International Journal of Computer Applications (0975 - 8887)

Volume 187 - No.136, August 2026

Benchmarking LLM Tool Orchestration in Resource-Constrained Agentic Workflows: Native MCP Function Calling vs Code Mode

Abasiono Mbat
Pan-Atlantic University
Lagos, Nigeria

Oselumese Agbonrofo
Pan-Atlantic University
Lagos, Nigeria

Oladimeji Abaniwonnda
Pan-Atlantic University
Lagos, Nigeria

Samuel Oyefusi
Worcester Polytechnic Institute
Massachusetts, USA

ABSTRACT

Large Language Model (LLM) agents are commonly implemented as iterative function-calling loops that enable the use of external tools. The Model Context Protocol (MCP) has emerged as the main example of this paradigm. However, function-calling loops often reduce an agent's reasoning ability because of increased token usage and high context occupancy. This study compares native MCP function-calling loops with Code Mode under strict engineering constraints. Six scenarios were constructed comparing MCP with a TypeScript Code Mode environment across eight models, including Claude Opus 4.6, GPT-5.4 and Kimi-K2.5. Results show that Code Mode reduced mean token consumption per representative run by 37.6%, reduced mean cost by 40.7% and raised the individual model pass rate from 59.4% to 63.6%, with per-model cost reductions reaching 70.6% for GPT-5.4. However, these gains are not uniform. Weaker models underperform because compiler errors and execution-contract violations trigger retry loops that raise cost, and Gemini-family models regress sharply under Code Mode, passing roughly two thirds of scenarios under native function calling against roughly one third under Code Mode. Once Gemini models are excluded, the Code Mode advantage widens from 4.2 to 16.7%. Overall, Code Mode manages token usage more efficiently and executes tasks successfully when paired with highly capable models, but degrades when paired with less capable ones. These findings suggest that the choice of a tool orchestration paradigm should be an adaptive runtime decision informed by model capability rather than a fixed architectural choice.

General Terms

Artificial Intelligence, Large Language Models, Intelligent Agents

Keywords

Model Context Protocol (MCP), Code Mode, LLM Tool Use, Agentic Workflows, Agent Benchmarking

1. INTRODUCTION

The growth in the tool-use capabilities of LLMs has transformed them from plain text generators into goal-oriented agents capable of interacting with external environments. LLM tool use is currently enabled by model-native function calling. This was recently standardised with the introduction of the Model Context Protocol (MCP) [3], which was based on the ReAct framework [21]. In this paradigm, the LLM is both the central processor and the control bus. The LLM manages the entire conversation history, including all tool calls.

Native function calling becomes inefficient in constrained environments. As tool calls and tasks grow more complex, the LLM context window fills quickly. This leads to increased cost and network delay. It also weakens the agent's reasoning ability, since the context window becomes bloated with stale tool-call results. Model attention is then allocated to obsolete context rather than to active task constraints. This context bloat leads to the "instruction budget" constraint [8]. Instruction-following capability degrades, causing the model to call tools repetitively and to generate invalid tool calls. Such errors lead to wasted tokens and task failure.

One proposed solution to the issues posed by MCP-based function-calling loops is "Code Mode", introduced by Cloudflare [19]. This paradigm treats the LLM as a program compiler. Available tools are presented to the agent as a programming library. The LLM generates a single unified script, in any programming language, that encodes the entire task logic including data manipulation, control flow and error handling. The generated script is executed in a secure sandbox and its result is returned to

the LLM in a single pass. This approach reduces context bloat by relocating intermediate state into code execution.

Initial industry reports indicate that Code Mode can reduce the number of tokens consumed by an LLM and can eliminate multiple network round trips [19, 2]. However, little research has examined its performance and limits. In addition, no prior work establishes the relative reliability of native function calling and Code Mode under strict resource budgets. This gap matters because Code Mode shifts risk away from many individually faulty tool calls and towards the correctness of a single executable program.

This paper presents a comparison of native MCP function-calling loops against a custom implementation of the Code Mode paradigm built according to Cloudflare's design principles. A shared environment with identical tool definitions is established, and real-world tasks are simulated under strict resource limits. The benchmark enforces a fixed $15 API spend cap, hard caps on task retries, and per-run network latency tracking.

This paper makes three contributions. First, it designs and executes a reproducible framework that compares native MCP function calling and Code Mode under shared tools and strict budget constraints. Second, it categorises and documents the distinct failure modes that arise in each paradigm, including context saturation and tool looping in MCP loops, and compile-time errors, execution-contract violations and retry amplification in Code Mode. Third, it formulates a hybrid deployment policy that guides the selection of an orchestration paradigm from the observed results, together with an efficiency metric that normalises cost and token consumption by validation rate.

Three research questions (RQs) guide this investigation:

—**RQ1**: How do native MCP and Code Mode compare in terms of token consumption, financial cost and round-trip latency under tight resource constraints?

—**RQ2**: To what extent does the success of each paradigm depend on model capacity (frontier versus lightweight)?

—**RQ3**: What are the predominant failure modes in each orchestration style, and how do they affect execution reliability?

The remainder of this paper is organised as follows. Section 2 reviews related work on agent benchmarks, tool-use evaluation and code-based orchestration. Section 3 details the benchmark design, architecture, sandbox internals, observability pipeline and reproducibility protocol. Section 4 presents aggregate, efficiency-normalised, case-sliced and scenario-level results, together with failure diagnostics. Section 5 discusses deployment policy and capability-tier guidance. Section 6 identifies limitations and future directions. Section 7 concludes the paper.

2. RELATED WORK

2.1 Tool Use and Orchestration Paradigms

The fundamental components of tool-using LLM agents lie in reasoning and acting loops such as the ReAct framework [21] and self-supervised tool invocation in Toolformer [18]. Early implementations such as Gorilla [15] and ToolBench [17] scaled these capabilities to large repositories of real-world APIs. However, they relied on iterative function calling [12], in which the model calls tools and receives results across multiple turns. While effective for simple operations, iterative loops suffer from context bloat and increased network latency.

These inefficiencies motivated the idea of offloading execution to programming languages. The use of an external Python interpreter to run arithmetic or symbolic reasoning was first explored in Program-Aided Language models (PAL) [6] and Program of Thoughts (PoT) prompting [5]. The Code Mode paradigm extends this idea by using a sandboxed environment to run multi-step calls to external APIs within a single execution block. TaskWeaver [16] adopted a similar paradigm, but was limited to data analytics rather than a controlled comparison across task families. Modern agent orchestration has recently moved towards standardised interfaces such as MCP [3]. Cloudflare outlined the design concepts of Code Mode atop MCP and demonstrated that type-safe tool definitions can substantially reduce the number of tokens required per task. This study evaluates that paradigm by running multi-step external API calls in a secure local sandbox, using a custom local implementation of Code Mode inspired by Cloudflare's design.

2.2 Agent System Architectures and Security Isolation

LLMs are increasingly treated as central processing units within a larger system, which has led to the formalisation of agent operating systems. MemGPT [14] pioneered this approach by introducing operating-system-like virtual memory management and context swapping, allowing LLMs to maintain long-term state beyond their physical context limits. Similarly, AIOS [7] proposed a comprehensive LLM agent operating system that schedules resources and tasks across multiple agents. Under local implementations of Code Mode, the interpreter and sandbox act as a local system runtime, formalising the interface between the model and the host environment.

Virtualisation solutions such as Firecracker micro VMs [1] provide secure, lightweight isolation, making them well suited to running untrusted model-generated code. These isolation techniques are a prerequisite for safely deploying Code Mode in production environments.

2.3 Cost-Efficient Deployment and Agent Benchmarking

Under limited budgets, cost is a first-class consideration. FrugalGPT [4] formalises cost-reduction techniques for budget-constrained deployments, including prompt adaptation, model cascades and model selection. These strategies are consistent with the need to preserve instruction-following capability under a limited instruction budget [8].

Because resources are consumed differently in each orchestration paradigm, a dynamic evaluation is required to characterise the differences. The general capability of LLM agents on meaningful real-world tasks is evaluated by benchmarks such as AgentBench [11] and TheAgentCompany [20]. MCPAgentBench [10] evaluates the ability of models to use tools through MCP. Task families have expanded from simple software-engineering exercises to more complex settings, including SWE-bench [9], where agents resolve real GitHub issues autonomously inside a sandboxed repository. The present study continues this evaluation tradition, but treats the orchestration mechanism itself as the principal independent variable, comparing native MCP function calling with a Code Mode implementation.

3. METHODOLOGY

3.1 Experimental Design

The benchmark is designed to enable a fair comparison of two end-to-end orchestration architectures for LLM agents. The first architecture is native MCP function calling, which is iterative: tool definitions are supplied to the model, the model emits a

structured tool call, the call is executed, the result is appended to the conversation, and the model is invoked again until the task terminates or the run budget is exhausted. The second architecture, Code Mode, requires the model to write a complete TypeScript program that invokes scenario APIs exposed inside a sandbox. The comparison is therefore not a comparison of prompting styles, but of the execution paths and recovery behaviour of two orchestration mechanisms.

Scenario goals, fixture state, validation logic, model identifiers, timeout configuration, retry limits, logging infrastructure and the result aggregation pipeline are held constant across both architectures. The primary independent variable is the orchestration architecture. Both arms operate on the same initial state and are judged against the same terminal state. Observed discrepancies are attributed to three sources: the orchestration strategy itself, model behaviour under that strategy, and retry behaviour. Differences in outcome are therefore not attributable to variation in task definition.

The benchmark was implemented in TypeScript on the Bun runtime. All model calls are routed through OpenRouter [13], a unified API gateway that provides access to models from multiple providers. The purpose of this routing layer is to standardise model invocation across families so that request construction, response normalisation and usage accounting are identical for every model evaluated.

3.2 Benchmark Scenarios

The benchmark contains six scenarios, summarised in Table 1.

Scenario 1 asks the agent to identify the customer with the highest number of transactions, aggregate that customer's spend over the preceding seven days, and characterise the resulting spending trend. The scenario is backed by a relational database containing pre-seeded records, and the agent output is compared against a ground truth computed directly from that seed data.

Scenario 2 asks the agent to audit the Playwright documentation site at https://playwright.dev and to compare its findings against a deterministic set of expected observations.

Scenario 3 requests a summary of the preceding 24 hours of messages in a simulated Slack channel.

Scenario 4 requires structured extraction over a fixed corpus, producing a result table with the search keywords, speaker, date and matches.

Scenario 5 requires the agent to schedule meetings across participants in multiple time zones, resolving cross-zone availability and existing calendar conflicts.

Scenario 6 requires the agent to retrieve a code repository and apply a set of modifications to it.

Scenarios 2 to 6 use deterministic fixtures or simulated tool responses, which isolates orchestration behaviour from the variability of external services. Provider behaviour and routing are not deterministic: hosted model behaviour and provider infrastructure may change over time, and this is treated as a source of variance rather than as a controlled factor.

3.3 Execution Protocol

Each scenario has two implementations, Regular Mode and Code Mode, which share the same task statement and the same underlying tool implementations.

In Regular Mode, the model receives the task statement together with the JSON schemas of the available tools. The model emits a structured tool call, the harness executes it against the scenario fixture, and the serialised result is appended to the conversation history before the model is invoked again. The loop terminates when the model emits a final answer, when the bounded turn limit is reached, or when a timeout or provider error occurs.

Table 1. Scenario characteristics. "Plan-time control flow" indicates whether the full control flow of the task can be determined from the initial prompt without observing intermediate tool output.

ID	Task family	State source	Plan-time
1	Data aggregation	Seeded database	Yes
2	Live site audit	Live web target	Partial
3	Message summarisation	Fixture channel	Yes
4	Structured extraction	Fixture corpus	Yes
5	Cross-zone scheduling	Fixture calendars	Yes
6	Repository modification	Fixture repository	Yes

In Code Mode, the model receives the task statement together with a typed TypeScript interface describing the same tool set as a callable library. The model emits a single program. The harness compiles the program, executes it inside a sandbox with a restricted import surface and no ambient network access beyond the exposed scenario APIs, and returns the program's structured return value to the model in one pass. Intermediate values produced inside the program are never appended to the conversation history.

Both arms are additionally run under two tool-context configurations. Under *full context*, the complete tool or API surface is supplied in the initial prompt. Under *progressive discovery*, the agent must first enumerate the available surface and then request the definitions it intends to use.

Retries, turn limits and timeouts are bounded by fixed caps that are identical across both arms, and every attempt, whether successful or not, is recorded and billed against the run.

3.4 Evaluation System

The evaluation system is multi-level and combines terminal-state validation with process-level accounting.

Validation is deterministic. Each scenario ships a validator that compares the agent's terminal output, and where applicable the mutated fixture state, against a precomputed ground truth. A run is recorded as passing only when the validator succeeds, the required tool evidence is present in the trace, and the final content is correct.

Token consumption is recorded separately for input and output, summed to a total per attempt, and aggregated per representative run. Cost is recomputed from raw token counts using canonical per-model prices rather than provider-reported charges, which avoids distortion from mixed or promotional pricing. Round trips count the number of model inference calls issued to complete a task. Retry count records the number of repetitions of the model or execution operation after a failure signal. Latency is measured per round trip from recorded run timestamps and is reported separately from token and cost metrics, since it reflects network and provider performance rather than orchestration efficiency alone.

To compare paradigms on a single axis that accounts for both efficiency and reliability, this study defines the Orchestration Efficiency Ratio (OER). For a configuration c with validated fraction V_c and mean total tokens per representative run T_c, the OER is the number of validated runs obtained per one thousand tokens:

$$\mathrm{OER}_c = \frac{V_c}{T_c/1000} \tag{1}$$

Two companion normalisations are also reported, namely the cost and the token consumption required per *validated* run rather than per attempted run:

$$\hat{C}_c = \frac{C_c}{V_c}, \qquad \hat{T}_c = \frac{T_c}{V_c} \tag{2}$$

where C_c is the mean cost per representative run. These normalisations are necessary because a paradigm that is cheap per attempt but frequently fails validation is not cheap in deployment.

3.5 Experiment Matrix and Aggregation

The experimental matrix consists of eight model variants, six scenarios, two orchestration architectures and two tool-context configurations. Each architecture and tool-context pair therefore contributes 48 model–scenario cells, giving 192 base cells before repeated attempts. The realised number of runs exceeds this base count wherever a cell required retries or recovery attempts.
The benchmark records model responses, tool calls, tool results, generated code, execution errors and sandbox errors. Token usage, cost estimates, scenario identifiers, model identifiers, mode identifiers and validator verdicts are persisted as structured artefacts. These logs are used to generate Markdown reports and paired comparison artefacts.
One representative run is reported for each model–scenario–configuration cell. The representative attempt is selected by the ordering described in Section 4.1. This selection rule favours the more favourable end of each cell's distribution, so representative pass rates should be read as representative summaries rather than as unweighted averages over all valid runs. Best-, average- and worst-case aggregates are reported alongside the representative aggregate precisely to expose this sensitivity.

3.6 Failure Taxonomy

The benchmark distinguishes non-model failures from model failures, and further separates paradigm-specific failure classes. Non-model failures arise from infrastructure conditions such as timeouts, database connection problems, and provider or network errors; these are logged and excluded from paradigm attribution. Model failures occur when a run does not satisfy the validator, does not produce a valid output, lacks the required tool evidence, or produces incorrect final content. Code Mode adds a family of execution-contract failures, including disallowed imports, non-compiling programs, sandbox API misuse, invalid return values and retry exhaustion. The full taxonomy, together with the paradigm in which each class occurs and its observable signature, is given in Table 6.

3.7 Reproducibility Protocol

The repository retains a complete implementation record, including deterministic seeding, structured logging, scenario scaffolding, the discovery traces produced in Regular Mode, the sandbox execution artefacts produced in Code Mode, paired comparison artefacts and the result-report generator. Reproduction depends on the commit, the lockfile, the environment setup, the scenario data and the raw artefacts, rather than on commit history alone. Given a fixed repository state, fixed dependency versions, fixed model identifiers, fixed pricing inputs and fixed scenario data, the benchmark is reproducible up to the residual variance introduced by hosted models and provider routing. The benchmark setup and code are available at https://github.com/AJ-505/code-mode.

4. RESULTS

This section reports comparative results between native function calling and code-generated orchestration under the deployment setting defined in Section 3. Unless otherwise stated, reported metrics are means across three runs per model–scenario–architecture cell. Best- and worst-case slices are reported to expose sensitivity to variability and retry behaviour rather than to define the primary conclusions.

4.1 Cost Per Representative Run

A representative run is the single attempt retained for each model–scenario cell when that cell was attempted more than once. The representative attempt is chosen in the following order:

(1) A passing run is preferred over a failing run.
(2) If both attempts share the same outcome, the cheaper attempt is preferred.
(3) If cost is also tied, the attempt with fewer tokens is preferred.

Each model–scenario cell therefore contributes exactly one selected run to the case-level cost. For each paradigm p, let $C_{p,1}$, $C_{p,2}$ and $C_{p,3}$ denote the cost drawn from the best-case, worst-case and average-case tables respectively. The final cost per representative run is:

$$C_p = \frac{C_{p,1} + C_{p,2} + C_{p,3}}{3} \tag{3}$$

4.2 Aggregate Outcomes Across Configurations

Table 2 reports benchmark results across both paradigms and both tool-context configurations. At the all-model paradigm average, Code Mode leads on individual model pass rate (63.6% against 59.4%), on tokens per representative run (2,047 against 3,280, a reduction of 37.6%), on cost (₦7.67 against ₦12.94, a reduction of 40.7%) and on network round trips (1.22 against 1.45, a reduction of 15.9%). It concedes on retry pressure, requiring 2.21 attempts per covered cell against 1.49, an increase of 48.3%.
The four non-average rows of Table 2 also show that the two paradigms respond to the tool-context configuration in opposite directions, a point developed in Section 4.5.

4.3 Efficiency Normalised by Validation Rate

Per-attempt cost understates the true operating cost of a paradigm, because failed attempts are billed but produce no validated output. Table 3 therefore restates the aggregate figures per validated run using the normalisations defined in Section 3.4.
Three observations follow. First, normalisation widens the paradigm-level gap rather than narrowing it: cost per validated run falls by 44.6% under Code Mode (₦21.78 to ₦12.06), tokens per validated run fall by 41.7% (5,522 to 3,219), and the OER improves by a factor of 1.72 (0.181 to 0.311). The reliability advantage of Code Mode is therefore not purchased at a proportionate efficiency cost.
Second, the cheapest configuration in the entire matrix on a per-validated-run basis is Code Mode with progressive API discovery, at ₦8.25, which is 37.2% below the best Regular configuration (₦13.14) despite a lower headline pass rate. Cost-sensitive deployments and reliability-sensitive deployments therefore do not select the same configuration.
Third, the single highest OER in the matrix belongs to Regular Mode without progressive discovery (0.461) rather than to any Code Mode configuration, with Code Mode with progressive

Table 2. Representative-run aggregate outcomes across configurations and paradigms. Lower is better for tokens, cost, round trips and attempts per covered cell.

Configuration	Pass Rate	Tokens/Run	Cost/Run	Round trips	Attempts/Cell
Regular (with progressive discovery)	50.0%	5,066	₦16.83	1.96	1.96
Regular (without progressive discovery)	68.8%	1,493	₦9.04	0.94	1.02
Regular (paradigm average)	59.4%	3,280	₦12.94	1.45	1.49
Code Mode (progressive API discovery)	60.4%	1,497	₦4.98	0.96	2.50
Code Mode (full API context)	66.7%	2,597	₦10.35	1.48	1.92
Code Mode (paradigm average)	63.6%	2,047	₦7.67	1.22	2.21

Table 3. Efficiency normalised by validation rate, derived from Table 2. Lower is better for tokens, cost and attempts per validated run; higher is better for OER.

Configuration	Tokens/validated run	Cost/validated run	Attempts/validated run	OER
Regular (with progressive discovery)	10,132	₦33.66	3.92	0.099
Regular (without progressive discovery)	2,170	₦13.14	1.48	0.461
Regular (paradigm average)	5,522	₦21.78	2.51	0.181
Code Mode (progressive API discovery)	2,479	₦8.25	4.14	0.403
Code Mode (full API context)	3,894	₦15.52	2.88	0.257
Code Mode (paradigm average)	3,219	₦12.06	3.47	0.311

discovery close behind (0.403). This is the clearest evidence in the study against reading the paradigm averages as a universal dominance claim: when a small tool surface is supplied in full and the loop is short, iterative function calling remains highly token-efficient per validated task. The Code Mode advantage at paradigm level is driven substantially by how badly Regular Mode degrades under progressive discovery, which is examined next.

4.4 Best, Average and Worst Case Dynamics

The case-slice validation profile is given in Figure 1. In the best-case slice, Code Mode leads on pass rate, at 66.4% against 60.5%. In the average- and worst-case slices it does not sustain that lead over Regular Mode without progressive discovery, because weaker model families enter repeated retry cycles and fail to recover within the retry budget.

Latency per round trip is consistently higher in Code Mode across all three slices, as shown in Figure 2. This is driven by regeneration overhead when execution contracts fail, rather than by sandbox execution itself, since sandbox execution is local and contributes no provider round trip.

Token consumption, shown in Figure 3, is lowest for Code Mode in the best case (2,045 against 3,380, a reduction of 39.5%) and in the average case (2,136 against 2,918, a reduction of 26.8%), but rises above Regular Mode in the worst case (2,957 against 2,586, an increase of 14.3%). The direction of the token advantage therefore reverses between the average and worst slices, which confirms that retry amplification can erase the nominal context-efficiency gain of compiled orchestration.

The spread across slices is itself informative. Regular Mode token consumption varies by 794 tokens between its best and worst slices, a spread of 26.8% of its mean across slices, whereas Code Mode varies by 912 tokens, a spread of 38.3%. Compiled orchestration is thus both cheaper on average and more variable, which matters for capacity planning under a fixed budget: the same spend buys more expected work but with a wider tail.

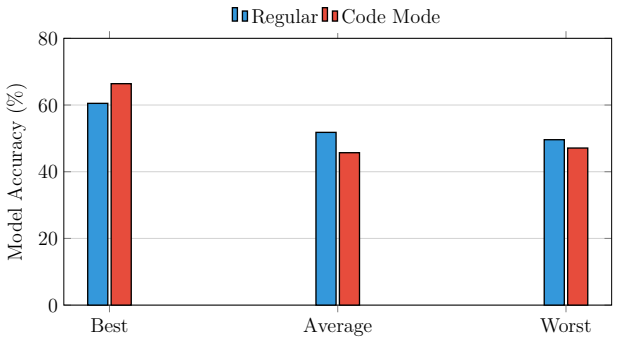

Fig. 1. Best-, average- and worst-case individual model pass rates by paradigm, aggregated across both tool-context configurations

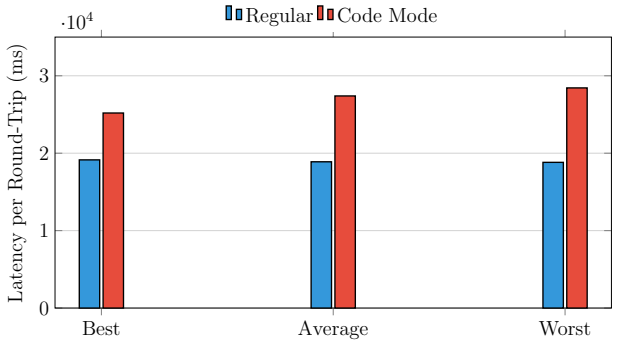

Fig. 2. Best-, average- and worst-case network latency per round trip by paradigm

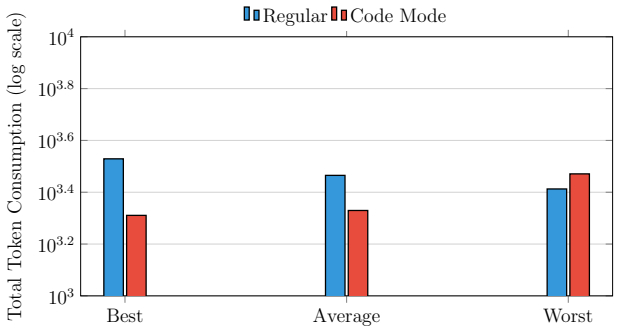

Fig. 3. Best-, average- and worst-case total token consumption by paradigm, log scale

4.5 Effect of Progressive Discovery

Progressive discovery reduced pass rates in both paradigms, but its cost effect runs in opposite directions, and this asymmetry is one of the more consequential findings of the benchmark.

Under Regular Mode, progressive discovery is doubly harmful. It lowers the pass rate by 18.8% (68.8% to 50.0%) while simultaneously inflating token consumption by a factor of 3.39 (1,493 to 5,066) and cost by a factor of 1.86 (₦9.04 to ₦16.83). The mechanism is visible in the round-trip column of Table 2: discovery turns are themselves model turns, so each enumeration and definition-fetch step adds a full inference round trip and permanently occupies context. Round trips more than double, from 0.94 to 1.96.

Under Code Mode, the same configuration lowers the pass rate by only 6.3% (66.7% to 60.4%) while *reducing* token consumption by 42.4% (2,597 to 1,497) and cost by 51.9% (₦10.35 to ₦4.98). The cost of discovery is largely absorbed inside the generated program rather than paid in model turns. The penalty appears instead as retry pressure, which rises from 1.92 to 2.50 attempts per covered cell, an increase of 30.2%, because a model that has seen only part of the API surface is more likely to emit a program that violates the execution contract.

The prior claim that progressive discovery is uniformly harmful therefore requires qualification. It is uniformly harmful to reliability at this scale, but its efficiency cost is paradigm-dependent: it is paid in context under iterative function calling and in retries under compiled orchestration. The reliability penalty is plausibly an artefact of scale, since for small tool suites the overhead of discovery outweighs any reduction in context load. In agentic systems where a very large tool surface cannot fit in the context window, the efficiency gains of progressive discovery would be expected to dominate, and the results here suggest that compiled orchestration is the better host for that strategy because it converts discovery overhead into program logic rather than into conversation turns.

4.6 Gemini Outlier Effect

Gemini-family models behaved differently from every other family evaluated, and results are therefore reported both with and without them. Removing both Gemini models clarifies the dominant trend: Code Mode moves from a modest lead of 4.2% to a large lead of 16.7%.

Table 4 summarises the effect of including Gemini in the aggregate. The performance of Gemini suggests that orchestration strategy cannot be treated as a fixed architectural commitment and must instead be tailored to model family and capability tier.

Table 4. Gemini effect on paradigm-level model pass rates.

Slice	Cells	Reg.	Code	Δ
All models	48	59.4%	63.6%	+4.2
Non-Gemini	36	56.9%	73.6%	+16.7

4.7 Model-Level Outcomes and Routing Implications

Table 5 summarises the qualitative outcome for each evaluated family together with the paradigm it should be routed to. Three patterns emerge. Frontier models complete every scenario under Code Mode with full API context and do so more cheaply than under native function calling: GPT-5.4 achieved 6/6 best-case coverage at ₦3.73 per representative run against a Regular range of ₦4.35 to ₦12.70, and Claude Opus 4.6 achieved the same coverage at ₦11.05 against a Regular range of ₦17.81 to ₦35.36. At the low end of those Regular ranges the saving is 14.3% for GPT-5.4 and 38.0% for Claude Opus 4.6; at the high end it reaches 70.6% and 68.8% respectively.

Mid-tier models show mixed behaviour, improving on some scenarios under Code Mode while increasing retry exposure on others. Gemini-family models regress sharply, as quantified in Section 4.6.

Table 5. Model-level outcome pattern and recommended default paradigm.

Model	Code Mode outcome	Default
GPT-5.4	Lower cost	Code Mode
Claude Opus 4.6	Lower cost	Code Mode
Claude Sonnet 4.6	Advantageous	Code Mode
Kimi-K2.5	Advantageous	Code Mode
GLM-5.1	Advantageous	Code Mode
Gemini 3.1 Pro Preview	Sharp regression	Regular
Gemini 3.1 Flash Lite Preview	Sharp regression	Regular
GPT-5.4 mini	Sharp regression	Regular

4.8 Failure Diagnostics

Table 6 sets out the failure taxonomy defined in Section 3.6 alongside the paradigm in which each class arises. The dominant expensive failure modes are retry amplification and tool-invocation ordering errors rather than sandbox compute cost, which is local and negligible relative to inference. This is reflected in the higher Code Mode retry figure in the average-case slice (0.64 against 0.00 for Regular) and in the degraded worst-case token profile reported in Figure 3.

The asymmetry of recovery cost between paradigms is the key diagnostic. Under native function calling, a failed tool call invalidates one step and the agent may retry that step alone. Under compiled orchestration, a contract violation invalidates the entire program, so every tool interaction the program encoded must be regenerated. The unit of failure is therefore the step in one paradigm and the whole plan in the other, which explains why the same retry count carries a much larger token penalty in Code Mode.

Table 6. Failure taxonomy and paradigm attribution. Infrastructure failures are logged but excluded from paradigm attribution.

Failure class	Paradigm
Infrastructure or provider	Both
Validator mismatch	Both
Missing tool evidence	Both
Incorrect final content	Both
Context saturation	Regular
Tool looping	Regular
Compile-time error	Code
Contract violation	Code
Sandbox API misuse	Code
Retry exhaustion	Code

4.9 Answers to Research Questions

RQ1. Under the evaluated resource constraints, compiled orchestration is more efficient at paradigm level on every cost axis except retries. It reduces tokens per representative run by 37.6%, cost by 40.7% and round trips by 15.9%, while increasing attempts per covered cell by 48.3%. Normalising by validation rate strengthens rather than weakens the result, with cost per validated run falling 44.6% and the OER improving by a factor of 1.72. The exception is latency, which is consistently higher per round trip under Code Mode because of program regeneration overhead.
RQ2. Outcomes differ sharply by model family and capability tier, and the difference is large enough to invert the architecture ranking. Frontier models complete all six scenarios under Code Mode at lower cost while GPT-5.4 mini, Gemini 3.1 Pro Preview and Gemini 3.1 Flash Lite Preview favour MCP-based function calling. Any paradigm ranking that is not stratified by model family is therefore misleading.
RQ3. The dominant expensive failure modes are retry amplification and tool-invocation ordering errors, not sandbox compute cost. Because a contract violation invalidates an entire program rather than a single step, retry cost in Code Mode scales with plan size rather than with step size, which is what produces the worst-case token reversal.

5. DISCUSSION

This study presents a repeatable systems-level benchmark comparing native function-calling loops with code-generated orchestration across six agentic workflows and eight model variants routed through OpenRouter. Across the representative aggregate in Table 2, Code Mode achieved a higher paradigm-average pass rate than native function calling (63.6% against 59.4%) while reducing average tokens per representative run from 3,280 to 2,047 and average cost from ₦12.94 to ₦7.67. These results support the central claim that compiling orchestration into an executable program can reduce context growth and cost in multi-step tool workflows.
The aggregate result should not be read as a universal dominance claim, for two independent reasons. The first is model heterogeneity. The benchmark reveals three qualitatively different family patterns: GPT-5.4, Claude Opus 4.6, Claude Sonnet 4.6, Kimi-K2.5 and GLM-5.1 show that Code Mode is advantageous when the model reliably emits programs that satisfy the sandbox contract; Gemini-family models form a clear outlier group that performs competitively under native function calling and regresses under Code Mode; and less capable models show mixed behaviour in which Code Mode improves selected scenarios but increases retries elsewhere. The second reason is configuration heterogeneity. As Table 3 shows, the highest OER in the matrix is achieved by native function calling with a fully supplied tool surface, not by Code Mode. The relevant deployment decision is therefore not whether Code Mode is better in the abstract, but whether a particular model, workflow and tool-context triple has enough executable-code reliability to justify dispatch through Code Mode.

5.1 Compiled Orchestration and Iterative Tool Use

To interpret the gains above it is first necessary to clarify how Code Mode changes tool usage. Code Mode does not eliminate tool usage; it relocates it. Regular Mode performs a separate model inference at each tool invocation: a structured call is emitted and executed, the resulting value is appended to the conversation history, and the model is invoked again. Code Mode instead has the model emit a program that invokes scenario API functions internally during sandbox execution. Intermediate values are consumed by the program's own control flow rather than added to the model's context.
The consequence is that Code Mode removes per-turn inference cost and, with it, the compounding context cost of intermediate results. It is therefore well suited to workflows whose control flow can be determined from the initial prompt, including data aggregation, deterministic retrieval, dispatch and scheduling, and repository changes. It is correspondingly less suited to problems that require an agent to observe unexpected intermediate states, revise plans mid-execution, or perform extended exploration, because in those settings the information needed to write a correct program is not available at generation time.

5.2 Token Reduction and Retry Amplification

The primary advantage of Code Mode is consolidation: a single program generation step replaces several sequential inference steps. The benchmark supports this expectation in the best- and average-case slices, where Code Mode required 2,045 tokens against 3,380 and 2,136 tokens against 2,918 respectively.
The worst-case slice locates the boundary beyond which consolidation provides no net advantage. Retry amplification occurs when a model emits invalid TypeScript, breaks the execution contract or misuses the sandbox API, so that additional model calls are required to regenerate the program. The output tokens spent on failed programs then erase the nominal context-efficiency gain, producing the 14.3% token penalty observed in the worst case. This failure family is specific to Code Mode: a program is designed to collapse many tool interactions into one interaction, so when it fails, all of the interactions it encoded must be regenerated together. The practical implication is that the retry budget, not the sandbox, is the cost control surface in Code Mode deployments, and that bounding it tightly is what preserves the average-case advantage.

5.3 Gemini Outlier Effect

The Gemini family produces a pronounced outlier effect. Including these models yields a Code Mode advantage of only 4.2%; excluding them yields 16.7%, as shown in Table 4. Capability at tool use in the iterative sense evidently does not predict reliability at program emission under a strict execution contract, and the two should be measured separately when selecting a routing policy.

5.4 Deployment Policy Implications

The benchmark supports a capability-aware orchestration policy with three rules. Models that achieve high validation rates with low retry pressure under Code Mode should be routed to Code Mode for workflows whose logic is known at plan time. Models exhibiting Code Mode regression, in particular the Gemini family observed here, should default to native function calling. In ambiguous cases a bounded fallback applies: dispatch to Code Mode only when recent measured reliability is high, abort after a short retry budget, and revert to native function calling when repeated execution-contract violations occur.

Table 3 adds a fourth consideration, namely that the tool-context configuration should be selected jointly with the paradigm rather than independently of it. Progressive discovery is the correct default only under Code Mode, where it yields the lowest cost per validated run in the matrix at ₦8.25, and is clearly contraindicated under native function calling, where it produces the worst cell in the matrix at ₦33.66 per validated run.

This policy is data-driven and specific to the architecture benchmarked here. The six workflows evaluated are retrieval-and-dispatch tasks with deterministic validators. Such workflows favour compiled execution because most of the task can be expressed as a program that runs from start to finish. Other workflow types, including open-ended exploration and settings with adversarial or mutable intermediate state, may produce a different ordering of paradigms.

6. LIMITATIONS AND FUTURE WORK

Although this study provides a first-order comparison of two orchestration paradigms, it represents an early characterisation, and the comparisons should be read as indicative rather than definitive.

Limited repetition and statistical power. Each model–scenario–architecture combination was executed a small number of times. This is sufficient to expose stable failure patterns and cost trends, but not sufficient for strong statistical inference about stochastic model behaviour. Future work should increase repetitions per cell and report confidence intervals for differences in pass rate, token consumption and cost. Paired tests such as McNemar's test for binary pass and fail outcomes, and permutation tests for token distributions, would strengthen claims about paradigm preference.

Limited prior-work baselines. This study compares native function calling with Code Mode but does not yet include enough alternative agent baselines. Natural comparisons include ReAct-style loops, plan-and-execute scaffolds, structured function-calling frameworks, and multi-shot tool-use prompting in which examples of correct tool sequences are supplied in the prompt. These baselines are needed to separate the architectural value of Code Mode from improvements recoverable through better prompting alone.

Need for formalisation. The boundary between compiled orchestration, interactive tool use, conventional workflow execution and DAG-based scheduling is not yet formalised here. A formal computational model would sharpen the paradigm boundary and would allow future work to analyse when Code Mode should dominate as a function of workflow depth, tool size, intermediate state size and dependency structure.

Granularity of the efficiency metric. The Orchestration Efficiency Ratio is reported at configuration level in Table 3. Future revisions should report it per model and per scenario, together with input tokens, output tokens and validation counts, so that the metric is independently auditable at cell level.

One-shot execution assumption. Code Mode is evaluated here on tasks that can be specified from the initial prompt. This excludes a broader class of agentic workflows requiring exploratory reasoning, dynamic hypothesis testing, conditional replanning after unexpected tool output, or recovery from partial execution. Future benchmarks should include these workflows to determine where iterative tool use remains structurally preferable.

Failure taxonomy coverage. The taxonomy in Table 6 separates model from non-model failures and identifies retry amplification as a Code Mode-specific risk, but it records failure classes rather than per-class frequencies. More complex agent systems may also require categories for long-horizon planning failure, inconsistent mutable state, partial-execution recovery failure, and globally invalid plans composed of locally valid steps.

Scenario diversity and simulated integrations. Scenarios 2 to 6 use deterministic fixtures or simulations. This improves repeatability and isolates orchestration behaviour, but limits ecological validity. Future work should add production-like integrations in domains such as customer support routing, procurement approval, multi-party document review, iterative software engineering and electronic design automation. The last of these is a particularly relevant extension because it combines persistent simulation state, deterministic validation, design-rule checking, rollback semantics and coordination across heterogeneous toolchains.

Model version and provider sensitivity. The results are a snapshot of specific model versions routed through OpenRouter. Model behaviour, provider routing, retry behaviour, response normalisation and pricing can all change over time. Longitudinal reruns and direct-provider comparisons are needed before treating any model-family ranking as stable.

Cost repricing assumptions. Costs are recomputed from raw token counts using canonical per-model prices. This avoids mixed-price distortion but does not model deployment-specific discounts, prompt caching, cached-input pricing or provider-specific billing behaviour.

Future work should therefore expand repeated trials, add baseline frameworks, report the OER at cell level, formalise compiled orchestration, conduct workflow-structure ablations, add domain-specific scenarios, and implement an adaptive hybrid controller that begins in the empirically preferred paradigm and falls back when bounded retry policies are exceeded.

7. CONCLUSIONS

This paper presented a reproducible benchmark that isolates the systems-level trade-off between native function-calling loops and code-generated orchestration for LLM agents. The main result is conditional rather than universal. Code Mode improved token efficiency and validation reliability at paradigm level, reducing tokens per representative run by 37.6%, cost by 40.7% and cost per validated run by 44.6%, but it regressed when program-generation failures triggered retry amplification, and it was outperformed on the orchestration efficiency ratio by native function calling with a fully supplied tool surface. The failure unit differs between the paradigms, a step in one and an entire plan in the other, and this asymmetry explains both the average-case advantage and the worst-case reversal.

The practical implication is that orchestration strategy should be treated as a runtime control decision informed by measured model reliability rather than as a fixed architectural commitment. For model families such as GPT-5.4, Claude Opus 4.6, Claude

Sonnet 4.6, Kimi-K2.5 and GLM-5.1, the benchmark provides evidence that Code Mode is effective for deterministic retrieval and dispatch workflows. For Gemini-family models in this benchmark, which passed roughly two thirds of cells under native function calling but only about one third under Code Mode, native function calling is the safer default. More broadly, the study shows that the right abstraction is not a fixed choice between function calling and Code Mode, but an adaptive controller that selects the orchestration paradigm according to model capability, workflow structure, tool-context configuration, retry behaviour and validated cost.

8. REFERENCES

[1] A. Agache, M. Brooker, A. Iordache, F. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Marin. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, pages 419–434, 2020.

[2] Anthropic. Code execution with mcp, 2026. Accessed: February 2, 2026.

[3] Anthropic. Model context protocol documentation, 2026. Accessed: January 18, 2026. https://modelcontextprotocol.io/docs/getting-started/intro.

[4] L. Chen, M. Zaharia, and J. Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. In *Proceedings of the 40th International Conference on Machine Learning (PMLR)*, volume 202, pages 4822–4846, 2023.

[5] W. Chen, X. Chang, T. Schick, and W. Y. Wang. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

[6] L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig. Pal: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning (PMLR)*, volume 202, pages 10764–10799, 2023.

[7] K. Ge, Y. Zhao, J. Sheng, Y. Jiang, C. A. Yang, K. Yang, X. Wang, Y. Wang, et al. Aios: Llm agent operating system. *Journal of Machine Learning Research (COLM 2025)*, 2025.

[8] D. Jaroslawicz, B. Whiting, P. Shah, K. Maamari, et al. How many instructions can llms follow at once? *arXiv preprint arXiv:2507.11538*, 2025.

[9] C. E. Jimenez, K. R. Narasimhan, J. Boyreau, R. Yang, Y. Xu, P. Rosand, S. Xia, J. Zhao, et al. Swe-bench: Can language models resolve real-world github issues? In *Proceedings of the Twelfth International Conference on Learning Representations*, 2024.

[10] W. Liu, Y. Zhang, X. Wang, H. Chen, J. Li, Q. Zhao, T. Yang, and Z. Huang. Mcpagentbench: A real-world task benchmark for evaluating llm agent mcp tool use. *arXiv preprint arXiv:2512.24565*, 2025.

[11] X. Liu et al. Agentbench: Evaluating llms as agents. In *Proceedings of the Twelfth International Conference on Learning Representations*, 2024.

[12] OpenAI. Function calling and tool use in the openai api, 2026. Accessed: February 4, 2026. https://platform.openai.com/docs/guides/function-calling.

[13] OpenRouter. Unified api for llm providers, 2026. Accessed: February 10, 2026. https://openrouter.ai/docs.

[14] C. Packer, V. Fang, S. G. Patil, K. Lin, S. Wooders, and J. E. Gonzalez. Memgpt: Towards llms as operating systems. In *Proceedings of the 2024 ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2024.

[15] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

[16] B. Qiao et al. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*, 2023.

[17] Y. Qin et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *Proceedings of the Twelfth International Conference on Learning Representations*, 2024.

[18] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, pages 68539–68551, 2023.

[19] K. Varda and S. Pai. Code mode: The better way to use mcp. Cloudflare Blog, 2025. Accessed: January 29, 2026. https://blog.cloudflare.com/code-mode/.

[20] F. F. Xu et al. Theagentcompany: Benchmarking llm agents on consequential real world tasks. *arXiv preprint arXiv:2412.14161*, 2024.

[21] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.