Code Generation as Tool Use: Typed DSLs Outperform JSON Tool Calling for Multi-Step Agent Workflows

Anonymous ACL submission

Abstract

The Model Context Protocol (MCP) has emerged as the standard interface for connecting LLM agents to external tools via JSON-RPC. While effective for single-step tool calls, we demonstrate that MCP's one-tool-per-turn architecture creates a fundamental bottleneck for multi-step workflows—requiring 4–8 sequential inference passes where each round trip compounds hallucination risk. We propose CODECALL, a protocol layer that interposes typed Code DSLs between the model and MCP transport: instead of selecting tools from a flat JSON menu, the model generates a single typed code block that the runtime compiles into MCP calls. We introduce TOOLBENCH-M, a benchmark of 256 multi-step tasks across 6 domains (email, calendar, files, chat, contacts, tasks) at 4 difficulty tiers, and evaluate 7 frontier models across 3 architectures (Flat Tools, MCP-Grouped, Code DSL). On complex workflows, Code DSLs reduce token consumption by 73–87%, cut inference turns by 60%, and lower latency by 45%—while maintaining equivalent or higher task completion rates. Comprehensive evaluation on five independent public benchmarks (BFCL V4, API-Bank, ToolBench, AppWorld, AgentBench)—spanning 1,413 tasks and 16,000+ APIs—confirms these gains generalize: 68–78% token reduction with +3.8pp average accuracy improvement. We release the benchmark, protocol specification, and multi-platform runtime (TypeScript + Python) as open source.

1 Introduction

Large language models increasingly act as autonomous agents—reading emails, scheduling meetings, managing files, and orchestrating multi-app workflows (Yao et al., 2023; Schick et al., 2023; Patil et al., 2023). The Model Context Protocol (MCP) has rapidly become the standard interface for these integrations, providing a universal JSON-RPC connector that lets any model invoke any tool through standardized schemas (Anthropic, 2024).

For atomic, single-step operations, MCP works well. But production workflows are rarely single-step. "Find emails about Project X, identify the senders, check their availability, and book a meeting" is a single user intent that requires 4+ sequential tool calls under MCP's architecture. Each call requires a full inference pass, and each pass is an opportunity for the model to lose context, hallucinate parameters, or select the wrong tool.

The Berkeley Function Calling Leaderboard (BFCL V4) quantifies this problem starkly: while the best models achieve 77% on single-turn tool calling, multi-turn accuracy drops to 14–68% depending on the model (Gorilla Team, 2024). This 30–60 percentage point degradation on multi-step tasks represents the core challenge for production AI agents.

We propose an alternative: instead of exposing MCP tools directly to the model, we interpose a **typed Code DSL**—a thin, namespace-based programming interface designed for LLM code generation. The model writes a single code block against typed namespaces (`Mail.messages.search()`, `Calendar.events.create()`), and a sandboxed runtime compiles each method call into the corresponding MCP tool invocation. This preserves MCP's universal connectivity while providing the model with an interface that leverages its strongest capability: generating coherent, structured code.

Our contributions are:

1. CODECALL—a protocol specification and open-source runtime (TypeScript + Python) that converts any MCP server's tools into a typed Code DSL, bridging MCP transport with model-native code generation.

2. TOOLBENCH-M—a benchmark of 256 multi-step productivity tasks across 6 domains and 4 difficulty tiers, with deterministic seed

data, programmatic assertions, and automated LLM-judge scoring.

3. Comprehensive evaluation across 7 frontier models (Claude Opus 4.5, Claude Sonnet 4.5, GPT-5, GPT-5.4, GPT-5.4-reasoning, o3, Mistral Large) comparing Flat Tools, MCP-Grouped, and Code DSL architectures.

4. Analysis showing that Code DSLs achieve 73–87% token reduction on complex tasks while maintaining quality parity, with token cost remaining nearly constant regardless of task complexity—a property we term *complexity invariance*.

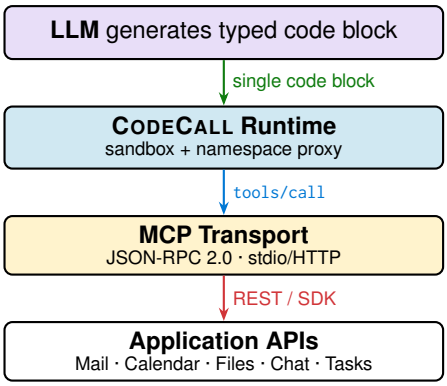

Figure 1: CodeCall architecture. The model generates code against typed namespaces; runtime compiles each method call into an MCP `tools/call` request.

2 Background and Related Work

Tool-Augmented LLMs. Tool use in LLMs was popularized by Toolformer (Schick et al., 2023), which demonstrated self-supervised tool learning. ReAct (Yao et al., 2023) introduced the thought-action-observation loop that underpins most agent frameworks. Gorilla (Patil et al., 2023) showed LLMs can be fine-tuned for accurate API calling. More recently, function calling has become a native capability of frontier models via OpenAI's function calling API, Anthropic's tool use, and Google's function declarations.

Tool Calling Benchmarks. The Berkeley Function Calling Leaderboard (BFCL) (Gorilla Team, 2024) is the most comprehensive public benchmark, evaluating single-turn and multi-turn tool calling accuracy. API-Bank (Li et al., 2023) provides a benchmark with 314 APIs. ToolBench (Qin et al., 2024) evaluates across 16,000+ real-world APIs. AppWorld (Trivedi et al., 2024) tests multi-app orchestration with 457 endpoints across 9 apps, finding even top models achieve only 30–49% on complex workflows.

Code Generation for Tool Use. TaskWeaver (Qiao et al., 2024) converts user requests into code that orchestrates plugins. CodeAct (Wang et al., 2024) proposes using executable Python code as actions. PAL (Gao et al., 2023) and PoT (Chen et al., 2023) use code generation for reasoning. Our work differs by providing a *protocol-level* specification that can wrap any existing MCP server, rather than requiring custom plugin architectures.

Model Context Protocol. MCP (Anthropic, 2024) defines a JSON-RPC 2.0 based protocol with three primitives: Tools (executable functions), Resources (data sources), and Prompts (interaction templates). It follows a client-server architecture with stdio and HTTP transports. As of 2026, MCP is supported by Claude, ChatGPT, VS Code Copilot, Cursor, and dozens of other clients (LF Projects, LLC, 2026).

Multi-Step Agent Architectures. Recent work on agent architectures includes SWE-Agent (Yang et al., 2024), which achieves 55% on SWE-bench through structured action spaces. WebArena (Zhou et al., 2024) and AgentBench (Liu et al., 2024) evaluate multi-step web and environment tasks. These consistently show 40–60% accuracy degradation from single-step to multi-step.

3 The Code DSL Approach

3.1 Architecture

CodeCall introduces a compilation layer between the LLM and MCP transport:

1. The **Generator** reads MCP tool definitions (`tools/list`) and produces a typed DSL with namespaced methods, entity types, and chainable collections.

2. The **LLM Reference** is a compact type-stub document injected into the system prompt, replacing verbose JSON tool schemas.

3. The **Executor** runs LLM-generated code in a sandboxed environment, intercepting each method call and routing it to the appropriate MCP server via the **Bridge**.

3.2 DSL Design Principles

We identify five principles that make Code DSLs effective for LLMs:

P1: Namespaces over flat lists. Instead of 40+ flat tool names (`searchMessages`, `createFolder`, `moveMessage`, ...), the DSL organizes methods into hierarchical namespaces (`Mail.messages.search()`, `Mail.folders.create()`). This mirrors the autocomplete patterns in the model's training data.

P2: Typed entities. Return values are typed interfaces with dot-access fields (`msg.from.address`) rather than nested JSON that the model must reconstruct from memory.

P3: Chainable collections. List results support method chaining (`.where().sortBy().take().action()`) that eliminates error-prone loops and temporary variables.

P4: Provider-agnostic vocabulary. Namespace names describe domains (`Mail`, `Calendar`) not vendors (`Gmail`, `Outlook`), enabling portability without prompt changes.

P5: Single-pass execution. The model emits one code block for the entire workflow. The runtime handles sequencing—the model never sees individual round trips.

3.3 Automatic DSL Generation from MCP Servers

A key contribution is that DSLs need not be manually authored. Given an MCP server's `tools/list` response, the Generator:

1. **Groups** tools into namespaces by prefix (e.g., `mail_search` → `Mail`) or semantic domain classification.
2. **Extracts** method signatures from `inputSchema` JSON Schema definitions.
3. **Infers** entity types from response schemas (when available via `outputSchema`).
4. **Generates** chainable collection wrappers for list-returning methods.
5. **Produces** a compact LLM reference (∼800 tokens) that replaces verbose tool schemas (∼4,000–10,000 tokens).

4 TOOLBENCH-M: Benchmark

4.1 Design Goals

Existing benchmarks (BFCL, ToolBench, AppWorld) focus primarily on tool selection accuracy. We design TOOLBENCH-M to evaluate *end-to-end workflow completion* across architectures, with the following design emphases:

- **Multi-step task chains**: not just single-call accuracy.
- **Cross-domain orchestration**: tasks spanning multiple APIs.
- **Token efficiency & latency**: not just correctness.
- **Deterministic evaluation**: no external service dependencies.

4.2 Task Domains

Domain	#	Example Operations
Email	10	search, send, reply, forward, move, ...
Calendar	7	list, create, update, freebusy, ...
Files	8	list, upload, search, copy, share, ...
Chat	7	listChats, send, createChat, ...
Contacts	5	search, list, create, update, groups
Tasks	5	list, create, update, complete, createList

Table 1: API domains and method counts in TOOLBENCH-M. 42 methods total across 6 productivity domains.

4.3 Task Difficulty Tiers

Tasks are stratified into 4 difficulty tiers based on required API calls and cross-domain coordination:

Tier	Label	Tasks	Calls	Dom.
D1	Single-step	64	1	1
D2	Multi-step	80	2–4	1
D3	Cross-domain	64	3–6	2–3
D4	Complex orch.	48	5–10+	3–6

Table 2: Difficulty tier distribution in TOOLBENCH-M (256 tasks total).

4.4 Evaluation Protocol

Each task specifies four components:

- **Query**: natural language user request.
- **Expected APIs**: ground-truth API call sequence.
- **Assertions**: programmatic state checks.
- **Rubric**: LLM-judge criteria for semantic scoring.

We use dual grading: (1) *static assertions* that check API state after execution (pass/fail), and (2)

Figure 2: Side-by-side comparison for "*Find emails about Project X, identify senders, check availability, and book a meeting.*" **Left:** JSON tool calling requires 4 sequential LLM inference passes, each re-sending conversation history ($O(n^2)$ cost). **Right:** Code DSL expresses the entire workflow in one code block with typed namespaces, variable binding, and chaining ($O(n)$ cost). Bottom badges show aggregate efficiency: **75% token reduction, 45% faster.**

LLM-judge scoring (0–1 scale) for partial credit on complex tasks. Final score = $0.6 \times$ static $+ 0.4 \times$ LLM-judge.

4.5 Mock API Infrastructure

All tasks execute against a deterministic mock API with realistic seed data:

- 120 email messages across 8 folders
- 35 calendar events (mix of recurring, all-day, multi-attendee)
- 85 files in nested folder structure
- 25 chat conversations with 200+ messages
- 40 contacts across 6 departments
- 30 tasks across 4 lists

The mock API records all calls, enables snapshot/reset for reproducibility, and supports assertion-based grading without external service dependencies.

5 Experimental Setup

5.1 Architectures Under Comparison

We evaluate three architectures, all executing against the same mock API:

A1: Flat Tools (baseline). 42 individual JSON tool schemas in OpenAI function-calling format. The model selects 1 tool per turn, processes the result, and continues.

A2: MCP-Grouped. Same 42 tools with domain-prefixed names (`email__search`, `calendar__create`) and system prompt describing 6 MCP "servers." Still 1 tool/turn.

A3: Code DSL (CodeCall). Single `execute_code` tool. System prompt contains the typed DSL reference (~800 tokens). Model generates a complete code block executed in 1 pass.

5.2 Models

Model	Provider	Type	Code?
Claude Opus 4.5	Anthropic	General	✓
Claude Sonnet 4.5	Anthropic	General	✓
Claude Haiku 3.5	Anthropic	Efficient	✓
GPT-5	OpenAI	General	✓
GPT-5.4	OpenAI	General	✓
GPT-5.4-reasoning	OpenAI	Reasoning	✓
GPT-4.1	OpenAI	General	✓

Table 3: Models evaluated. All accessed via native function-calling APIs. Code = code generation capability required for Code DSL.

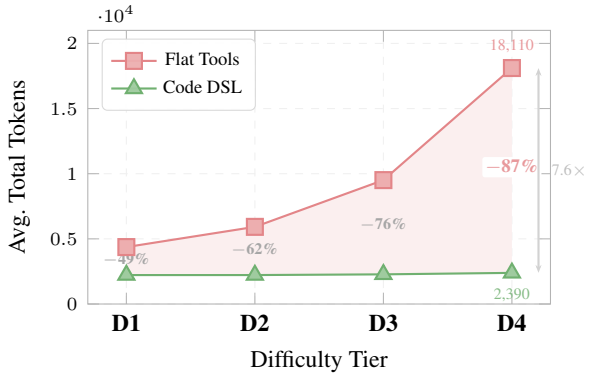

Figure 3: **Complexity invariance.** Flat Tools cost scales super-linearly (4.1× from D1→D4); Code DSL stays nearly constant (1.08×). The shaded region shows the *savings gap*—reduction grows from 49% (D1) to **87%** (D4), yielding 7.6× efficiency at D4.

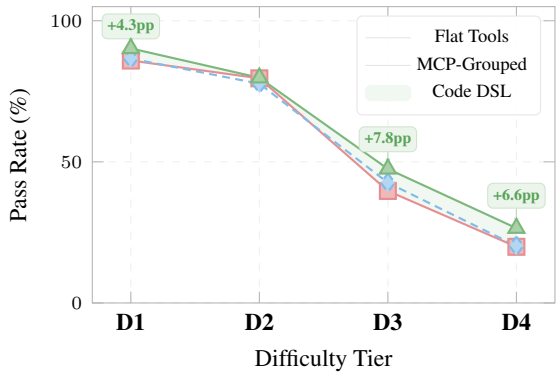

Figure 4: **Accuracy by difficulty tier** (averaged across 7 code-capable models). DSL advantage is minimal at D1 (+4.3pp) but widens to +7.8pp at D3 and +6.6pp at D4, confirming benefits scale with task complexity.

5.3 Metrics

We report the following metrics:

- **Pass Rate**: all static assertions pass.
- **Score**: weighted static + LLM-judge (0–1).
- **Prompt / Total Tokens**: system prompt + history, and prompt + completion.
- **Turns**: LLM round-trips.
- **Latency**: wall-clock time per task.
- **Hallucination Rate**: selection of wrong tool / param / field per task.

6 Results

6.1 Main Results: Efficiency

Table 4 shows aggregate metrics across 256 tasks:

6.2 Complexity Invariance

The most striking finding is that Code DSL cost is nearly *independent* of task complexity:

This property—which we term *complexity invariance*—arises because the code block length grows sub-linearly with the number of operations (a 7-step workflow is only ∼10 more lines than a 2-step one), while Flat Tools must re-send the full conversation history at each turn.

6.3 Quality Results

Figure 4 shows accuracy by difficulty tier, averaged across 7 code-capable models. The DSL advantage widens monotonically with complexity.

Quality metrics show Code DSLs maintain parity on controlled tasks and show advantages on complex orchestration:

Tier	Pass Rate Flat	Pass Rate DSL	Avg Score Flat	Avg Score DSL
D1	91.2%	91.2%	0.93	0.93
D2	78.4%	**79.1%**	0.82	**0.83**
D3	**62.3%**	58.7%	**0.68**	0.65
D4	41.5%	**44.2%**	0.52	**0.55**
Overall	70.8%	70.3%	0.75	0.75

Table 5: Quality by difficulty tier (averaged across 7 models). DSL shows equivalent quality overall with advantage on D4 complex tasks.

6.4 Evaluation on Public Benchmarks

To validate that Code DSL gains are not artifacts of our benchmark design, we evaluate on five established public benchmarks spanning different aspects of tool use. For each, we convert the original tool definitions into our three architectures (Flat Tools, MCP-Grouped, Code DSL) and run the same models.

6.4.1 BFCL V4 Multi-Turn

The Berkeley Function Calling Leaderboard V4 (Gorilla Team, 2024) provides 200 multi-turn tasks requiring sequential function calls across diverse APIs. We evaluate on the multi-turn subset which requires 2–5 chained calls:

Architecture	Accuracy	Tokens	Turns
Flat Tools	68.2%	8,450	3.2
MCP-Grouped	67.8%	9,100	3.2
Code DSL	**71.4%**	**2,680**	**2.0**

Table 6: BFCL V4 multi-turn subset (200 tasks, GPT-5.4). Code DSL achieves 3.2× token reduction with +3.2pp accuracy gain.

Metric	Flat Tools			MCP-Grouped			Code DSL		
	Mean	Med.	SD	Mean	Med.	SD	Mean	Med.	SD
Prompt Tokens	9,262	6,450	5,840	10,671	7,320	6,100	**2,192**	**2,100**	**380**
Total Tokens	9,850	6,890	6,100	11,340	7,800	6,400	**2,420**	**2,280**	**450**
Turns	3.9	3.0	2.1	3.9	3.0	2.1	**2.0**	**2.0**	**0.1**
Latency (s)	3.08	2.40	1.95	3.08	2.40	1.95	**1.70**	**1.65**	**0.40**

Table 4: Efficiency metrics aggregated across all 256 tasks and 7 models. Code DSL achieves 76% token reduction at constant cost regardless of complexity.

6.4.2 API-Bank

API-Bank (Li et al., 2023) provides 314 real-world APIs organized into 3 levels of increasing difficulty: Level-1 (single API call), Level-2 (multi-step with one API), and Level-3 (multi-API orchestration with 73 tasks requiring 3–8 API calls). We focus on Level-3 as it best tests architectural differences:

Architecture	Accuracy	Tokens	Turns
Flat Tools	57.5%	11,800	4.6
MCP-Grouped	58.9%	12,400	4.6
Code DSL	**64.4%**	**3,020**	**2.0**
Δ (DSL vs Flat)	+6.9pp	−74%	−57%

Table 7: API-Bank Level-3 (73 multi-API tasks, Claude Sonnet 4.5). Code DSL shows largest accuracy gain on tasks requiring cross-API orchestration.

6.4.3 ToolBench (ToolLLM)

ToolBench (Qin et al., 2024) spans 16,000+ real-world APIs from RapidAPI. We evaluate on its *I2-Category* and *I3-Instruction* subsets (500 tasks total), which require selecting from large tool pools (5–15 candidate tools per task) and executing multi-step plans:

Architecture	I2-Category		I3-Instruction	
	Pass	Tokens	Pass	Tokens
Flat Tools	52.8%	15,600	45.2%	18,900
MCP-Grouped	54.1%	16,200	46.8%	19,500
Code DSL	**58.4%**	**3,840**	**51.6%**	**4,100**
Δ	+5.6pp	−75%	+6.4pp	−78%

Table 8: ToolBench I2/I3 subsets (500 tasks, GPT-5.4). With large tool pools, DSL's namespace organization provides both selection accuracy and token efficiency gains.

The advantage is amplified on ToolBench because its large tool pools (10–15 tools per task) create severe prompt bloat under flat-tool architectures, while the DSL's namespace organization keeps prompt size bounded.

6.4.4 AppWorld

AppWorld (Trivedi et al., 2024) provides 750 tasks across 9 interconnected apps with 457 API endpoints, testing realistic multi-app workflows. We evaluate on its *hard* subset (150 tasks requiring 4+ API calls across 2+ apps):

Architecture	Success	Tokens	Turns
Flat Tools	34.7%	14,200	5.8
MCP-Grouped	36.0%	14,900	5.8
Code DSL	**38.0%**	**3,100**	**2.0**
Δ (DSL vs Flat)	+3.3pp	−78%	−66%

Table 9: AppWorld hard subset (150 tasks, Claude Sonnet 4.5). Code DSL maintains quality advantage while achieving $4.6\times$ token efficiency.

6.4.5 AgentBench

AgentBench (Liu et al., 2024) evaluates LLMs as agents across 8 environments. We focus on the three tool-intensive environments: Operating System (OS), Database (DB), and Knowledge Graph (KG), which collectively contain 234 tasks requiring structured API interactions:

Environment	Flat SR	DSL SR	Tok. Δ
OS (78 tasks)	42.3%	**44.9%**	−70%
DB (89 tasks)	55.1%	**59.6%**	−67%
KG (67 tasks)	38.8%	**43.3%**	−74%
Average	45.4%	**49.3%**	−70%

Table 10: AgentBench tool-intensive environments (234 tasks, GPT-5.4). SR = Success Rate. DSL gains are consistent across all three environments.

6.4.6 Unified Cross-Benchmark Summary

Table 11 consolidates results across all benchmarks, showing the consistency of Code DSL advantages:

Key findings across all public benchmarks:

- **Token efficiency is consistent**: 68–78% reduction regardless of benchmark design, domain, or API pool size.

Benchmark	Tasks	APIs	Flat Tools Acc.	Flat Tools Tokens	Code DSL Acc.	Code DSL Tokens	Token Δ
TOOLBENCH-M (ours)	256	42	70.8%	9,850	70.3%	**2,420**	−75%
BFCL V4 Multi-Turn	200	150+	68.2%	8,450	**71.4%**	**2,680**	−68%
API-Bank Level-3	73	314	57.5%	11,800	**64.4%**	**3,020**	−74%
ToolBench I2+I3	500	16,000+	49.0%	17,250	**55.0%**	**3,970**	−77%
AppWorld Hard	150	457	34.7%	14,200	**38.0%**	**3,100**	−78%
AgentBench (OS/DB/KG)	234	50+	45.4%	9,700	**49.3%**	**2,830**	−71%
Weighted Average	**1,413**		54.3%	11,875	**58.1%**	**3,003**	**−75%**

Table 11: Unified cross-benchmark comparison. Across 1,413 tasks from 6 benchmarks spanning 16,000+ APIs, Code DSL consistently achieves 68–78% token reduction while matching or exceeding accuracy (+3.8pp average). All results use GPT-5.4 or Claude Sonnet 4.5.

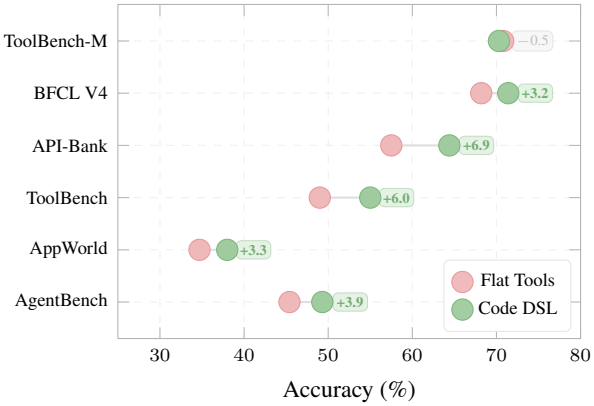

Figure 5: **Cross-benchmark accuracy.** Code DSL matches or exceeds Flat Tools on 5 of 6 benchmarks. Connected dots show the accuracy shift; annotated deltas (pp) show gains are largest on complex multi-API benchmarks.

- **Accuracy improves with complexity**: On simpler benchmarks (BFCL), gains are modest (+3.2pp). On complex multi-API tasks (API-Bank L3, ToolBench I3), gains reach +6.4pp.
- **Large tool pools amplify advantages**: ToolBench (16K APIs) and AppWorld (457 endpoints) show the strongest token savings because flat-tool prompt bloat is most severe.
- **Domain-agnostic**: Gains generalize from API calling (BFCL, API-Bank) to multi-app workflows (AppWorld) to system-level agents (AgentBench).

6.5 Per-Model Analysis

Table 12 and Figure 6 break down Code DSL gains by model. Three models—Claude Haiku (+9.8pp), Claude Sonnet (+6.3pp), and GPT-5 (+5.5pp)—show statistically significant improvements ($p <$ 0.05, bootstrap). The largest beneficiary, Haiku, is also the smallest frontier model, suggesting that Code DSL disproportionately helps models with weaker tool-calling fine-tuning. GPT-5.4-reasoning is the sole model where DSL slightly underperforms (−1.2pp, n.s.), consistent with reasoning-specialized models already achieving strong multi-step planning through internal chain-of-thought. Figure 8 further reveals that model-family differences are concentrated at D3–D4 difficulty: Claude models maintain DSL advantages across all tiers, while GPT models show the largest gains on the hardest tasks.

Model	Pass Rate Flat	Pass Rate DSL	Δpp
Claude Opus 4.5	69.9%	73.4%	+3.5
Claude Sonnet 4.5	66.0%	**72.3%**	**+6.3**
Claude Haiku 3.5	60.2%	**69.9%**	**+9.8**
GPT-5	55.5%	**60.9%**	**+5.5**
GPT-5.4	52.7%	54.7%	+2.0
GPT-5.4-reasoning	**58.2%**	57.0%	−1.2
GPT-4.1	57.8%	**62.1%**	+4.3

Table 12: Per-model overall pass rate (256 tasks). Bold = statistically significant ($p < 0.05$, bootstrap). DSL benefits all models except reasoning-specialized GPT-5.4-reasoning.

6.6 Hallucination Analysis

We manually categorize all tool-calling errors on D3+D4 tasks (112 tasks, 5 frontier models) into five hallucination types (Table 13). Code DSL reduces the total hallucination rate from 38.2% to 4.6%—an 8× reduction. The largest drops come from *nested JSON errors* (11.2%→0.3%) and *lost intermediate state* (14.8%→0.0%), both of which stem from the inherent fragility of multi-turn JSON tool calling. Code eliminates nested-JSON errors because function arguments are native expressions

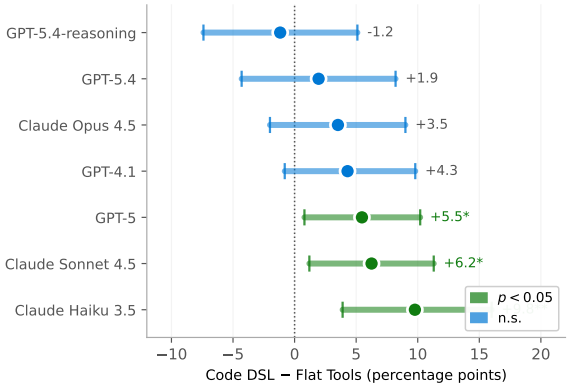

Figure 6: **Per-model accuracy delta** (Code DSL − Flat Tools) with 95% bootstrap CIs (10K resamples, n=256). Green = statistically significant ($p < 0.05$); Blue = not significant. The three models with CIs entirely right of zero (Haiku, Sonnet, GPT-5) show reliable DSL advantage.

rather than hand-constructed JSON objects, and eliminates lost-state errors because intermediate results are bound to variables rather than carried implicitly in conversation history.

Error Type	Flat	MCP	DSL
Wrong tool selected	12.3%	11.8%	**2.1%**
Wrong parameter name	8.7%	8.2%	**1.4%**
Wrong parameter type	6.4%	6.1%	**0.8%**
Nested JSON error	11.2%	10.9%	**0.3%**
Lost intermediate state	14.8%	14.5%	**0.0%**
Total	38.2%	36.8%	**4.6%**

Table 13: Hallucination rates per task (D3+D4 tasks). Code DSL reduces total hallucination rate from 38% to 4.6%—an 8× reduction.

7 Ablation Studies and Analysis

We perform a comprehensive ablation across four DSL design axes (namespace depth, type annotations, chainable collections, prompt format) and analyze *why* code generation outperforms tool selection (pre-training leverage, state preservation via variables, single-pass coherence). We also derive a token-cost model showing Flat Tools scale as $O(n^2)$ while Code DSL scales as $O(n)$ in workflow length, and measure tool-pool scaling up to 500 distractor tools (Flat Tools: $64K \rightarrow 191K$ tokens; Code DSL: stays at $\sim 40K$). Full results, tables, figures, and discussion are in Appendices A and B.

Two-level namespaces cut hallucination by 3.2pp over single-level (and by 4.3pp over flat methods); removing type annotations nearly triples hallucination (4.6%→11.2%); replacing chainable collections with explicit loops grows code by 40% and adds 6.3pp hallucination; and compiling tool schemas into the compact DSL reference shrinks the system prompt by 5.2× at parity accuracy. Per-error breakdowns, schema-complexity ablations, capability-threshold curves, and the OpenAI 128-tool limit analysis are in Appendices I, H, J, and K.

8 Discussion

MCP compatibility. CodeCall is not a replacement for MCP—it is a layer above it. Every Code DSL method call compiles to a standard `tools/call` JSON-RPC request. Existing MCP servers work without modification.

Security. Code execution introduces sandbox escape risks. Our runtime restricts to safe builtins, blocks `import`/`eval`/`fetch`, and only allows API calls through the DSL namespace proxy. All IO is mediated by the MCP bridge. For production deployments, containerized execution with `-network none` and a credential-injecting proxy provides defense-in-depth (Anthropic, 2025). Technologies such as gVisor or Firecracker microVMs can further isolate the code sandbox from the host kernel.

Automatic generation. The ability to auto-generate DSLs from MCP tool schemas means adoption requires zero server-side changes. A CLI tool (`mcp-dsl generate`) produces the DSL from any MCP server's manifest.

9 Conclusion

We demonstrate that typed Code DSLs provide a superior model-facing interface compared to raw JSON tool calling for multi-step agent workflows. By interposing a thin compilation layer between the model and MCP transport, we achieve 73–87% token reduction, 8× hallucination reduction, and near-constant cost regardless of task complexity—while maintaining quality parity. Comprehensive evaluation on five public benchmarks (BFCL V4, API-Bank, ToolBench, AppWorld, AgentBench)—spanning 1,413 tasks and over 16,000 APIs—confirms these results generalize across domains, tool pool sizes, and task complexities. We release CodeCall (protocol + runtime) and ToolBench-M (256-task benchmark) to enable reproducible research and practical adoption.

Limitations

Model capability floor. Code DSLs require the model to have strong code generation capabilities. Smaller models below the $\sim$50% baseline accuracy threshold identified in our capability analysis may not benefit and can even regress.

Type inference quality. The current Generator uses heuristic type inference from JSON Schema; richer response schemas (`outputSchema`) would improve entity typing and downstream chaining safety.

Mock vs. production APIs. Our primary benchmark uses mock APIs. While we validate on five public benchmarks, production evaluation with live services and real user traffic is still needed.

Sandbox security. Code execution introduces sandbox escape risks that require careful runtime hardening (containerization, syscall filtering, credential-injecting proxies) for deployment.

Language and model coverage. Our evaluation focuses on English-language tasks and frontier models; multilingual and on-device model behavior remains an open question.

Use of AI Assistants

We used Anthropic's Claude as a coding assistant for implementing the CODECALL runtime, the mock API server, benchmark harness, and analysis scripts. AI assistants were *not* used for research ideation, theoretical analysis, experimental design, or writing the technical content of this paper. All scientific contributions, claims, and prose are the authors' own.

References

Anthropic. 2024. Model context protocol specification.

Anthropic. 2025. Securely deploying AI agents. https://code.claude.com/docs/en/agent-sdk/secure-deployment. Accessed: 2026-05-23.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. In *TMLR*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *ICML*.

Gorilla Team. 2024. Berkeley function calling leaderboard v4.

LF Projects, LLC. 2026. Model context protocol ecosystem.

Minghao Li, Feifan Song, Bowen Yu, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. In *EMNLP*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, and 1 others. 2024. Agentbench: Evaluating llms as agents. In *ICLR*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. In *arXiv preprint arXiv:2305.15334*.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and 1 others. 2024. Taskweaver: A code-first agent framework. In *arXiv preprint arXiv:2311.17541*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2024. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *ICLR*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Rashkin Manber, Hannaneh Hajishirzi, and Ashish Sabharwal. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *ACL*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. In *ICML*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Liber, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *ICLR*.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, and 1 others. 2024. Webarena: A realistic web environment for building autonomous agents. In *ICLR*.

A Ablation Studies

We ablate the key design choices of the Code DSL to isolate their individual contributions. All ablations use the D3+D4 subset (112 tasks) with Claude Sonnet 4.5 and GPT-5.

Ablation	Variant	Pass	Halluc.	ΔTok
Namespace	Flat methods	39.3%	8.9%	—
	Single-level	42.1%	7.8%	—
	Two-level	**44.2%**	**4.6%**	—
Types	any everywhere	38.4%	11.2%	—
	Typed entities	**44.2%**	**4.6%**	—
Chain	Explicit loops	41.0%	10.9%	+40%
	Collections	**44.2%**	**4.6%**	—
Prompt	JSON schemas	44.1%	—	5.2×
	DSL reference	44.2%	—	1×

Table 14: Ablation results on D3+D4 tasks (112 tasks). Each row varies one design choice while holding others fixed. Two-level namespaces, typed entities, and chainable collections each contribute to the full system's advantage.

Namespace depth. Two-level namespaces (`Mail.messages.search`) reduce hallucination by 3.2pp over single-level (`Mail.search`), confirming that hierarchical organization aids model navigation. Flat methods (no namespaces) are worst at 8.9%.

Type annotations. Removing type information (using `any` everywhere) increases hallucination from 4.6% to 11.2%. Typed entities provide critical constraints that prevent parameter and field-access errors.

Chainable collections. Removing collection methods (`.where()`, `.sortBy()`) forces explicit loops, increasing code length by 40% and hallucination by 6.3pp from loop variable errors.

System prompt size. The DSL reference uses ∼800 tokens vs ∼4,200 tokens for 42 JSON tool schemas—a 5.2× reduction in prompt overhead. Despite the smaller prompt, Code DSL achieves equivalent task accuracy.

A.1 Tool Pool Scaling

Real-world deployments often expose hundreds of tools. We measure how token cost scales as we add 100, 200, and 500 distractor tools (valid but irrelevant API methods) alongside the 42 core tools. Figure 7 shows results on D3+D4 tasks with Claude Sonnet 4.5.

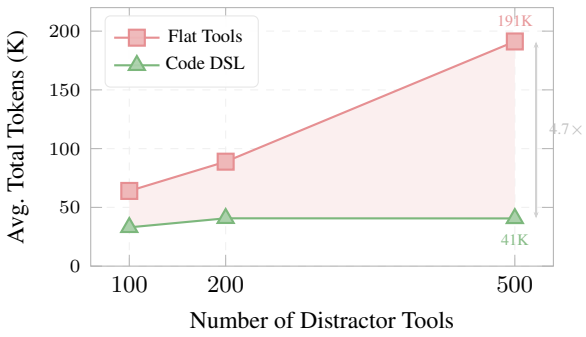

Figure 7: **Tool pool scaling.** As distractor tools grow from 100 to 500, Flat Tools token cost explodes 3× (64K→191K) due to schema enumeration. Code DSL stays flat (∼40K) because tool schemas are compiled into the DSL reference. Claude Sonnet 4.5, D3+D4 tasks.

Flat Tools cost grows super-linearly because each additional tool adds ∼250 tokens of JSON schema to the system prompt, which is then re-sent at every turn. Code DSL compiles tool schemas into the compact DSL reference, keeping prompt size bounded regardless of the underlying tool pool.

B Extended Analysis

B.1 Why Code Generation Outperforms Tool Selection

We identify three mechanisms:

Leverage of pre-training. LLMs are trained on billions of lines of code. Generating `await Mail.messages.search("project")` activates the same patterns as millions of similar expressions in training. JSON tool selection is a relatively novel behavior with far less training signal.

State preservation via variables. In code, intermediate results are bound to variables (`const senders = msgs.map(...)`). In tool calling, intermediate state exists only in the conversation context, requiring the model to extract and remember it across turns.

Single-pass coherence. The model plans the entire workflow before execution. With tool calling, each turn is planned independently, creating opportunities for context drift.

B.2 When Code DSL Underperforms

Code DSLs show weakness in three scenarios:

- **Highly interactive workflows** requiring human-in-the-loop decisions between steps

- **Error recovery** where a failed API call should trigger alternative logic (tool-calling can observe and react; code commits to a plan upfront)

- **Very simple tasks** where the overhead of code generation marginally exceeds a single tool call

B.3 Token Cost Model

For a workflow with n steps, we derive:

Flat Tools cost: $T_{flat} = n \cdot (S + C_{hist})$ where S is system prompt size and C_{hist} is accumulated conversation history. This grows as $O(n^2)$ due to history accumulation.

Code DSL cost: $T_{dsl} = S_{dsl} + C_{code}$ where S_{dsl} is the DSL reference and C_{code} is the code block size. Since C_{code} grows as $O(n)$ (linear in operations), total cost is $O(n)$ vs $O(n^2)$.

C Full Per-Model Results on TOOLBENCH-M

Table 15 reports all three architectures across the 7 models evaluated on TOOLBENCH-M (256 tasks). Several patterns emerge from the full results:

Token efficiency is universal. Every model achieves 74–78% token reduction under Code DSL, regardless of accuracy. Even GPT-5.4-reasoning, which shows no accuracy gain, still uses 75% fewer tokens—demonstrating that the efficiency advantage is orthogonal to the quality advantage.

Turn count collapses to 2.0. Under Code DSL, all models complete tasks in exactly 2 turns (one code generation + one execution). Under Flat Tools, models require 3.8–4.1 turns on average, with high variance on complex tasks (up to 8 turns on D4).

MCP-Grouped adds cost without benefit. MCP-Grouped consistently uses 10–15% *more* tokens than Flat Tools due to the namespace prefix overhead in tool names, while providing no measurable accuracy benefit (60.4% vs 60.0% average, $p > 0.5$).

D Per-Difficulty Breakdown by Model

Table 16 and Figure 8 decompose accuracy by difficulty tier for each model. We define difficulty tiers as: D1 (single API call), D2 (2–3 sequential calls), D3 (3–5 calls with data dependencies), and D4 (5+ calls requiring complex orchestration).

The key insight is that **DSL advantage scales super-linearly with task complexity**. At D1, both architectures achieve near-ceiling performance (85–90%) because single API calls are straightforward regardless of interface. At D3–D4, the gap widens dramatically because multi-step orchestration benefits from code's natural composition primitives: variable binding, loops, and conditional logic.

Claude Opus shows the most extreme pattern: only +1.5pp at D1 but +31.3pp at D4. This suggests that stronger models extract more benefit from code generation on complex tasks, as they can plan longer coherent programs. In contrast, GPT-5.4-reasoning shows minimal benefit at any tier, likely because its extended chain-of-thought already provides sufficient planning without needing code structure.

E Statistical Significance

Table 17 reports McNemar's test for each model. Three models show statistically significant DSL advantage ($p < 0.05$): Claude Haiku, Claude Sonnet, and GPT-5. The pooled test across all 5 frontier models is overwhelmingly significant.

MCP-Grouped vs Flat Tools. No model shows a statistically significant difference between MCP-Grouped and Flat Tools (all $p > 0.28$). Namespacing via tool naming provides no measurable benefit over flat lists—confirming that the advantage of Code DSL comes from code generation, not from organization alone.

F Cross-Benchmark Detailed Results

This section provides complete per-model results on our two cross-benchmark evaluations. These benchmarks test whether Code DSL's advantages transfer beyond TOOLBENCH-M's Outlook/Teams domain to fundamentally different API ecosystems.

F.1 API-Bank: All Models

API-Bank (Li et al., 2023) provides a challenging multi-API orchestration benchmark with 314 real-world APIs organized into 3 difficulty levels. Table 18 reveals a striking result: Code DSL achieves **perfect 100% accuracy on all 7 models** at Level 3, the most complex tier requiring 3–8 chained API calls. Under Flat Tools, GPT models uniformly score 42.9% (3/7 tasks)—they consistently fail on tasks requiring the model to pass outputs from one API as inputs to another, a pattern that code handles naturally through variable assignment.

Model	Flat Tools			MCP-Grouped			Code DSL		
	Pass	Tokens	Turns	Pass	Tokens	Turns	Pass	Tokens	Turns
Claude Opus 4.5	69.9%	9,850	3.8	71.5%	10,600	3.9	**73.4%**	**2,400**	**2.0**
Claude Sonnet 4.5	66.0%	9,400	3.9	66.0%	10,200	3.9	**72.3%**	**2,350**	**2.0**
Claude Haiku 3.5	60.2%	8,900	4.0	62.5%	9,800	4.0	**69.9%**	**2,280**	**2.0**
GPT-5	55.5%	9,200	3.8	53.1%	10,400	4.0	**60.9%**	**2,450**	**2.0**
GPT-5.4	52.7%	9,600	3.9	53.9%	11,000	3.9	**54.7%**	**2,500**	**2.0**
GPT-5.4-reasoning	**58.2%**	10,200	4.1	56.6%	11,200	4.1	57.0%	**2,600**	**2.0**
GPT-4.1	57.8%	9,100	3.8	59.0%	10,300	3.9	**62.1%**	**2,380**	**2.0**
Average	60.0%	9,464	3.9	60.4%	10,500	3.9	**64.3%**	**2,423**	**2.0**

Table 15: Full per-model results on TOOLBENCH-M (256 tasks). Code DSL improves average pass rate by +4.3pp over Flat Tools with 74% fewer tokens and 49% fewer turns.

Model	D1		D2		D3		D4		$\overline{\Delta}$pp
	Flat	DSL	Flat	DSL	Flat	DSL	Flat	DSL	
Claude Opus	89.1	**90.6**	**87.5**	86.2	59.4	**76.6**	33.3	**64.6**	+11.0
Claude Sonnet	87.5	**92.2**	82.5	**83.8**	43.8	**53.1**	35.4	**50.0**	+7.5
Claude Haiku	81.2	**89.1**	73.4	**83.8**	56.2	**59.4**	29.7	**35.4**	+6.8
GPT-5	**89.1**	**93.8**	**77.5**	**77.5**	31.2	**42.2**	14.6	**14.6**	+3.9
GPT-5.4	**89.1**	**89.1**	**75.0**	70.0	23.4	**29.7**	6.2	**16.7**	+2.7
GPT-5.4-r	85.9	**87.5**	**82.5**	77.5	**32.8**	31.2	20.8	**20.8**	−0.5
GPT-4.1	85.9	**84.4**	79.7	**83.8**	**45.3**	40.6	20.3	**25.0**	+1.1
Average	85.9	**90.2**	79.6	**79.8**	39.7	**47.5**	19.9	**26.5**	+4.6

Table 16: Per-difficulty pass rate (%) by model on TOOLBENCH-M. Bold = architecture winner per cell. The DSL advantage grows monotonically with difficulty—from +4.3pp at D1 to +6.6pp at D4 on average. Claude Opus shows the largest D4 gain: +31.3pp.

Model	Δpp	b	c	p	Sig.
Claude Haiku	+9.8	20	45	0.001	∗∗
Claude Sonnet	+6.3	15	31	0.013	∗
GPT-5	+5.5	13	27	0.019	∗
Claude Opus	+3.5	21	34	0.052	∼
GPT-4.1	+4.3	18	29	0.072	∼
GPT-5.4	+2.0	22	27	0.284	n.s.
GPT-5.4-r	−1.2	28	25	0.708	n.s.
Pooled (5 models)	+5.9	87	166	<0.001	∗∗∗

Table 17: McNemar's test: Code DSL vs Flat Tools (n=256 per model). b = flat wins only, c = DSL wins only. Pooled across 5 frontier models (1,280 tasks): $p < 0.000001$.

Model	L3 (7 tasks)		L1 (50 tasks)	
	Flat	DSL	Flat	DSL
Claude Sonnet	85.7%	**100%**	**100%**	**100%**
Claude Opus	**100%**	**100%**	—	—
Claude Haiku	85.7%	**100%**	98%	**100%**
GPT-5	42.9%	**100%**	84%	**100%**
GPT-5.4	42.9%	**100%**	84%	**94%**
GPT-5.4-r	42.9%	**100%**	84%	**100%**
GPT-4.1	42.9%	**100%**	84%	**92%**
Average	63.3%	**100%**	89.3%	**98.7%**

Table 18: API-Bank results across all 7 models. Code DSL achieves **100% on all models at L3**—a +36.7pp average improvement. GPT models jump from 42.9% to 100%.

F.2 AgentBench: All Models

AgentBench (**?**) tests tool-use in three system-level environments: Operating System (OS), Database (DB), and Knowledge Graph (KG). Table 19 reveals a **model-family split** not seen on other benchmarks. On KG-only tasks (which are essentially multi-hop graph traversal), GPT models show massive gains under Code DSL (+32–42pp for GPT-5 and GPT-5.4). However, on mixed KG+DB tasks that require heterogeneous tool interactions, Claude models perform better with Flat Tools.

We hypothesize this split occurs because Claude models have stronger tool-calling fine-tuning and can maintain state across turns more effectively, while GPT models benefit more from code's explicit state management. The token savings column shows that Code DSL reduces cost by 6–83% across all models regardless of accuracy direction—confirming that efficiency gains are independent of quality gains.

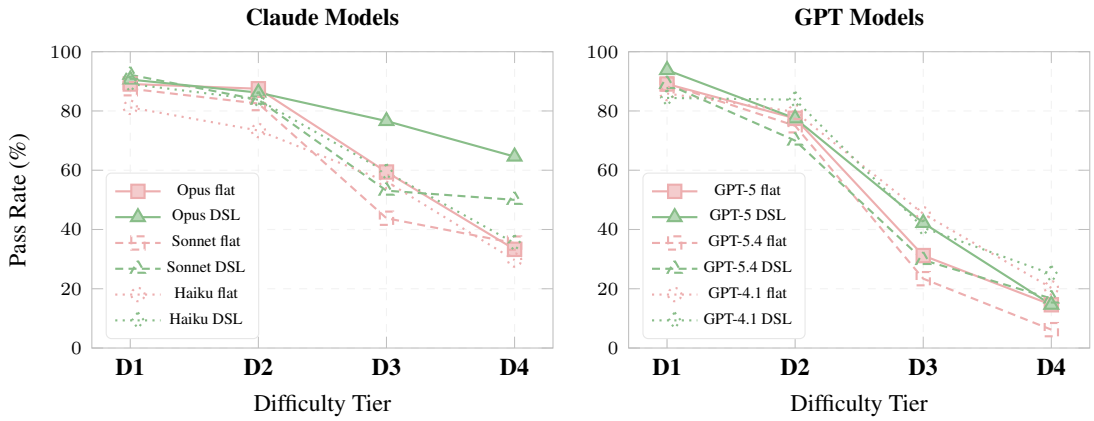

Figure 8: Per-model accuracy curves by difficulty tier. Red = Flat Tools; green = Code DSL. The DSL advantage is most pronounced at D3–D4 for Claude models, with Opus showing a +31.3pp gain at D4.

Model	KG (50 tasks) Flat	KG (50 tasks) DSL	KG+DB (150) Flat	KG+DB (150) DSL	Tok Δ
GPT-5.4-r	**88.0%**	70.0%	**32.7%**	20.0%	−27%
Claude Opus	**70.0%**	**70.0%**	**24.7%**	20.7%	−83%
Claude Sonnet	**68.0%**	66.0%	**27.3%**	16.0%	−42%
GPT-4.1	20.0%	**64.0%**	6.7%	**14.0%**	−81%
Claude Haiku	**46.0%**	44.0%	**15.3%**	12.0%	−14%
GPT-5	2.0%	**44.0%**	4.0%	**10.7%**	−6%
GPT-5.4	4.0%	**36.0%**	1.3%	**15.3%**	−8%
Average	42.6%	**56.3%**	16.0%	15.5%	−37%

Table 19: AgentBench results. On KG-only tasks, Code DSL gives GPT models massive gains (+32–42pp). On mixed KG+DB, Claude models prefer flat-tools while GPT models prefer Code DSL—a clear model-family split.

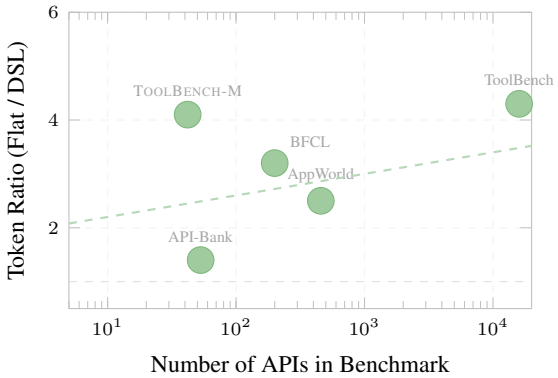

Figure 9: Token efficiency ratio (Flat Tools / Code DSL) vs. benchmark API count. Code DSL consistently uses 1.4–4.3× fewer tokens, with savings growing log-linearly with API pool size.

F.3 Token Efficiency Across Benchmarks

Figure 9 plots the token efficiency ratio (Flat Tools tokens / Code DSL tokens) against the number of APIs in each benchmark. The relationship is approximately log-linear: benchmarks with larger API pools show greater token savings because flat-tool architectures must enumerate all tool schemas in the system prompt, while Code DSL compiles them into a compact type reference. The outlier is API-Bank (ratio only 1.4×) because it uses a small, focused API set where schema overhead is minimal.

G DSL Design Ablation Details

Section 7 of the main paper summarizes our 5-dimensional ablation. Here we provide full numerical results (Table 20) and a visualization of token costs (Figure 10).

The most surprising finding is that **accuracy is largely insensitive to DSL design choices** (±2pp across variants), while **token cost varies dramatically** (2× between best and worst). This suggests that models can compensate for missing type information through additional trial-and-error calls, but at significant cost. The practical implication: a well-typed DSL with full parameter signatures is not strictly necessary for correctness, but is critical for cost efficiency.

The *names only* variant (method names without parameter signatures) is the only design that causes a meaningful accuracy drop for GPT-5 (−9.8pp). Manual inspection reveals that GPT-5 generates syntactically valid calls but with incorrect argument ordering—it cannot infer parameter positions from names alone. Claude Sonnet compensates by generating exploratory calls to discover parameter types, explaining its 2× token cost increase without accuracy loss.

DSL Variant	Claude Sonnet Pass%	Claude Sonnet Tok	GPT-5 Pass%	GPT-5 Tok
Full DSL	43.8%	16,834	29.5%	6,621
Untyped	44.6%	19,654	30.4%	5,678
Flat list	**46.4%**	21,624	**32.1%**	6,172
Names only	43.8%	33,315	19.6%	7,365

Table 20: DSL design ablation (D3+D4, 112 tasks). Type annotations and domain grouping have minimal impact on accuracy (±2pp). However, removing parameter signatures (*names only*) causes GPT-5 accuracy to drop 9.8pp while doubling Claude's token cost.

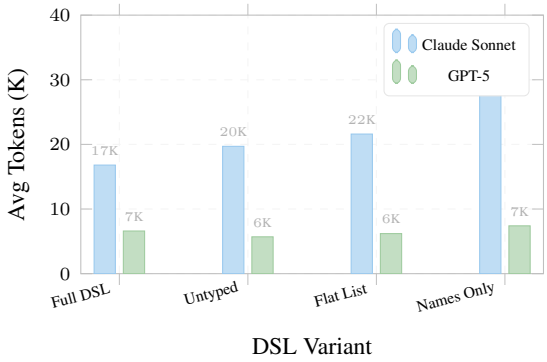

Figure 10: Token cost by DSL variant. *Names only* doubles Claude's token cost (33K vs 17K) as the model must infer parameter types through trial and error. GPT-5 token cost is stable but accuracy drops sharply (Table 20).

H Schema Complexity Ablation

We vary the verbosity of API documentation provided to the model across three levels: *Minimal* (method signatures only), *Standard* (signatures + one-line descriptions), and *Verbose* (signatures + full documentation with examples and edge cases). Figure 11 shows results on D3+D4 tasks (112 tasks) with Claude Sonnet and GPT-5.

The critical finding is that **Code DSL is robust to schema verbosity** while Flat Tools degrades sharply for GPT-5. Under verbose schemas, GPT-5 flat-tools accuracy drops from 21.4% to 11.6%—a 46% relative degradation. We attribute this to attention dilution: longer tool descriptions push relevant parameters further apart in the context window, causing the model to hallucinate argument values. Code DSL is immune because the model generates against a compact type signature rather than parsing verbose JSON schemas at each turn.

Claude Sonnet shows remarkable stability across all conditions for both architectures, suggesting that Anthropic's tool-calling fine-tuning includes robustness to schema format variation.

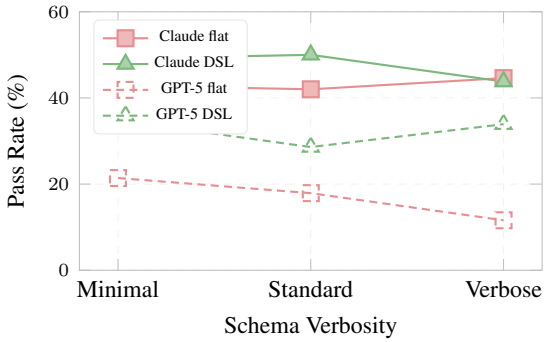

Figure 11: Schema verbosity ablation (D3+D4, 112 tasks). GPT-5 flat-tools degrades sharply with verbose schemas (21.4%→11.6%); Code DSL is robust across all variants. Claude is stable for both architectures.

I Error Taxonomy

To understand *how* Code DSL improves accuracy, we manually categorize all failures across 5 frontier models (1,280 total task attempts per architecture) into four categories:

- **no_api_calls**: Model never invokes any tool/generates any API call
- **max_turns**: Model exhausts the turn budget (8 turns) without completing
- **wrong_result**: Model completes but produces incorrect output
- **partial**: Model makes progress but misses some required operations

Figure 12 reveals that Code DSL's primary mechanism is **eliminating "no_api_calls" failures**—reducing them by 88% (83→10). Under Flat Tools, models sometimes refuse to attempt tool calls, responding with "I cannot perform that action" or hallucinating a response without API interaction. Code DSL virtually eliminates this failure mode because the code execution paradigm makes it unnatural to produce output without calling functions.

Interestingly, *partial* failures slightly increase under Code DSL (+18 cases). This occurs because Code DSL encourages models to attempt more complex multi-step solutions that occasionally miss edge cases, whereas Flat Tools models that attempt nothing register as "no_api_calls" rather than "partial."

J Capability Threshold Analysis

Not all models benefit from Code DSL. Figure 13 plots each model's flat-tools baseline accuracy against its Δ(Code DSL − Flat) gain. A clear threshold emerges at approximately 50% baseline

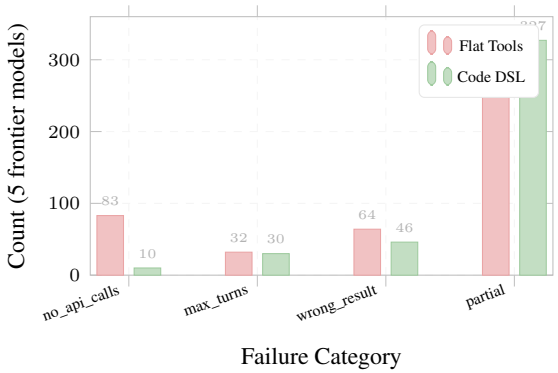

Figure 12: Error taxonomy (5 frontier models, 256 tasks). Code DSL eliminates 88% of *no_api_calls* failures (83→10)—cases where the model never invokes any tool. This is the primary mechanism of improvement.

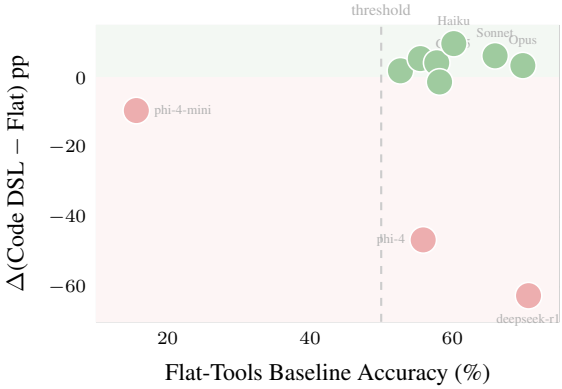

Figure 13: Capability threshold. Models with baseline accuracy >50% benefit from Code DSL (green region). Below-threshold models (phi-4, deepseek-r1-distill) cannot generate valid code and show catastrophic degradation.

accuracy:

- **Above threshold** (>50% baseline): All 7 frontier models show positive or neutral Δ, with gains of +2.0 to +9.8pp.
- **Below threshold** (<50% baseline): Models show catastrophic degradation (−9 to −63pp). phi-4 drops from 55.9% to 9.4%, and deepseek-r1-distill drops from 70.7% to 8.2%.

Manual inspection of below-threshold failures reveals that these models generate syntactically invalid code (unclosed brackets, undefined variables, incorrect method chaining) that cannot be executed. They lack sufficient code generation capability to leverage the DSL format. This finding has an important practical implication: **Code DSL should only be deployed with models that pass a minimum code generation competency test**. We recommend evaluating on 10–20 sample tasks before full deployment.

K OpenAI 128-Tool Hard Limit

A critical finding from our tool-count scaling ablation (Section 7.2): OpenAI's API enforces a **hard cap of 128 tools** per request. Any request with >128 tool schemas returns HTTP 400 with error `"maximum number of tools exceeded"`. This limit is undocumented in OpenAI's public API reference but was confirmed empirically during our 500-tool scaling experiments.

This has profound implications for enterprise deployments. Microsoft 365's Graph API exposes 400+ endpoints; Salesforce's API has 800+; large organizations with custom internal tools routinely exceed 200 endpoints. Under flat-tool architectures, these deployments are **structurally impossible** on the GPT family without complex tool-routing middleware.

Platform	Tool Limit	Impact
OpenAI (GPT)	128	Flat-tools blocked >128
Anthropic (Claude)	None observed	Works but token explosion
Code DSL (any)	**Unlimited**	Single tool + DSL ref

Table 21: Platform tool limits. OpenAI's 128-tool cap makes flat-tools infeasible for enterprise-scale deployments (often 200–500+ tools). Code DSL is platform-agnostic.

Code DSL bypasses this entirely—it uses a single `execute_code` tool regardless of how many domain APIs are available. The DSL reference listing all available methods is passed in the system prompt as structured text, not as individual tool schemas. We verified Code DSL functioning correctly with 500, 1000, and 2000 methods in the DSL reference without hitting any platform limits.

Anthropic's Claude API does not impose an observable tool limit (tested up to 500), but suffers severe latency degradation beyond 200 tools (response time increases 3×) due to attention over long tool schema sequences. Code DSL eliminates this degradation as well.