BEDROCKBENCH: CAN LLMS SPEAK MACHINE CODE?

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models achieve strong results in high-level code generation, but this success does not establish whether they can directly construct executable machine-code bytes. We introduce BedrockBench, the first benchmark for direct machine-code function synthesis from natural-language specifications, with 100 tasks targeting x86-64. Models submit hexadecimal function bytes without external tools or execution feedback. A shared evaluator executes these bytes unchanged and checks functional behavior, memory accesses, return state, and instruction and resource constraints across legal execution environments. Diagnostics distinguish submission failures, execution violations, and wrong answers. Across seven model configurations, pass@1 ranges from 13% to 100%, with GPT-6-Astra passing all 100 tasks. Among 240 format-valid unsuccessful candidates, unauthorized memory accesses are observed in 160 (66.7%), making memory-contract violations a prominent observed failure mode. BedrockBench provides an explicit and auditable basis for comparing models and studying how to improve the reliability of direct machine-code synthesis. The benchmark data and code are publicly available.[1]

1 INTRODUCTION

Large language models (LLMs) have shown strong performance in generating programs in high-level languages such as Python and C, as measured by benchmarks including HumanEval, MBPP, and LiveCodeBench (Chen et al., 2021; Austin et al., 2021; Jain et al., 2025). However, success at source-level generation does not establish whether these models can directly construct executable machine-code bytes. In conventional compiled-code pipelines, toolchains perform instruction selection, register allocation, and binary encoding. Direct machine-code generation places these responsibilities on the LLM, requiring it to translate a functional specification into instructions whose behavior and encoding satisfy the target machine interface. This raises a basic question: how reliably can LLMs generate working machine-code functions without a compiler or assembler?

Research on low-level program synthesis provides motivation for studying this question. Stochastic superoptimization demonstrates that searching over machine-level instruction sequences can outperform optimizing compilers on selected tasks (Schkufza et al., 2013), while SuperCoder shows that LLMs can improve compiler-generated assembly, with further gains from reinforcement learning using correctness and performance rewards (Wei et al., 2025). Other work explores compiler representations and reasoning over compiled bytes (Cummins et al., 2025; Störtz et al., 2026). These studies establish the relevance of learned reasoning about low-level programs, but their evaluations do not directly measure whether LLMs can synthesize executable machine-code bytes from natural-language functional specifications. A systematic evaluation of this capability must check both the intended computation and the machine-level requirements that make the generated function executable.

We introduce BedrockBench, the first benchmark for evaluating the ability of LLMs to generate machine code directly from natural-language specifications. It contains 100 programming tasks adapted from LiveCodeBench, each providing a natural-language specification, public examples, a function interface, and an explicit execution contract. An LLM submits a hexadecimal representation of the

[1] https://anonymous.4open.science/r/bedrockbench-A03E

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

function's x86-64 machine-code bytes in a single response, without external tools or execution feedback. A shared evaluator executes the submitted bytes unchanged and checks functional outputs, memory accesses, return state, and instruction and resource constraints. Diagnostics distinguish observed failures and examine whether candidate behavior changes across legal execution environments. Figure 1 summarizes this workflow.

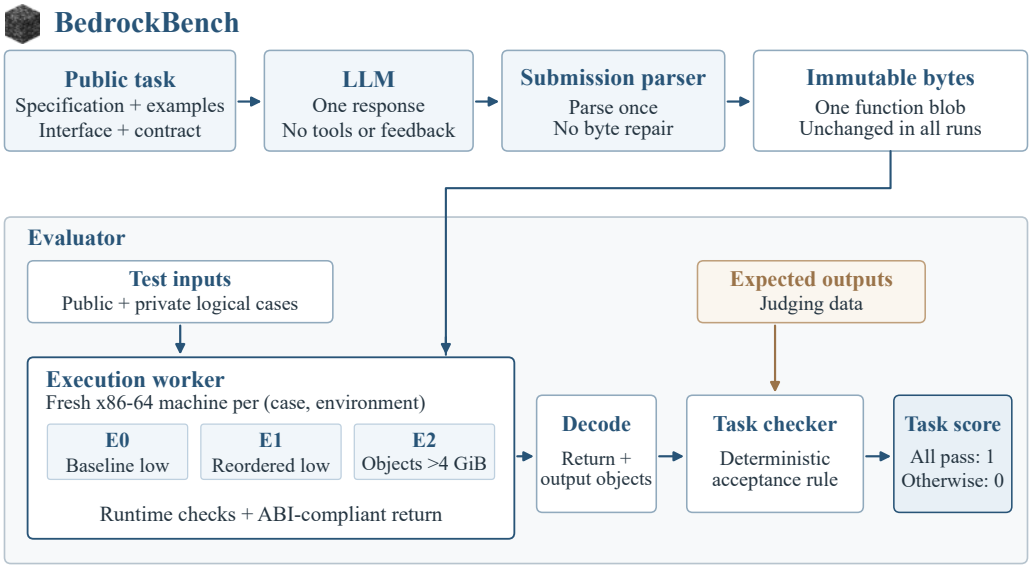

Figure 1: Overview of BedrockBench. The model receives the public task and submits a single response. Parsing produces a fixed byte blob without repair. Each public or private logical input is executed in three legal environments with fresh machine state and a reset instruction budget. The execution worker receives inputs but no expected outputs or judging data. Only a valid return permits output decoding and deterministic functional checking. A task succeeds only if every case–environment execution passes. Format-rejected responses score zero without execution. Any observed violation also yields zero.

Our contributions comprise a benchmark for specification-to-byte function synthesis, an execution and diagnostic protocol that jointly checks functional behavior and machine-level obligations, and an empirical analysis of seven model configurations on the same 100 tasks. Pass@1 ranges from 13% to 100%, with GPT-6-Astra solving all 100 tasks (Section 4). These results show that direct byte generation can satisfy the full observed contract, while performance varies substantially across configurations. The sources of non-success also differ: Claude passes all 93 tasks for which it provides a format-valid submission, while its remaining seven tasks receive refusals. Among the 240 format-valid unsuccessful candidates from the other configurations, unauthorized memory accesses are observed in 160 (66.7%). BedrockBench provides a controlled setting for studying how reliably LLMs can turn functional specifications into executable machine code under explicit machine-level constraints.

2 Benchmark Construction

2.1 Data Source

We constructed BedrockBench from LiveCodeBench (Jain et al., 2025). LiveCodeBench curates programming tasks from LeetCode, AtCoder, and Codeforces. Each record contains a natural-language problem statement, difficulty metadata, public and private input–output cases, and starter code when applicable. Tasks use either function-call interfaces or standard-input/standard-output programs.

We used `release_v6`, which contains 1,055 programming tasks.[2] We selected 100 tasks from this release and adapted them into machine-code function tasks. Selection balanced eight primary algorithm classes and the source difficulty labels, with 30 easy, 50 medium, and 20 hard tasks. Table 1 reports the final distribution.

2.2 Task Adaptation

We used GPT-6-Astra for task adaptation. For each task, the constructing agent read the source problem and tests, specified the function interface, deterministic output checker, and resource limits, and implemented a deterministic converter. The converter produced the task specification, public and private cases, and source-to-case mappings. After mechanical replay and validation, the agent inspected the complete public prompt and recorded its semantic and resource assessment.

For function-style sources, adaptation retained parameters, return values, and observable mutation. For standard-input tasks, it replaced input/output transport with a function interface while retaining the computational problem and legal input domain. The evaluator handles input encoding and argument passing, while parsing or formatting intrinsic to the problem remains the candidate's responsibility. Integer widths, string encodings, permissions, aliasing, and initialization guarantees are explicit. Arrays and strings expose their effective lengths through parameters, public length declarations, or self-describing representations. An allocation capacity alone does not establish the input length. Output and workspace capacities depend on public inputs and source constraints, not expected answers. Additional workspace has a public pointer and capacity, and the allocated stack is also available.

Resource assessment combines analysis of a reasonable algorithm over the original input domain with measured execution throughput. It accounts for input and output storage, temporary memory, stack use, and the cost of helpers implemented without an external runtime. Conservative throughput estimates and allowances for startup and data transfer inform the instruction and wall-clock budgets.

The final interfaces include 77 tasks with scalar-return results, 24 with buffer outputs, 29 with explicit workspace, and 4 with stack-passed arguments. Appendix A illustrates both a scalar interface and a buffered graph task.

2.3 Test Construction and Judging

For each task, we built the test suite from upstream logical inputs and expected results, without adding new random, boundary, exhaustive, or metamorphic cases. Of 2,548 upstream test records, 2,372 were retained directly, 81 were split, 62 were deduplicated, and 33 were excluded because their inputs violated the original problem constraints. Each source record retains its identity and disposition. Retained, split, or deduplicated records link to the resulting cases. Splitting converts independent batched instances into function calls while preserving their parent mappings.

After conversion, the final benchmark contains 37,584 logical cases: 295 public examples and 37,289 private tests. Source records, logical calls, and machine executions are different units: running the same call under several legal environments does not create a new logical input. Tasks, rather than individual cases, receive equal weight in the benchmark score.

Output correctness is assessed using exact matching for 95 tasks, floating-point tolerance checks for 2, multiset comparison for 2, and field-wise checking for 1. Multiset comparison accepts different output orders while checking element multiplicities. These tasks require a fixed collection of answers rather than an arbitrary valid witness. Deduplication checks agreement under the declared comparison relation instead of requiring identical serialized output order.

2.4 Validation

Validation separates task-level review, mechanical consistency checks, and shared evaluator tests. The constructing agent checks the original problem semantics, input information, output comparison, test adjustments, resource feasibility, and the final public prompt. This is self-review by the constructing agent rather than an independent semantic review. Mechanical validation replays each

[2] `https://github.com/LiveCodeBench/LiveCodeBench#dataset-versions`

Table 1: Joint distribution of the 100 tasks in BedrockBench by primary algorithm class and source difficulty. Each task contributes to one cell.

Primary algorithm class	Easy	Medium	Hard	Total
Counting	3	10	2	15
Dynamic programming	0	9	6	15
Graphs	0	8	7	15
Greedy algorithms	5	4	1	10
Mathematics	3	4	3	10
Ordering	2	8	0	10
Search	3	7	0	10
Simulation	14	0	1	15
Total	30	50	20	100

converter against its source and checks agreement with the submitted task and mappings. It verifies complete source-test accounting, schema consistency, input encoding round trips, accessible input lengths, output capacities and return widths, application binary interface (ABI) argument placement, checker compatibility, and consistency between the public prompt and structured task fields. Shared positive and negative machine fixtures separately exercise instruction behavior, ABI obligations, memory boundaries, and execution-state isolation.

3 Execution and Evaluation Protocol

To assess both functional correctness and compliance with the execution contract, we design an evaluator that parses submissions, coordinates execution across test cases and environments, and aggregates outcomes. Its executor runs candidate machine code and checks the machine-level contract, while the task-specific checker judges the decoded output.

3.1 Submission Format and Parsing

The submission parser reads final answer content, excluding separately returned reasoning, and accepts a nonempty blob as raw hexadecimal text or inside a unique complete `hex` block. Response-body and decoded-byte limits both apply. Extraction removes only the wrapper and byte-boundary whitespace, without repairing instructions or choosing candidates. Parsing occurs once, and every execution reuses the resulting bytes. Appendix B.1 specifies the byte grammar and packaging diagnostics.

3.2 Machine Contract

The executor uses the Unicorn emulation engine (Nguyen & Dang, 2015) with fixed floating-point behavior and support for the admitted instruction encodings. The evaluator invokes the candidate according to the System V AMD64 (x86-64) ABI.[3] Scalar integers and pointers use integer argument registers and overflow stack slots, whereas scalar floating-point arguments use their own register class. Returns are interpreted at their declared widths. The current interface supports scalar signatures, while general aggregates, vector-valued signatures, and `long double` are outside its scope. Internal single-instruction, multiple-data (SIMD) operations remain available within the target instruction profile.

Execution begins at the public code base with an aligned stack, red zone, typed objects, and a return sentinel. A successful return reaches the sentinel with the prescribed stack pointer, preserves callee-saved registers and floating-point control state, clears the direction flag, and leaves MMX mode if it was entered. An equivalent return sequence is accepted without requiring a particular final instruction. These obligations apply to the outer function, leaving internal helper conventions to the candidate.

[3] `https://gitlab.com/x86-psABIs/x86-64-ABI`

The executor emulates an x86-64 processor with x86-64-v2 instruction-set support, subject to the declared restrictions on system services and instruction extensions. The instruction policy checks encodings as execution reaches them, so embedded constants need not disassemble. Code is readable and executable but not writable, while data and stack are non-executable. Every actual memory access must fit a permitted logical region in its entirety, even when an overrun would remain within a mapped page. For example, writing to read-only input fails even if the program would later restore its original contents. Writable storage is confined to the allocated stack and objects with write permission. Gaps between objects are not workspace.

Public task limits bound response size, code size, stack space, and instruction units. The instruction budget resets for each case–environment invocation. Ordinary and SIMD instructions count once, while repeated string instructions count per iteration. Appendix B specifies zero-count and return-boundary rules.

3.3 Execution and Functional Checking

For each logical input, the evaluator constructs the initial machine state from the declared arguments, objects, capacities, and permissions using the interface from Section 2.2. It supplies no answer-relevant precomputed features. The executor runs in an execution worker process that receives the immutable blob and the current input's machine state. Expected outputs and other judging data remain with the evaluator's checker and are not passed to the worker. Each invocation creates a fresh emulated machine, even when the worker process is reused.

Checks follow the execution order in Table 2. The first dynamic violation stops that invocation. Only normal return with a valid ABI state permits result extraction and functional checking. The evaluator decodes the declared return value and output objects, rejecting lengths outside declared capacities and invalid encodings. The task's deterministic checker then applies its public acceptance relation to the logical result. It neither compares against reference machine code nor invokes an LLM to grade the answer.

Table 2: Ordered checks from submission through functional judgment. A candidate violation leaves later stages unobserved, so a correct return value cannot compensate for an earlier contract violation.

Stage	Acceptance condition
Submission	One valid blob within the response and code limits.
Dynamic execution	Reached instructions, control flow, memory accesses, and resource use obey the contract.
Return state	Sentinel reached with the required stack, register, and numerical state.
Output decoding	Returned values and output objects satisfy declared lengths, capacities, and encodings.
Logical checking	Decoded result satisfies the task's acceptance relation.

3.4 Environment Variation

Let C_t contain all retained public examples and private tests for task t, and let $L = \{0, 1, 2\}$. We execute each case under all three legal environments to test dependence on addresses or initial state that the public contract does not guarantee. Environment 0 uses a baseline low-address object layout. Environment 1 changes object order and offsets within the low-address pool. Environment 2 places objects above 4 GiB. The variants also change unspecified register values, arithmetic flags, and memory contents. Logical inputs, expected results, capacities, permissions, and all public guarantees remain fixed.

These are deterministic variations of the machine environment, with no new logical inputs or model requests. A format-valid candidate is evaluated in $3|C_t|$ executions over $|C_t|$ logical cases. Candidate failures do not stop the remaining cases or environments, allowing diagnostics to capture additional observed violations.

3.5 Scoring and Diagnostics

Let r denote the model's generated response for task t. Define $F_t(r) \in \{0, 1\}$ as the format-validity indicator under the submission grammar and size limits. When $F_t(r) = 1$, $P_t(r)$ denotes the extracted machine-code blob. For each case c and environment ℓ, let $A_t(b, c, \ell) \in \{0, 1\}$ indicate whether blob b passes all checks in Table 2. The task outcome is

$$Y_t(r) = \begin{cases} 0, & F_t(r) = 0, \\ \displaystyle\prod_{c \in C_t} \prod_{\ell \in L} A_t(P_t(r), c, \ell), & F_t(r) = 1. \end{cases} \tag{1}$$

Because each factor is binary, the product equals one only when every case–environment execution passes. A single failure makes it zero. Format-rejected responses score zero without execution.

Each task has equal weight, regardless of its number of cases. Averaging Y_t over one generated response per task gives pass@1 (Chen et al., 2021), as reported in Section 4. Reporting format validity separately from success conditional on valid format distinguishes format-rejected responses from format-valid unsuccessful candidates. The format-valid subsets can differ across models.

Individual checks record pass, fail, or not observed. For example, an out-of-bounds read fails the task but stops that execution before functional checking, leaving its answer correctness unobserved. Each candidate is counted once per observed failure category, even if the failure occurs in many executions. A candidate can exhibit several categories across inputs or environments, so category counts overlap and cannot be added to obtain the number of unsuccessful candidates.

Environment sensitivity is observed when the same logical case passes in some environments and fails in others. Three passes show no observed sensitivity. Three failures do not establish whether the candidate is sensitive. Different outputs accepted by the checker do not count as sensitivity. Because several state components vary together, identifying a specific cause requires instruction-level evidence or controlled replay.

4 Experiments and Analysis

4.1 Experimental Setup and Scoring

The study evaluates flagship models from seven developers at the time of evaluation: GPT-6-Astra, Claude Opus 5.5, Grok 4.7, DeepSeek V4 Pro 0813, Qwen 3.8 Max, Kimi K3, and GLM-5.3. All requests use the same public task prompts and request reasoning effort `high` and a 128,000-token completion cap. All other generation settings use provider defaults. Each request starts a fresh conversation without external tools, and submitted bytes are evaluated unchanged under Section 3. Exact identifiers and routes appear in Appendix C.

For model m, we report pass@1 as

$$\widehat{S}_m = \frac{1}{100} \sum_{t=1}^{100} Y_t(r_{mt}), \tag{2}$$

where r_{mt} is the response for task t and Y_t is the full-contract outcome defined in Equation 1. Responses that provide no valid submission score zero. Every task retains equal weight, regardless of its number of test cases.

4.2 Overall Performance and Task Difficulty

Table 3 and Figure 2(a) show substantial variation across configurations. GPT passes all 100 tasks, establishing that direct byte generation can satisfy the complete observed contract on this suite. Claude achieves 93%[4] pass@1, followed by Grok (72%), DeepSeek (62%), Qwen (58%), Kimi (45%), and GLM (13%)[5].

[4]Claude refused seven tasks, which score zero and remain in the 100-task denominator. We made five attempts in total for each of these tasks to confirm refusal persistence. All responses were refusals: "This request triggered restrictions on violative cyber content and was blocked under Anthropic's Usage Policy."

[5]GLM's four responses that exhaust the generation-token budget score zero. All terminate with `length` at the reported 128,000-token completion cap and provide no valid blob.

Table 3: Results on 100 tasks per model. Format validity follows the submission grammar and task limits. Solving a task requires passing every retained test case in all three environments. Pass@1 uses all 100 tasks.

Model	Format valid	Solved	Pass@1
GPT-6-Astra	100	100	100%
Claude Opus 5.5	93	93	93%
Grok 4.7	99	72	72%
DeepSeek V4 Pro 0813	98	62	62%
Qwen 3.8 Max	100	58	58%
Kimi K3	98	45	45%
GLM-5.3	95	13	13%

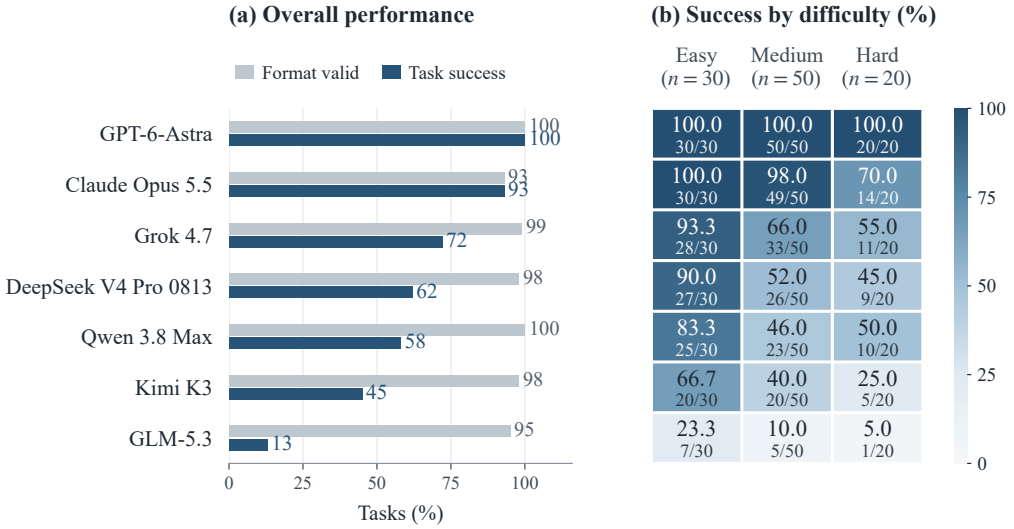

Figure 2: Overall performance and source difficulty on the 100-task suite. (a) Format validity and pass@1, using all 100 tasks per model. (b) Pass@1 by source difficulty. Each cell reports a percentage and the solved/total count. Difficulty labels are inherited from the source tasks.

Figure 2(b) shows lower success on hard than easy tasks for every model except GPT. Grok decreases from 93.3% to 55.0%, DeepSeek from 90.0% to 45.0%, and Kimi from 66.7% to 25.0%. The pattern is not uniformly monotonic: Qwen scores 46.0% on medium and 50.0% on hard tasks. Claude's six hard-task non-successes are refusals, so its difficulty gradient does not demonstrate declining execution correctness among valid submissions.

4.3 Failure Analysis

There are 240 format-valid unsuccessful candidates. Figure 3 summarizes their observed failures. Each candidate is counted once per observed status, even when many executions exhibit it. Counts can overlap across inputs and environments. They do not partition failures or identify independent root causes. All format-valid submissions from GPT and Claude succeed.

Unauthorized reads or writes are observed in 160 of these 240 candidates (66.7%), counting their union rather than adding the two rows. Memory-contract violations are therefore a prominent observed failure mode. Wrong answers are observed in 78 candidates after execution reaches functional checking, demonstrating a further gap beyond legal execution.

Response-stage failures require separate interpretation. Beyond the four generation-budget-exhausted responses and Claude's seven refusals, six responses fail parsing: one Grok, two DeepSeek, two Kimi, and one GLM. Together with the 240 format-valid unsuccessful candidates, these account for the 257 non-successes.

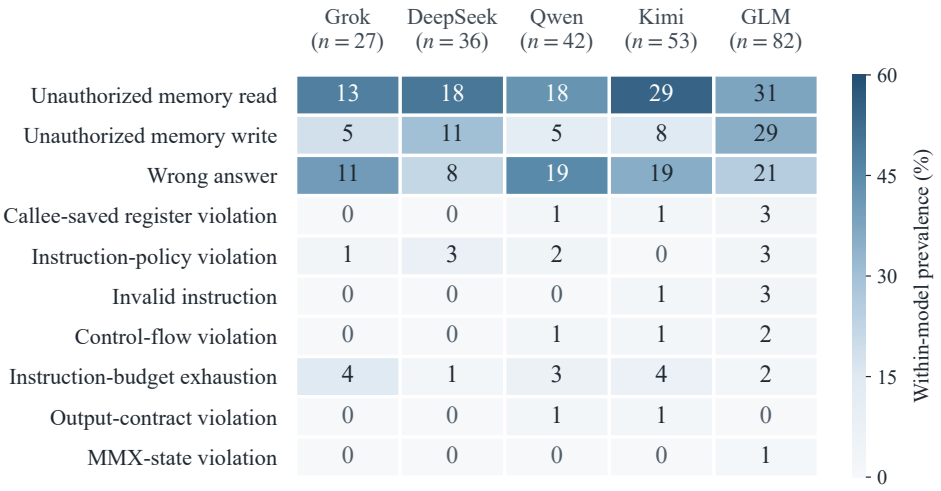

Figure 3: Observed failure categories among 240 format-valid unsuccessful candidates. Cell labels give candidate counts, and color gives the percentage of the n failed candidates in each column. Each candidate is counted once per observed status, but categories may overlap across inputs and environments. All format-valid submissions from GPT and Claude succeed.

4.4 Environment Sensitivity

In the available complete per-execution reports, requiring all three environments does not reduce task-level success relative to environment 0 alone. Environment variation nevertheless reveals mixed pass/fail behavior on identical logical inputs for 12 candidates: three Grok, two Qwen, three Kimi, and four GLM, covering 85 candidate–case pairs. All 12 already fail at least one environment-0 case. The extra environments therefore add diagnostic coverage without discovering additional unsuccessful tasks among these candidates.

5 Related Work

Evaluation of source-code generation. HumanEval and MBPP evaluate synthesis from natural-language specifications (Chen et al., 2021; Austin et al., 2021), while APPS and SWE-bench cover programming challenges and repository-level issue resolution (Hendrycks et al., 2021; Jimenez et al., 2024). LiveCodeBench collects timestamped competition problems and evaluates coding scenarios including generation and self-repair (Jain et al., 2025). EvalPlus shows that stronger tests can expose incorrect programs accepted by existing suites (Liu et al., 2023). BedrockBench evaluates adapted machine-code tasks, so its scores do not describe the original benchmark. Machine-state checks do not replace semantic test coverage, and adaptation does not establish absence of training contamination.

Low-level synthesis and optimization. BinMetric includes 100 natural-language-to-assembly tasks with compilation and execution checks (Shang et al., 2025). Its output is assembly text, whereas BedrockBench executes model-generated function bytes unchanged and checks functional behavior, ABI state, and memory accesses. Stochastic superoptimization and SuperCoder optimize existing low-level programs. SuperCoder uses source code, compiler-generated assembly, and correctness and performance feedback (Schkufza et al., 2013; Wei et al., 2025).

Neural compilation and binary understanding. Cummins et al. (2023) predict optimization options from LLVM assembly, while LLM Compiler studies optimization and translation from assembly to LLVM intermediate representation (IR) (Cummins et al., 2025). QiMeng-NeuComBack translates LLVM IR to assembly with execution-feedback self-debugging and optimization (Fang et al., 2025). These settings supply an existing program representation, whereas BedrockBench

requires deriving an implementation from a functional specification and encoding its instructions. LLM4Decompile reconstructs source from disassembled binaries, Nova studies assembly representations for decompilation and binary similarity, and Large Byte Model analyzes compiled bytes (Tan et al., 2024; Jiang et al., 2025; Störtz et al., 2026). These analysis tasks establish relevant low-level modeling capabilities but do not evaluate specification-to-byte function synthesis.

6 Discussion and Limitations

The results reveal substantial performance differences across models, with some already capable of directly synthesizing machine-code functions that satisfy the benchmark's functional and machine-level requirements.

BedrockBench measures function-level synthesis for one emulated target profile. Success on retained inputs does not prove correctness over a task's full domain, and three legal environments cannot establish independence from every unspecified initial state. The evaluation does not measure executable packaging, operating-system interaction, portability, native runtime performance, or general software development. The fixed emulator and shared validation fixtures also do not establish complete hardware equivalence. Tool-assisted generation, repeated sampling, and iterative repair require separate experiments. Scores remain conditional on the generation settings. Nominal token caps and reasoning-effort labels do not establish equivalent inference compute across models.

The targeted 100-task suite inherits potential upstream exposure and test weaknesses. The perfect observed GPT score therefore motivates a broader set of independent tasks and tests before drawing conclusions about general machine-code competence or improvements beyond this suite's ceiling.

7 Conclusion

We introduced BedrockBench, a benchmark of 100 programming tasks for direct machine-code function synthesis from natural-language specifications, together with an evaluator that checks functional correctness and compliance with memory, ABI, instruction, and resource requirements. The evaluated models achieve pass@1 ranging from 13% to 100%, with GPT-6-Astra satisfying the full observed contract on all 100 tasks. These results demonstrate that some models can already synthesize working machine-code functions without compiler or assembler assistance, while substantial performance differences remain across models. By requiring executable instructions as the output, BedrockBench brings algorithm implementation, instruction encoding, and machine-interface compliance into a common evaluation framework. It provides an explicit and auditable basis for comparing models and studying how to improve the reliability of direct machine-code synthesis.

AI use statement

We used generative AI tools to assist with benchmark dataset construction, literature retrieval and discovery, language polishing, figure generation, and the development of scripts for auditing and aggregating experimental records. The retrieved references and the audit and aggregation scripts were manually reviewed and verified. The authors take responsibility for the final content of this work, including its claims, references, benchmark artifacts, code, and reported results.

Ethics statement

BedrockBench evaluates machine-code generation using programming tasks adapted from LiveCodeBench and does not recruit human participants. The tasks concern general-purpose algorithmic functions and do not request malicious payloads. Nevertheless, advances in machine-code generation may also be used for harmful purposes, so model-generated bytes must be treated as untrusted code. Our evaluator enforces memory-access, instruction, and resource constraints within an emulated machine, but these checks do not establish host-system isolation or guarantee that generated code is suitable for deployment. Safely executing untrusted submissions requires an independently validated isolation mechanism. We use LiveCodeBench for academic evaluation and retain original

problem sources, attribution, and task provenance. Data use and distribution are subject to the applicable upstream licenses and terms of use. Experimental results apply only to the tested tasks and execution environments and should not be interpreted as guarantees of general code correctness or software safety.

REPRODUCIBILITY STATEMENT

The benchmark data and code are available at `https://anonymous.4open.science/r/bedrockbench-A03E`. Sections 2–4 specify conversion, execution, and the direct machine-code generation study. Appendices A–C provide worked interface examples, machine-contract details, and evaluated model information. The analysis code aggregates task outcomes, format validity, observed failure categories, and environment sensitivity. Model identifiers and provider tags are documented in Appendix C.

REFERENCES

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Rozière, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. LLM compiler: Foundation language models for compiler optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, pp. 141–153. Association for Computing Machinery, 2025.

Hainan Fang, Yuanbo Wen, Jun Bi, Yihan Wang, Tonghui He, Yanlin Tang, Di Huang, Jiaming Guo, Rui Zhang, Qi Guo, and Yunji Chen. QiMeng-NeuComBack: Self-evolving translation from IR to assembly code. In *Advances in Neural Information Processing Systems*, volume 38, pp. 169507–169530, 2025.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025.

Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. Nova: Generative language models for assembly code with hierarchical attention and contrastive learning. In *International Conference on Learning Representations*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pp. 21558–21572, 2023.

Anh Quynh Nguyen and Hoang Vu Dang. Unicorn: Next generation CPU emulator framework. Black Hat USA presentation, 2015. URL https://www.blackhat.com/docs/us-15/materials/us-15-Nguyen-Unicorn-Next-Generation-CPU-Emulator-Framework.pdf. 5 August 2015.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 305–316, 2013.

Xiuwei Shang, Guoqiang Chen, Shaoyin Cheng, Benlong Wu, Li Hu, Gangyang Li, Weiming Zhang, and Nenghai Yu. BinMetric: A comprehensive binary code analysis benchmark for large language models. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pp. 7715–7723, 2025.

Florian Störtz, Catalin-Andrei Stan, Alexandru Dinu, Sandra Servia-Rodríguez, Mihaela Gaman, Calin Miron, and Edward Raff. Large byte model: Teaching language models about compiled code. *arXiv preprint arXiv:2606.02834*, 2026.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. LLM4Decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3473–3487. Association for Computational Linguistics, 2024.

Anjiang Wei, Tarun Suresh, Huanmi Tan, Yinglun Xu, Gagandeep Singh, Ke Wang, and Alex Aiken. SuperCoder: Assembly program superoptimization with large language models. *arXiv preprint arXiv:2505.11480*, 2025. URL https://arxiv.org/abs/2505.11480v2. Version 2.

A WORKED TASK ADAPTATIONS

Scalar interface: Right Triangle. The supplied task `lcb_abc362_b`, *Right Triangle*, asks whether three non-collinear integer-coordinate points form a right triangle. The source domain constrains each coordinate to $[-1000, 1000]$. The adaptation keeps those coordinates and the non-collinearity condition, removes only standard-input transport, and maps the original yes/no output to a Boolean return. It does not provide distances, angle information, or a precomputed classification to the candidate. Table 4 summarizes the source-to-machine mapping.

```
_Bool solve(int64_t x_A, int64_t y_A,
            int64_t x_B, int64_t y_B,
            int64_t x_C, int64_t y_C);
```

Table 4: Concrete source-to-machine mapping for the example. These values are read from the supplied task, not inferred from a model response.

Aspect	Adaptation
Logical input	Six signed integer coordinates, in point order.
Parameter locations	`RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`.
Logical result	Boolean. `AL` must contain 0 or 1. Higher return-register bits are unspecified.
Public example	$A = (0, 0)$, $B = (4, 0)$, $C = (0, 3)$, with expected result `true`.
Memory objects	No input, output, or workspace buffers are supplied.
Public limits	4,096 code bytes; 32,768 response bytes; 4,096 stack bytes; 10,000 instruction units per call.
Retained cases	Three public examples and 20 private logical cases.

The public prompt combines the problem, examples, interface, judging rule, execution environment, resource limits, and submission format. Private expected outputs remain with the evaluator. The candidate performs the geometric decision and instruction encoding. This example illustrates interface conversion, not a successful generated program.

Buffered interface: Relative Position. In *Relative Position* (`lcb_abc320_d`), the evaluator supplies N, M, the original relations as $4M$ signed 64-bit integers, an output buffer of $2N$ such integers, and $32N + 64M$ bytes of workspace. All three buffers are writable. Output and workspace contents are initially unspecified. The candidate computes coordinates relative to person 1 and writes pairs in person order, encoding an undetermined position as $(-2^{63}, -2^{63})$. The marker lies outside the legal coordinate range. The function returns `void`, and the checker compares the complete output exactly. The evaluator precomputes neither coordinates nor connectivity.

B EXTENDED MACHINE CONTRACT

B.1 SUBMISSION GRAMMAR

The submission parser accepts a raw hexadecimal answer body or exactly one complete, non-nested `hex` fence. Fence markers occupy separate lines, allowing surrounding ASCII spaces or tabs and LF or CRLF line endings. The opening marker is three backticks immediately followed by case-insensitive `hex`. The closing marker is three backticks. Multiple submission fences are rejected even if their bytes agree. Text outside a unique valid fence is permitted. Unlabelled and other-language fences supply no candidate.

The candidate contains at least one byte and only hexadecimal digits or ASCII space, tab, carriage-return, and line-feed characters. Each whitespace-separated group contains an even number of digits: `48 c3` is valid, while `4 8 c3` is not. Prefixes, comments, punctuation, Unicode whitespace, and byte-order marks are invalid. The response limit counts UTF-8 bytes including surrounding text. The code limit counts decoded bytes.

Successful extraction records the parser version, path, span, normalized bytes, and SHA-256. `submission_format_valid` applies the parser and size rules. After the response-size check,

`strict_raw_hex_valid` tests raw-hex grammar before fence extraction. This packaging diagnostic is normally false for valid fenced answers and does not affect success.

B.2 ABI AND NUMERICAL STATE

The evaluated implementation uses instruction policy `x86-64-psabi-v2.1`, parser `d15.1`, layout policy `object-layout.2`, and diagnostics `machine-diagnostics.1`.

Integer-class parameters use `RDI`, `RSI`, `RDX`, `RCX`, `R8`, and `R9`, followed by stack slots. Scalar floating-point parameters allocate `XMM0`–`XMM7` independently before overflowing to the stack. At entry, $(\texttt{RSP} + 8) \bmod 16 = 0$. The executor records `RBX`, `RBP`, and `R12`–`R15` and checks their restoration. At the return sentinel, `RSP` must equal its entry value plus eight bytes. Arithmetic flags other than the direction flag are not required to be preserved.

The initial x87 control word is `0x037F` and the x87 stack is empty. The SIMD floating-point control/status register (`MXCSR`) is initialized to `0x1F80`. The x87 control word and MXCSR control bits are preserved at return, while MXCSR exception-status bits are caller-saved. A candidate that enters MMX mode must leave it before returning. Ordinary alignment checking is disabled, but instruction-specific alignment rules still apply.

The implementation uses Unicorn 2.1.4, Capstone 5.0.7, and iced-x86 1.21.0, under `UC_CPU_X86_NEHALEM`. Admitted features include baseline x87/MMX/SSE/SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, CMPXCHG16B, and LAHF/SAHF. AVX, AVX2, BMI, AES, and other unprovided extensions are outside the target. The function environment excludes external I/O, system calls, dynamic linking, thread-local storage (TLS), far transfers, privileged operations, and CPU/device/clock/random services. These are public restrictions, not evidence that models cannot use those capabilities elsewhere.

B.3 MEMORY AND RESOURCE INTERPRETATION

The code base is `0x01000000`, the return sentinel is `0x02000000`, and the stack base is `0x70000000`. Code padding introduced by page rounding is not part of the submitted blob. Objects lie in disjoint slots and have declared alignments and permissions. Low and high object pools begin at `0x10000000` and `0x1010000000`, respectively. Their address ranges do not authorize access to gaps.

Writable output and scratch storage have unspecified initial contents. Reading such storage is not intrinsically illegal, but relying on a particular initial value can lead to a wrong answer in another legal environment. Unused output capacity is normally unobserved. Preserving tails or requiring input immutability must follow the task's explicit semantics. The monitored stack is the entire allocated region, not a hidden moving window tied to the instantaneous stack pointer.

Engineering limits distinguish address-pool extent from mapped memory. Each pool defaults to 16 MiB and can be explicitly enlarged, whereas total mapped guest memory is bounded at 64 MiB. Worker requests and responses are bounded at 32 MiB each. These preflight checks establish that the interface fits the implementation envelope. They do not prove full-domain algorithmic feasibility.

An initial zero-count REP costs one instruction unit. The engine's final zero-count callback is not charged again. Reaching the return sentinel incurs no charge, allowing a correct return on the final permitted instruction.

B.4 OBSERVABLE FAILURE CATEGORIES

Table 5 summarizes the reporting vocabulary. Reports retain case and environment identifiers, observed check outcomes, instruction counts, wall time, and available first-violation details. These records summarize observed failures. They are not complete instruction or memory-access traces.

The observed execution outcomes use the committed diagnostic reports for unchanged response bytes. A reached, validated UD2 instruction is classified as an invalid-instruction exception after normal budget accounting. The package retains engineering regression-test source and versioned assets. A resource-limited Linux worker with seccomp filters alone does not establish that an outer isolation boundary has been independently validated.

Table 5: Failure reporting vocabulary. A stopped execution does not supply counterfactual evidence about later checks.

Stage	Observed condition
Submission	Invalid wrapper or byte grammar, or a response or blob that exceeds its limit.
Instruction	Reached bytes are undecodable or require an unavailable feature.
Memory/control	Unauthorized read/write, alignment violation, or execution outside the allowed code.
Processor/resource	Arithmetic exception or instruction-budget exhaustion.
ABI/return	Incorrect preserved state, return stack, direction flag, floating controls, or MMX state.
Output/function	Invalid output length/encoding, or decoded result rejected by the checker.

C MODEL IDENTIFIERS AND PROVIDERS

Table 6 lists the evaluated model identifiers and provider tags.

Table 6: Service-reported model identifiers and pinned provider tags.

Model identifier	Provider tag
`openai/gpt-6-astra`	`openai`
`anthropic/claude-opus-5.5`	`anthropic`
`x-ai/grok-4.7`	`xai`
`deepseek/deepseek-v4-pro-0813`	`streamlake`
`qwen/qwen3.8-max-prime`	`alibaba`
`moonshotai/kimi-k3`	`moonshotai/mxfp4`
`z-ai/glm-5.3-prime`	`alibaba`