
Low Low	5	weaker on both splits
Atomic only	3	composite split not executed

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

Model	Mean Δ	Over-opt (%)	P90 Δ	Comp. score
gemini-2.5-flash	-0.40	24	1	3715.7
gemini-3-pro-preview	-0.69	33	1	3756.8
gpt-5	-0.98	22	1	2443.3
deepseek-v3.2	+0.56	42	1	4088.3
gpt-oss-120b	+0.44	38	1	3682.3
kimi-k2-thinking	+0.71	49	2	3504.9
grok-4-fast	+1.00	53	3	3357.5
claude-haiku-4.5	+2.49	89	4	2214.7
glm-4.6	+2.91	100	5	1145.1

Table 9: Step overhead and decay impact summary.

G.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:** 45%.

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

G.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:** 40%.

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.

Model	Atomic	Composite	AtomicAvg	CompAvg	Total	Quadrant
gemini-3-pro-preview	62/62	45/45	71.9	83.5	8216.8	High-High
claude-sonnet-4.5	62/62	45/45	65.9	84.0	7868.5	High-High
kimi-k2-thinking	62/62	45/45	71.0	77.9	7904.9	High-High
gpt-5.1	62/62	45/45	66.9	77.7	7642.3	High-High
deepseek-v3.2	62/62	45/45	47.0	90.9	7003.3	High-High
gemini-2.5-flash	62/62	45/45	36.5	82.6	5980.7	Low-High
gpt-oss-120b	62/62	45/45	40.8	81.8	6213.3	Low-High
grok-4-fast	62/62	45/45	46.0	74.6	6210.5	Low-High
gpt-5.1-codex-mini	62/62	45/45	25.4	73.0	5286.3	Low-High
gpt-5	62/62	45/45	56.7	54.3	5959.3	High-Low
claude-haiku-4.5	62/62	45/45	61.5	49.2	6030.7	High-Low
qwen3-235b-thinking	62/62	45/45	57.7	37.6	5270.6	High-Low
qwen3-max	62/62	45/45	44.8	60.0	5472.9	Low-Low
glm-4.6	62/62	45/45	44.7	25.4	3916.1	Low-Low
qwen3-30b-thinking	62/62	45/45	45.9	41.9	4732.0	Low-Low
mimo-v2-flash	62/62	45/45	44.2	49.1	4949.4	Low-Low
qwen3-coder-flash	62/62	45/45	4.7	0.8	327.5	Low-Low
qwen3-coder	62/62	0/45	45.6	0.0	2825.0	Atomic-Only
qwen3-coder-30b	62/62	0/45	48.6	0.0	3015.0	Atomic-Only
devstral-2512	62/62	0/45	46.9	0.0	2905.0	Atomic-Only

Table 10: Model coverage and quadrant classification. Averages are normalized to the 0–100 scale.