PROTOCAL: Natural Language to Executable Code for LLM Tool-Calling

Anonymous ACL submission

Abstract

In this paper, we introduce PROTOCAL (PROgram-mediated TOol CALling), a framework that improves LLM tool calling by generating and executing code scripts. At its core, PROTOCAL uses an LLM-based program generator to synthesize robust Python programs that can validate and repair tool-use errors (e.g., invalid function calls or malformed arguments) and decompose complex user instructions into modular sub-tasks when needed. We evaluate PROTOCAL with three program generators: two off-the-shelf models (8B and 120B parameters) and one supervised fine-tuned 8B model. Across seven executor LLMs with diverse architectures and scales, executing the synthesized programs consistently improves tool-calling accuracy, with average gains of 3.94% and up to 12%. We further analyze which executor failures the synthesized programs repair and through what mechanisms, and isolate the design choices that drive these gains.

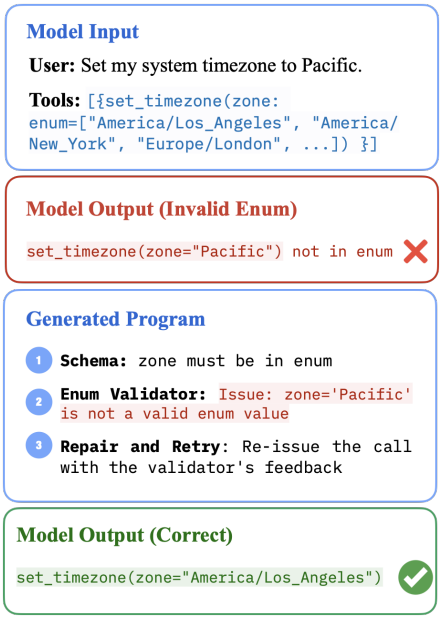

Figure 1.1: An example of a generated program detecting and repairing schema violations during tool calling.

1 Introduction

Tool-calling has become a core capability in agent systems, retrieval pipelines, and a wide range of AI assistants (Schick et al., 2023; Qin et al., 2024; Yao et al., 2022; Qu et al., 2025), and its performance strongly influences the reliability of the overall systems built on top of it. Yet even strong instruction-tuned models still struggle on standard benchmarks such as the Berkeley Function Calling Leaderboard (BFCL) (Patil et al., 2025). Common failure modes include hallucinated function names (Zhang et al., 2024), malformed arguments (Huang et al., 2024), missing parallel calls (Patil et al., 2025), and confident calls on prompts the user never asked the model to act on. The LLM often *knows* which tool is appropriate but commits to a wrong specific value, or commits when it should have abstained.

A growing line of work contains this kind of brittleness by re-casting the LLM as one component inside a *program* that the model itself helps write (Chen et al., 2022; Gao et al., 2023; Madaan et al., 2023; Yao et al., 2022; Shinn et al., 2023; IBM, 2025). The key intuition is that LLM stochasticity is best handled *outside* the model: a program can validate a structured output, retry on a typed failure signal, decompose a compound task, or preserve a deliberate refusal, capabilities that a single inference call cannot reliably do on its own.

What such a program should look like *specifically for tool-calling* is, however, an open question. Existing scaffolds bundle many design choices, such as when to decompose, when to retry, what counts as a refusal, whether to override the executor's output, and the literature reports aggregate gains without isolating which choices carry the load (Qin et al., 2024; Yao et al., 2022; Shinn et al., 2023; Madaan et al., 2023) (See comprehensive related work in Appendix A). Practitioners building on these systems are left with no clear guidance on

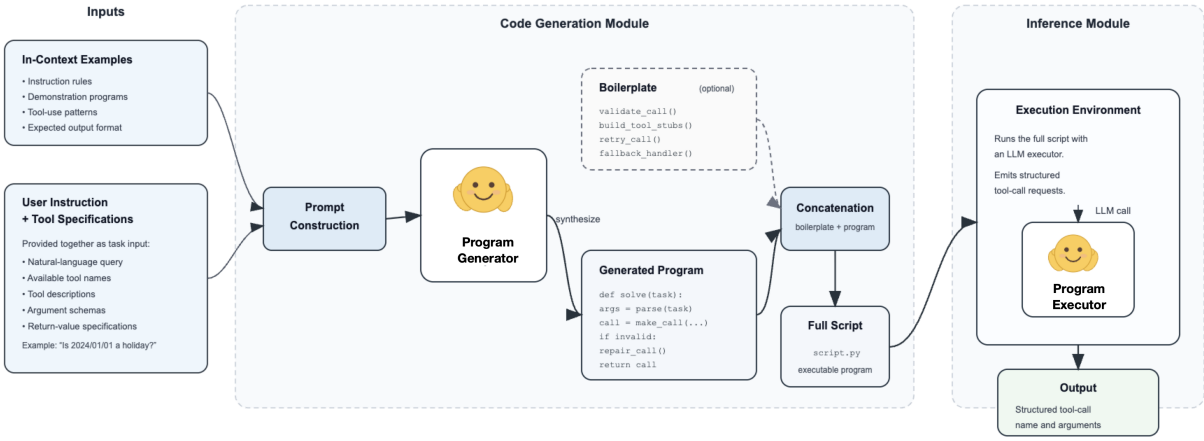

Figure 2.1: The PROTOCAL pipeline. A program generator turns each prompt into an executable script that wraps the executor LLM with validation, retry, and decomposition.

what the program is doing, which executor failures it actually fixes, or where it makes things worse.

This paper addresses this question directly and introduces PROTOCAL (PROgram-mediated TOol CALling), a framework that synthesizes a Python program per user prompt. The program can validate and repair tool-use errors (as shown in Figure 1.1), decompose complex user instructions into modular sub-tasks when needed, and is executed by an LLM. We use the resulting artifacts to study three research questions: (i) Does program-mediated scaffolding improve tool-calling over direct prompting, and how does the gain scale with executor size? (§3.2) (ii) Why does it help: which executor failures are repaired by the program, and through what mechanisms? (§4.1–4.4) (iii) What is required for it to help: which program design choices are necessary to obtain gains? (§4.4 and §4.5).

We evaluate PROTOCAL on two public tool-calling benchmarks using three LLM-based program generator (two off-the-shelf, one supervised fine-tuned) across seven executors spanning four model families. PROTOCAL consistently improves tool-calling performance, yielding gains of up to 12%. We further show that the dominant repair mechanism varies across executors, reflecting their specific failure patterns rather than a universal fix. Finally, a compact SFT-distilled program generator nearly matches a teacher model more than an order of magnitude larger, making PROTOCAL especially effective at small scale. Our contributions are threefold:

- **Framework**: We introduce PROTOCAL, which generates a per-prompt Python program to scaffold tool-calling and runs it against an executor LLM, supporting both off-the-shelf and supervised fine-tuned program generators.
- **Dataset and program generator**: We publicly release the generated programs and our fine-tuned program generator model for reproducible research and practical adoption of program-mediated tool-calling across new benchmarks, executors, and downstream tool-calling applications [1].
- **Empirical study**: We systematically treat scaffold design decisions as testable hypotheses, identifying which mechanisms drive performance gains and which assumptions do not hold empirically. Our analysis further shows a compounding effect of scale: lightweight program generators can substantially improve small executors, narrowing the gap to much larger models.

2 The PROTOCAL Framework

We introduce PROTOCAL (PROgram-mediated TOol CALling), a framework that scaffolds tool-calling with synthesized Python programs. Rather than asking an LLM to emit a tool call directly, PROTOCAL asks a *program generator* LLM to write a Python program that wraps a second *executor* LLM.

2.1 Overview

PROTOCAL consists of a *Code Generation Module* and an *Inference Module* (Figure 2.1). Standard tool-calling inputs (a natural-language user

[1] Model and dataset available at: [REDACTED]

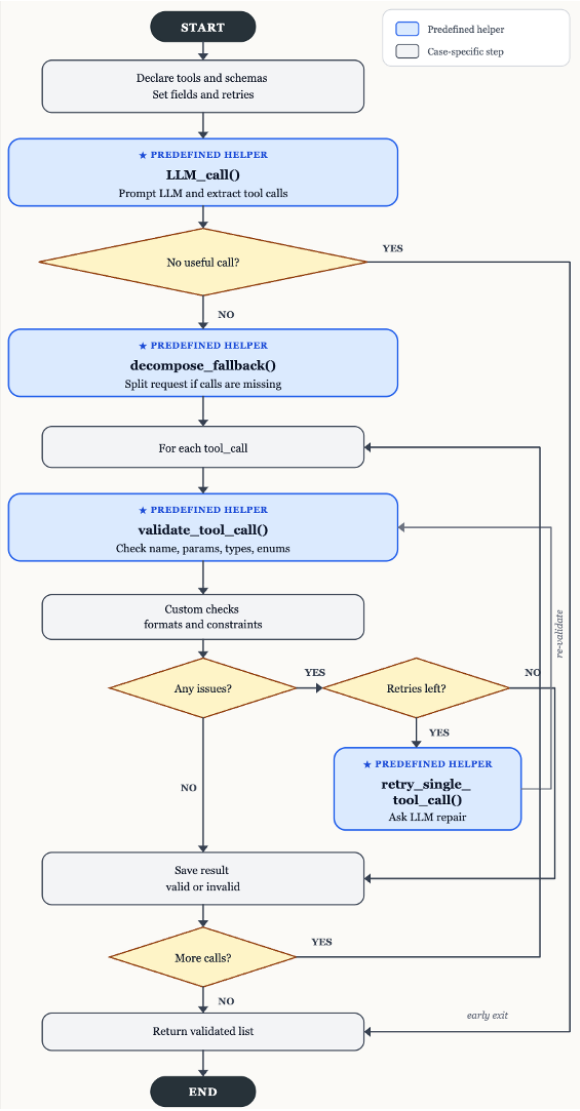

Figure 2.2: Tool-calling workflow in inference.

prompt and a JSON-formatted list of available tools) are concatenated with predefined coding rules and a small set of in-context learning (ICL) examples (see Appendix B). The Code Generation Module passes this input to the program generator, which synthesizes an executable Python script; reusable boilerplate may optionally be prepended. The Inference Module then runs the script in a controlled runtime that supports standard Python execution and LLM API calls, producing the final structured tool-call requests. The scope of PROTOCAL is single-turn tool-calling and the structural correctness of the issued calls.

2.2 Code Generation Module

The program generator receives the constructed prompt and generates a Python script that orchestrates calls to the executor, validates the returned tool calls, and applies repairs when validation fails. It writes the overall control flow, such as when to call the executor, how to iterate over returned candidates, how to branch on validation outcomes, and the prompt-specific custom validators discussed in §2.3. It does not redefine the recurring routines below; these are provided as predefined helpers that the generated program invokes by name:

- `LLM_call()` – prompts the executor and extracts a list of tool-call candidates.
- `decompose_fallback()`: re-calls each subtask individually only when the model underproduces tool calls
- `validate_tool_call()`: checks a candidate against the declared schema (function name, parameter set, types, enum values).
- `retry_single_tool_call()`: retries only the failing tool call with typed error feedback

2.3 Tool-calling Workflow in Inference

Figure 2.2 shows the control flow of a synthesized program. The program starts by setting a retry budget and issuing an initial `LLM_call()`. If the executor underproduces tool calls, it triggers `decompose_fallback()` which breaks the request into smaller sub-requests and reissues them individually.

Each returned call then passes through a two-stage validation process. The first stage, `validate_tool_call()`, is schema-level and uniform across all prompts, checking basic constraints such as function name, parameter presence, types, and enum membership. The second stage consists of *custom validators* generated for the specific prompt. These capture constraints that cannot be expressed in the schema alone e.g., date/time formats (`YYYY-MM-DD` versus Unix timestamps), location encodings (two-letter state codes versus full names, `city, state` pairs), unit conventions (Celsius/Fahrenheit, miles/kilometers), currency codes, and value ranges implied by the user's phrasing. These checks are inherently prompt-specific and therefore cannot be pre-defined as helper functions.

When either validation stage fails and retries remain, `retry_single_tool_call()` is invoked to repair the offending call using the specific validation error as context, after which the call is re-validated. Once all calls pass validation, the program returns the final list.

Table 3.1: Accuracy (%) of seven executors under Direct and PROTOCAL with three program generators, on both BFCL v3 and CCTU. Δ is the gain/loss of the best program over Direct. Bold/underline mark 1st/2nd best in each row.

Dataset	Executor	Direct	PROTOCAL			
			GPT-OSS-120B	Granite 8B	Granite 8B (SFT)	Δ
BFCL v3	Granite 4.1 3B	82.32%	**83.18%**	78.65%	82.77%	+0.86
	Granite 4.1 8B	84.70%	**84.87%**	80.90%	84.58%	+0.17
	Qwen 2.5 3B	73.40%	**77.25%**	72.30%	76.51%	+3.85
	Qwen 2.5 7B	77.81%	**81.37%**	75.73%	79.70%	+3.56
	GPT-OSS 20B	69.06%	**79.28%**	73.57%	78.25%	+10.22
	GPT-OSS 120B	68.94%	**77.72%**	72.05%	76.59%	+8.78
	Claude Haiku 4.5	83.75%	**83.99%**	80.31%	83.62%	+0.24
CCTU	Granite 4.1 3B	24.50%	33.50%	31.00%	**34.50%**	+10.00
	Granite 4.1 8B	37.50%	**39.00%**	34.00%	38.00%	+1.50
	Qwen 2.5 3B	31.00%	**34.00%**	25.00%	33.50%	+3.00
	Qwen 2.5 7B	**40.00%**	37.50%	31.50%	39.00%	-1.00
	GPT-OSS 20B	22.00%	**29.00%**	20.50%	27.00%	+7.00
	GPT-OSS 120B	22.00%	**34.00%**	21.00%	**34.00%**	+12.00
	Claude Haiku 4.5	**42.00%**	37.00%	33.50%	36.50%	-5.00

2.4 Program Generator Variants

We instantiate PROTOCAL with three program generators: two off-the-shelf instruction-tuned models used zero-shot with ICL examples, and one supervised fine-tuned (SFT) generator trained on program-correct pairs distilled from a stronger teacher. All three target the same boilerplate interface, so the Inference Module is unchanged across variants. The empirical study (§3.2) shows that the SFT generator paired with a small executor matches the performance of substantially larger counterparts.

3 Experiments

This section evaluates PROTOCAL against direct LLM inference and reports the per-executor gains, scaling behavior, and how smaller program generators compare to larger ones.

3.1 Experimental Setup

Benchmarks. We evaluate on two tool-calling benchmarks: **BFCL v3** (Patil et al., 2025) and **CCTU** (Ye et al., 2026). **BFCL v3** includes single- and multi-call settings (simple, multiple, parallel, and parallel-multiple) as well as relevance and irrelevance detection, and we report the official AST accuracy. **CCTU** evaluates tool use under complex constraints not covered by BFCL v3, making it a more challenging out-of-distribution setting. A prediction is correct if the function name and returned count match the `unsolved_set`. All results are computed over fixed evaluation sets: BFCL v3 single-turn inputs ($n = 4079$; excluding SQL and REST subsets) and CCTU ($n = 200$), and we evaluate only the tool-structure component of each sample, excluding downstream assistant content construction.

Models. We evaluate seven executors spanning open- and closed-weight families and scales: Granite 4.1 3B and 8B (IBM Research, 2026), Qwen 2.5 3B and 7B (Yang et al., 2025), GPT-OSS-20B and 120B (Agarwal et al., 2025), and Claude Haiku 4.5 (Anthropic, 2025).[2] For program generators, we use two off-the-shelf models (Granite 8B and GPT-OSS-120B) and one SFT variant of Granite 8B (recipe in Appendix C). Decoding is greedy with a fixed maximum length, and all models share the same tool schema and system prompt within a condition.

Conditions. For each of the seven executors we compare two conditions on the same inputs: (i) **Direct**, in which the executor performs function calling directly from the user prompt and (ii) **PROTOCAL** in which the program generator first maps the prompt into a program that the executor then runs. Comparing the two conditions on identical executors isolates the contribution of PROTOCALfrom raw model capability. A worked example in Figure E.1 shows a representative case from BFCL v3.

SFT data. We construct SFT data mixture by self-distillation through PROTOCAL. We sample prompts from xLAM (Liu et al., 2024), Toucan (Xu et al., 2025) and Granite SFT splits (IBM Research, 2025), and use the *Code Generation Module* with GPT-OSS-120B as the program generator to produce candidate prompt-program pairs. Each pro-

[2]We omit Llama-Instruct executors (3.2 3B, 3.1 8B, 3.3 70B), as they emit multi-call outputs as JSON in the assistant content rather than as structured tool calls, which standard parsers do not surface.

Table 3.2: Error category transitions under Direct prompting vs. Program execution. Green cells = fewer errors under PROTOCAL; red cells = more errors under PROTOCAL. Counts are sample-level errors. PROTOCALcounts are the mean over three adapters, rounded to the nearest integer.

(a) BFCL v3 top-10 error categories ($n = 4079$)

	Granite 4.1 3B		Granite 4.1 8B		Qwen 2.5 3B		Qwen 2.5 7B		GPT-OSS-20B		GPT-OSS-120B	
Error Category	**Direct**	**P**	**Direct**	**P**	**Direct**	**P**	**Direct**	**P**	**Direct**	**P**	**Direct**	**P**
runtime_error	4	5	0	4	0	2	0	2	3	24	0	10
irrelevance:called_tool	173	128	166	140	364	231	357	225	194	162	179	156
value_error:string	166	164	165	159	198	174	179	168	164	152	180	174
parallel_function_checker:wrong_count	15	8	20	18	56	59	26	15	440	68	440	94
multiple_function_checker:wrong_count	75	94	42	61	37	66	52	67	88	111	71	93
exec:mismatch	56	58	58	59	81	79	61	60	127	76	135	91
value_error:others	58	57	56	57	67	65	78	76	76	76	70	76
parallel_function_checker:cannot_find_match	40	42	29	32	58	57	39	43	0	49	0	54
simple_function_checker:wrong_count	38	39	10	13	22	22	14	18	48	43	68	70
simple_function_checker:missing_optional	33	31	22	21	82	74	12	9	18	19	13	17
wrong:type	13	16	18	19	31	30	29	26	32	24	42	33

(b) CCTU error transitions ($n = 200$)

	Granite 4.1 3B		Granite 4.1 8B		Qwen 2.5 3B		Qwen 2.5 7B		GPT-OSS-20B		GPT-OSS-120B	
Error Category	**Direct**	**P**	**Direct**	**P**	**Direct**	**P**	**Direct**	**P**	**Direct**	**P**	**Direct**	**P**
runtime_error	0	0	0	1	0	0	0	0	0	2	0	0
multiple_function_checker:wrong_count	100	100	100	100	92	96	98	95	100	99	100	100
parallel_function_checker:wrong_count	40	9	13	3	20	14	2	5	50	18	50	13
parallel_function_checker:cannot_find_match	4	17	10	16	10	16	14	19	0	10	0	11
simple_function_checker:wrong_count	1	4	0	2	6	2	0	2	1	11	1	6
simple_function_checker:wrong_func_name	6	3	2	4	6	5	6	3	5	5	5	2

gram is then executed using two strong executors, GPT-OSS-120B and Claude Haiku 4.5, and we retain only those pairs whose execution outputs match the ground-truth tool calls. Using the resulting filtered data, we fine-tune Granite 4.1 8B under two mixture settings: GPT-Correct and GPT-Haiku-Correct, yielding two supervised program generation models. The pool of source prompts is summarized in Table C.1. We report results for GPT-Correct in Table 3.1; full results for both SFT-based program generator are provided in Table F.1 and F.2.

3.2 Results

Table 3.1 reports BFCL v3 and CCTU results across seven executors under **Direct** and **PROTOCAL** conditions. For PROTOCAL we report results with three program generators: two off-the-shelf (Granite 8B, GPT-OSS-120B) and one fine-tuned on Granite 4.1 8B. The Δ column shows the best program's improvement over that executor's Direct baseline.

Main finding. Averaged across the seven executors, PROTOCAL improves BFCL v3 AST accuracy from 77.14% (Direct) to 81.09% (best Program), an absolute gain of **+3.95%**, with every executor improving under at least one program generator; per-executor gains range from +0.17% on Granite 4.1 8B to +10.22% on GPT-OSS-20B. CCTU shows the same direction on average (+3.93%), but two executors regress: Qwen 2.5 7B (-1.00%) and Claude Haiku 4.5 (-5.00%). We attribute these CCTU regressions to insufficient multi-hop coverage in the program generator and its SFT mixture.

Effect of executor scale. The gain from program is not a function of executor size. On BFCL v3, larger executors tend to benefit less from the program within each model family, with an average reduction of the gain of $\sim$0.81% on BFCL v3. A more consistent predictor than model size is the executor's Direct accuracy. Executors with lower Direct accuracy benefit more from program mediation: the GPT-OSS models, which start at $\sim$69% Direct accuracy, achieve the largest gains, whereas Granite models, with higher Direct accuracy (82-85%), exhibit the smallest improvements. On CCTU the within-family pattern is mixed: Granite and Qwen follow the same direction, but GPT-OSS reverses. Scaffolding is most effective when the executor leaves room for correction, but the relationship is benchmark-dependent rather than determined by scale alone.

Off-the-shelf vs. fine-tuned program generator. The off-the-shelf GPT-OSS-120B generator is the strongest on BFCL v3, leading on all seven executors with the SFT Granite 8B generator trailing by only 0.29-1.67%. On CCTU the two are effectively interchangeable: GPT-OSS-120B leads on four executors, the SFT generator leads on Granite 4.1 3B and Qwen 2.5 7B (by 1.0% and 1.5%), and they tie on GPT-OSS-120B. The SFT recipe therefore successfully distills GPT-OSS-120B-quality programs

into a Granite 8B-sized student, closing most of the gap to the much larger teacher.

Error category transitions across benchmarks. Table 3.2 groups the top-10 error categories into three patterns. (i) PROTOCAL sharply cuts `parallel_function_checker:wrong_count` on every executor with a non-trivial Direct baseline, with the largest reductions on GPT-OSS (∼80% on BFCL, ∼70% on CCTU) and Granite on CCTU (∼75%); it only regresses where Direct is already near-zero. (ii) Part of that reduction is a relabeling: the related `cannot_find_match` category (right count, wrong names) gains errors on every CCTU executor and on GPT-OSS in BFCL. The net is still strongly positive – on GPT-OSS, ∼320 wrong-count fixes against ∼50 new name errors – but smaller than the wrong-count drop alone suggests. (iii) Single-call prompts are not where PROTOCAL helps: `simple_function_checker:wrong_count` regresses on five of six CCTU executors. The benchmarks also diverge on `multiple_function_checker:wrong_count`, which regresses 25–78% on every BFCL executor but stays flat on CCTU; CCTU's Multi-Hop classes have ground-truth counts of 6–11 that all executors miss under both conditions, saturating the cell and hiding the failure mode BFCL exposes. The same saturation explains why `runtime_error` is essentially zero on CCTU (only GPT-OSS-20B 0→2) but non-trivial on BFCL. Moreover, for BFCL v3[3], PROTOCAL reduces errors on every executor for `irrelevance:called_tool` (calling a tool when none was warranted; 15-37%) and `value_error:string` (a string argument in a non-canonical form; 1-12%). We unpack which scaffold mechanism drives each direction in §4.

4 Ablation Studies and Discussion

This section unpacks the headline gain in Table 3.1 by asking why does it help: which executor failures are repaired by the program, and through what mechanisms and what is required for it to help: which program design choices are necessary to obtain gains?

To understand why programs fixes and how (and when they hurt), we analyze three cases: *prog-win* (Direct wrong, PROTOCAL correct), *direct-win* (Direct correct, PROTOCAL wrong), and *both-fail*. To attribute each program win or loss to a specific mechanism, we use the program's runtime trace: `llm_call` (single-shot), `llm_retry` (the executor was re-prompted with the typed `issues` list), or `no_llm_call` (the program short-circuited the call). We combine these traces with a fine-grained comparison between Direct and Program outputs over function names, argument keys, argument types, and values. The complete repair-mechanism taxonomy is provided in Appendix G.

4.1 What the program fixes, and how

Table 4.1 breaks down the repair mechanisms for *prog-win* cases across executors (849 total cases), showing both what kinds of errors programs fix and how those fixes are achieved.

The dominant repair mechanism depends on the executor's native failure tendencies rather than the scaffold itself. For three of the four executors (Granite 3B and 8B, Qwen 7B), most gains come from correcting over-eager tool use: when the prompt does not require any tool, the executor often hallucinates a call, while the generated program correctly infers that no tool applies and returns the empty list (`force_empty`: 47.4-74.9%). This mechanism therefore contributes most on executors that struggle to abstain when no tool is appropriate. GPT-OSS-20B shows a different pattern: the executor more often under-decomposes parallel-multiple prompts and the validator detects the incomplete call set and triggers a per-subtask retry that generates the missing calls (`decompose+retry`: 71.0%).

The remaining wins come from repairing schema and argument-level mistakes that are individually small but collectively account for most of the long tail. Some of these errors can be corrected directly by the validator without another LLM call, such as canonicalizing enum or format values, dropping unsupported arguments, or coercing types (`validate+enum/format snap`, `validate+drop_unknown_arg`, and related `validate+*` variants). These lightweight repairs account for roughly 7-13% of wins per executor at no additional generation cost. Other cases require retrying generation after validator feedback, especially when the executor chooses the invalid function, omits required arguments, or produces incorrect values (`retry+enum/format snap`, `retry+rename_function`, `retry+add_arg`, `retry+value_rewrite`). Although each individual mechanism is relatively uncommon, together they explain most of the remaining gains outside

[3] Argument-level categories doesn't applied to CCTU because `unsolved_set` carries only function-name ground truth.

Table 4.1: Per-executor distribution of repair mechanisms among prog-wins (Direct ×, Program ✓). Each column is normalized by that executor's prog-win total. Mechanisms with $< 3\%$ share in every executor are folded into *other*; the full breakdown is in Table F.5 (Appendix).

Mechanism	Gran. 3B	Gran. 8B	Qwen 7B	GPT-OSS-20B
`decompose+retry`	6.0%	3.5%	4.1%	**71.0%**
`force_empty`	**55.4%**	**47.4%**	**74.9%**	9.7%
`force_tool_call`	1.2%	0.0%	2.6%	5.1%
`retry+enum/format snap`	12.0%	31.6%	4.6%	1.2%
`validate+enum/format snap`	4.8%	1.8%	2.6%	2.1%
`retry+type_coerce`	0.0%	3.5%	0.0%	1.0%
`retry+rename_function`	1.2%	0.0%	1.0%	2.7%
`retry+add_arg`	6.0%	1.8%	0.5%	0.4%
`validate+drop_unknown_arg`	1.2%	1.8%	4.6%	2.1%
`retry+value_rewrite`	4.8%	5.3%	2.6%	0.4%
other ($< 3\%$ everywhere)	7.2%	3.5%	2.6%	4.3%
Total prog-wins (n)	83	57	195	514

the dominant patterns.

4.2 What the program cannot fix, and why

Most remaining failures occur when the executor produces outputs that are structurally valid enough to pass the validator, leaving the program with little opportunity to intervene. In these cases, the error is not syntactic or schema-level — the function name is valid, the call count is correct, and the arguments are well-typed — but semantic, such as selecting the wrong tool or providing incorrect argument values. Because these failures fall outside of the validator's scope, the program often reproduces the same incorrect call as Direct without triggering a retry or repair (`passthrough`: 41.5-72.9% per executor across 2670 both-fail samples; Table F.8). Another common pattern is that both direct and the program abstain when neither identifies a valid tool (`both_empty`: 5.9-16.2%). Together, `passthrough` and `both_empty` account for 57.7-80.7% of all *both-fail* cases, showing that the scaffold's effectiveness is ultimately bounded by the validator's scope: when failures are semantically wrong but structurally well-formed, PROTOCAL cannot detect or correct them.

4.3 When the program introduces regressions

A small fraction of failures arise not from the executor, but from the program's corrective behavior itself. These are cases where the scaffold is successfully enforcing consistency or specification compliance, but that very correction conflicts with how the benchmark is graded.

The most common source of regression comes from string-level rewrites. Here, the program follows validator constraints and rewrites outputs into canonical forms, but the benchmark expects more literal strings. For example, in `flight.status.check`, the rule "if only one name is mentioned assume last name is not specified and instead use LNU" leads the program to rewrite `'Karan'` as `'Karan LNU'`, which is flagged as incorrect. Similarly, `Hotels_2_SearchHouse` requires a `'City, State'` format, producing `'Delhi, DL'` instead of the expected `'Delhi'`. In `cmd_controller.execute`, the only allowed units are `['seconds', 'milliseconds']`, so the program selects `'seconds'` when no unit is given, while the evaluator expects `'N/A'`. Overall, these string-level rewrites account for 30% of all regressions (74 cases), concentrated in `retry+rewrote_string` and `validate+rewrote_string`.

Other regressions come from more discrete control decisions. In some cases, the program suppresses tool usage when a call is actually required, leading to `force_empty_wrong` errors (20% of losses), while in others it introduces unnecessary tool calls where Direct would correctly return text (`force_tool_call_wrong`, 13%). A smaller portion (12%) arises from over-decomposition, where the program generates multiple calls even though a single call is sufficient.

4.4 Which design choices matter?

We compare three pipeline revisions that isolate the two decisions affecting performance the most: V1 (baseline), V2 (`decompose_fallback` added), and V3 (silent corrections converted to flag-and-retry). Their feature presence is summarized in Table I.1; Table 4.2 reports the paired flip results.

Decomposition as recovery (V1 → V2). V1 issues a single LLM call per input and validates each returned tool call in place; it has no branch for re-issuing calls when the count is wrong. V2 keeps the same single-shot entry point and adds

Table 4.2: Paired flip results across pipeline versions on both executors. All rows use $n = 4079$ with runtime errors counted as wrong. The *Feature change* column names the behavioral row from Table I.1 that flips between a and b.

Pair	Executor	a AST	b AST	ΔAST	Feature change
V1 $\rightarrow$ V2	Granite 4.1 3B	83.33%	83.62%	$+0.29$	`decompose_fallback` added
V1 $\rightarrow$ V2	GPT-OSS-20B	72.69%	80.34%	$+7.65$	`decompose_fallback` added
V2 $\rightarrow$ V3	Granite 4.1 3B	83.62%	82.96%	-0.66	silent correction to flag and retry
V2 $\rightarrow$ V3	GPT-OSS-20B	80.34%	77.03%	-3.31	silent correction to flag and retry

a `decompose_fallback` step: the program generator declares one sub-task per expected tool call, and whenever the single-shot output returns fewer tool calls than declared sub-tasks, the fallback reissues one LLM call per missing sub-task. This is the only behavioral row that flips between V1 and V2. The contrast is the largest movement we measure in the version walk: $+7.65\%$ on GPT-OSS-20B ($72.69\% \rightarrow 80.34\%$) and $+0.29\%$ on Granite 4.1 3B. The asymmetry across executors reflects what each model makes mistakes: GPT-OSS-20B under-returns calls on parallel inputs , and the fallback supplies the missing calls; Granite 4.1 3B's failures are dominated by other categories, so it has little for the fallback to repair. What we learned is that the recovery branch matters more than the planning declaration itself — declaring `sub_tasks` only earns AST when a runtime branch consumes them.

Silent corrections vs flag-and-retry (V2 $\rightarrow$ V3). The two versions differ in what the validator does when it spots a problem in the executor's output. V2 *silently rewrites* the call: it deletes unknown arguments, fills missing required arguments from the spec's `default`, and strips optional arguments the prompt did not explicitly request, all before scoring. V3 instead *flags and retries*: the same diagnostic checks fire, but each finding is recorded as an `issue` and the retry loop asks the model to fix the call rather than the harness rewriting it behind the model's back. The inference-time cost is real: -3.31% on GPT-OSS-20B ($80.34\% \rightarrow 77.03\%$) and -0.66% on Granite 4.1 3B. The silent corrections were doing useful work for the executor at inference time.

4.5 Implications: program generator quality matters; size does not

The program does not generate tool calls; instead, the program generator only writes the control flow around that execution, including when calls are issued, how outputs are validated, and when retries are triggered. The impact of program-mediation therefore largely depends less on the quality of the program generator that produces it. This is most apparent in the effect of generator choice. Strong program generators consistently improve performance across executors, while a much smaller SFT-distilled generator can achieve nearly the same gains as a substantially larger teacher model. The difference is therefore not primarily about parameter count, but about whether the generator can reliably produce executable and well-structured control flow. When generator quality drops further, however, the scaffold becomes harmful rather than merely less helpful. Weak generators introduce brittle or incorrect control decisions that outweigh the benefits of validation and retry, leading to systematic degradation relative to Direct prompting. In this sense, the scaffold amplifies the quality of the code it is given: strong generators improve performance, while weak ones systematically degrade it. The most effective recipe that works is either (i) a strong off-the-shelf program generator, or (ii) SFT distillation from such a model into a smaller one.

5 Conclusion

We introduced PROTOCAL, a framework that scaffolds LLM tool-calling by synthesizing a Python program per prompt and running it against an executor. The synthesized programs decompose complex instructions into sub-tasks and detect, validate, and repair malformed calls at runtime. Across two benchmarks, seven executors, and three program generators, PROTOCAL delivers consistent gains over direct prompting and the generated programs are reliably executable, with averaged runtime errors 0.2% across all runs. A small SFT-distilled 8B program generator matches its $15\times$ larger teacher, showing that the lever is program quality, not generator scale. These results make a concrete case for generative computing as a way to orchestrate stochastic LLM components into reliable tool-calling workflows.

Limitations

Because PROTOCAL's program is generated once and then applied wholesale, its blanket validation-and-repair logic can introduce new errors: turning valid empty calls into refusals ("over_refused") or splitting single calls unnecessarily ("over_decomposed"). These misfires stem from the one-off code synthesis: there is no opportunity to detect that a program's intervention was unwarranted before it's executed, and our current setup lacks a built-in mechanism to selectively undo or refine those decisions. One of the future work direction can be investigate an agentic, iterative workflow in which the program generator, validator, and repair components loop with one another, resulting in feedback-driven agents that are able to catch and correct "over-refusal" or "over-decomposition" without sacrificing the robustness gains PROTOCAL has demonstrated.

PROTOCAL has so far been evaluated only on single-turn tool calls, i.e., each prompt yields one round of program synthesis and execution. Therefore, in another line of work, we will investigate an agentic, iterative workflow that loops between synthesis, validation, and repair across multiple turns, enabling the system to adapt its programmatic decisions in a conversational context.

Acknowledgments

Anonymized.

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, and 1 others. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*.

Anthropic. 2025. Claude haiku 4.5.

Hyungjoo Chae, Yeonghyeon Kim, Seungone Kim, Kai Tzu-iunn Ong, Beong-woo Kwak, Moohyeon Kim, Sunghwan Mac Kim, Taeyoon Kwon, Jiwan Chung, Youngjae Yu, and 1 others. 2024. Language models as compilers: Simulating pseudocode execution improves algorithmic reasoning in language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 22471–22502.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. 2025. Self-play with execution feedback: Improving instruction-following capabilities of large language models. In *International Conference on Learning Representations*, volume 2025, pages 39286–39313.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International conference on machine learning*, pages 10764–10799. PMLR.

Shijue Huang, Wanjun Zhong, Jianqiao Lu, Qi Zhu, Jiahui Gao, Weiwen Liu, Yutai Hou, Xingshan Zeng, Yasheng Wang, Lifeng Shang, and 1 others. 2024. Planning, creation, usage: Benchmarking llms for comprehensive tool utilization in real-world complex scenarios. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4363–4400.

IBM. 2025. Mellea. https://mellea.ai/.

IBM Research. 2025. Granite 4.0 language models. https://github.com/ibm-granite/granite-4.0-language-models. Accessed: 2025-10-01.

IBM Research. 2026. Granite 4.1 language models. https://huggingface.co/blog/ibm-granite/granit-4-1. Accessed: 2026-04-28.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, and 1 others. 2024. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *arXiv preprint arXiv:2406.18518*.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P Murphy. 2026. Autoharness: improving llm agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1384–1403.

Marvinai. 2023. Marvin. https://askmarvin.ai/welcome.

Mayank Mishra, Prince Kumar, Riyaz Bhat, Rudra Murthy, Danish Contractor, and Srikanth Tamilselvam. 2023. Prompting with pseudo-code instructions. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 15178–15197.

Van-Quang Nguyen and Takayuki Okatani. 2026. Coretab: Improving multimodal table understanding with code-driven reasoning. *arXiv preprint arXiv:2601.19193*.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, Lingjie Chen, Yanjun Zhao, Ke Yang, Bingxuan Li, Cheng Qian, Gaotang Li, Xiao Lin, Zhichen Zeng, Ruizhong Qiu, and 23 others. 2026. Code as agent harness. *Preprint*, arXiv:2605.18747.

Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

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 *International Conference on Learning Representations*, volume 2024, pages 9695–9717.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343.

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. *Advances in neural information processing systems*, 36:68539–68551.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

Nathaniel Weir, Muhammad Khalifa, Linlu Qiu, Orion Weller, and Peter Clark. 2024. Learning to reason via program generation, emulation, and search. *Advances in Neural Information Processing Systems*, 37:36390–36413.

Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. 2025. Toucan: Synthesizing 1.5m tool-agentic data from real-world mcp environments. *Preprint*, arXiv:2510.01179.

Qwen: An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, and 1 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Junjie Ye, Guoqiang Zhang, Wenjie Fu, Tao Gui, Qi Zhang, and Xuanjing Huang. 2026. Cctu: A benchmark for tool use under complex constraints. *CoRR*, abs/2603.15309.

Li Zhang, Liam Dugan, Hainiu Xu, and Chris Callison-Burch. 2023a. Exploring the curious case of code prompts. In *Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations (NLRSE)*, pages 9–17.

Li Zhang, Hainiu Xu, Yue Yang, Shuyan Zhou, Weiqiu You, Manni Arora, and Chris Callison-Burch. 2023b. Causal reasoning of entities and events in procedural texts. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 415–431.

Yuxiang Zhang, Jing Chen, Junjie Wang, Yaxin Liu, Cheng Yang, Chufan Shi, Xinyu Zhu, Zihao Lin, Hanwen Wan, Yujiu Yang, and 1 others. 2024. Toolbehonest: A multi-level hallucination diagnostic benchmark for tool-augmented large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11388–11422.

A Related Work

The use of code and pseudo-code as inputs to LLMs to enhance their performance has gained significant attention in recent years. A few initial attempts including (Mishra et al., 2023; Madaan et al., 2022; Zhang et al., 2023b,a) have demonstrated that pseudo-code instructions provide better performance over natural language instructions on a wide variety of tasks, such as, logical reasoning and instruction following, etc. More recently, a growing body of work has concentrated on reasoning tasks (Nguyen and Okatani, 2026; Weir et al., 2024; Chae et al., 2024), data generation and validation (Dong et al., 2025) and for Chess in GameArena (Lou et al., 2026). Another emerging focus involves exploiting Python programs for LLM interaction (IBM, 2025) and using structured code to design more effective AI applications and agents (Marvinai, 2023; Ning et al., 2026). These approaches highlight the versatility of code representations in extending the capabilities of LLMs across diverse domains.

A parallel line of work targets tool calling directly, training or prompting LLMs to invoke external functions (Schick et al., 2023; Qin et al., 2024; Yao et al., 2022; Qu et al., 2025), and benchmarks such as BFCL (Patil et al., 2025) have made the recurring failure modes concrete: executors hallucinate function names that do not exist in the provided schema (Zhang et al., 2024), emit malformed or type-mismatched arguments (Huang et al., 2024), miss required parallel calls when a prompt demands several tools at once (Patil et al., 2025), and commit to a confident call on prompts that ask only for information or that the available tools cannot satisfy. Mitigations in this literature largely operate *inside* the executor, through stronger instruction tuning, larger tool corpora, or richer in-context demonstrations (Schick et al., 2023; Qin et al., 2024), leaving the structural failure modes above only partially addressed.

Unlike aforementioned works, we focus on improving LLMs' tool-calling capabilities through synthesized Python programs.

B Main components of PROTOCAL

In Figure B.1, we provide a comparison of a standard tool-calling user prompt and a user input in our framework side by side.

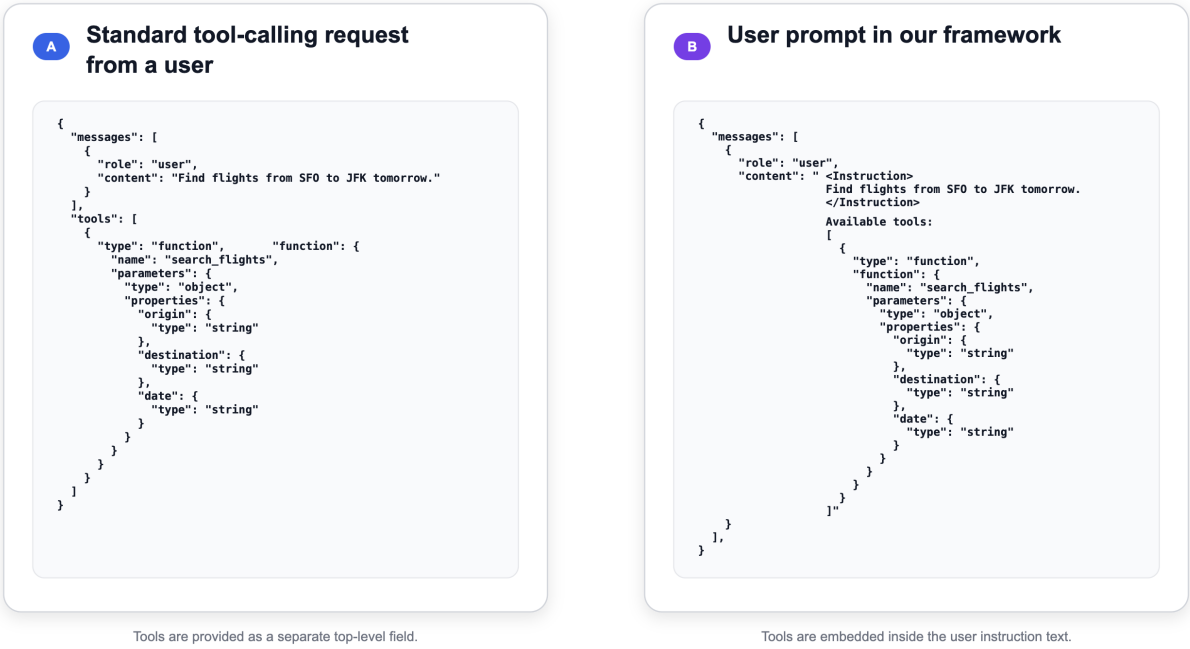

Figure B.1: User prompt in standard tool-calling request v.s. in our framework

In Listing 1, we show the complete input to a program generated by off-the-shelf open-source models, which includes the coding rules and six ICL examples, followed by the reformatted user prompt. Line 1 - 23 are the preamble coding rules and instructions, and starting from line 24, six ICL examples are provided before we present the current user prompt at line 386.

```
You are provided a natural language description of a task and a set of available tools. Your task is to generate a Python program that uses LLM_call() with tools and standard Python to solve the task. Use clear, direct, pythonic constructs. Explicitly handle all stated constraints. Keep the solution minimal, correct, and robust.

Key rules:
1. Strategy: First call LLM with the user's original instruction verbatim. Iterate over the returned tool calls and run validate_tool_call + any case-specific validators; if issues remain, call retry_single_tool_call. Do NOT re-implement name fuzzy-matching, required-param checking, enum validation, type coercion, variable-reference detection, or retry-prompt construction — those live in the helpers. Trust the model's output: do NOT strip optional params, do NOT auto-fill defaults.
2. Task decomposition (strictly limited): Do NOT decompose by default. Only split the request if it contains multiple explicitly independent actions that cannot be handled in a single tool call. One sub_task must correspond to exactly one tool call. If decomposition is necessary, define sub_tasks (minimal, coarse-grained), first try a single LLM call with the full prompt, then pass tool_calls through decompose_fallback to recover from count mismatch.
3. Irrelevance: If the tools list is non-empty but NO tool is relevant to the question, return an empty list []. Do not force a tool call.
4. Empty tools: If the tools list is empty [], just call LLM_call(prompt) without tools and return "content": result, "tool_calls": [].
5. Optional params: Trust the model's first-try output. Do NOT strip optional params. Do NOT add optional params the model didn't provide. Do NOT fill defaults for missing required params — let validate_tool_call flag them and retry fix them.
6. Prompt preservation: For the first LLM_call, use the user's original instruction verbatim. Retries are handled by retry_single_tool_call.
7. Respect empty tool_calls: If the LLM_call returns 0 tool calls AND returns text content, accept it and return []. Only retry when LLM_call returns 0 tool calls AND returns empty text content.
8. Nested objects: ensure the top-level parameter exists (if required) and is a dict; pass its contents through unchanged.
9. Typos in user input: Do NOT attempt to fix typos. Pass the user's words through as-is. Fixing typos risks changing the user's intent — especially in function names, variable names, parameter names, enum values, or domain-specific terms that may look misspelled but are intentional.
10. Tool metadata: When building valid_tool_names, tool_params, and required_params, copy ALL tools, ALL parameters, ALL enums, and ALL defaults from the tool spec exactly. Do not skip or abbreviate any tool or parameter.
11. Case-specific validators: Format constraints stated in the tool description (e.g. 'YYYY-MM-DD', 'City, ST') are NOT in the helpers. Add them after validate_tool_call, appending to the returned issues list so retry_single_tool_call picks them up.

Shared helpers (already defined in the boilerplate, do NOT redefine):
- validate_tool_call(name, args, tool_params, required_params, valid_tool_names) -> (name, args, issues): function-name fuzzy-match; FLAGS missing required args (does not auto-fill); enum + type coercion ONLY for args the model returned.
- retry_single_tool_call(original_task, name, args, issues, tools, valid_tool_names) -> (name, args): single-tool retry with original-task context.
- decompose_fallback(tool_calls, sub_tasks, tools, max_retries) -> tool_calls: if the model returned FEWER calls than sub_tasks, re-call each sub-task individually; if it returned equal or more calls, pass through unchanged (trust the model). Contract: one sub_task == one tool call. If you can't cleanly split the user's request into N pieces where each piece is exactly one call, don't decompose — just call LLM_call once with the full prompt and trust the mdel.

Each ICL example below should contain ONLY case-specific content: tool metadata (valid_tool_names, tool_params, required_params), sub_tasks (when decomposing), the original prompt, and case-specific custom validators (e.g. date format, "City, ST" format). Generic validation and retry must use the helpers above. Do NOT add pre-validation strip blocks — trust the model's output.

Below are a few examples enclosed in <Example> tags. Each has an <Instruction>, Available tools, and <Pseudocode>. Every pseudocode MUST end with exactly '# program' (lowercase, no extra words) followed by {"method", "return", "execute"}. Now generate pseudocode for the task that follows the examples.
<Example>
<Instruction>
Is 2024/01/01 a holyday in the Enoch Calendar?
</Instruction>

Available tools:
[{"type": "function", "function": {"name": "is_holyday", "description": "Checks if a given date is a holyday according to the Enoch Calendar API and retrieves associated scriptures and Sabbath status.", "parameters": {"type": "object", "properties": {"datestring": {"description": "The date string to check for holyday status.", "type": "string", "default": "2021-03-30"}}, "required": []}}}]

import re

def check_enoch_holyday() -> list:
    # tools is pre-injected  do not redefine it
```

```
    valid_tool_names = {"is_holyday"}
    tool_params = {
        "is_holyday": {
            "datestring": {"type": "string", "default": "2021-03-30", "required": False},
        },
    }
    required_params = {"is_holyday": set()}
    max_retries = 3

    prompt = "Is 2024/01/01 a holyday in the Enoch Calendar?"
    result = LLM_call(prompt, tools)

    tool_calls = result.get("tool_calls", []) if isinstance(result, dict) else []
    content = result.get("content", "") if isinstance(result, dict) else ""

    if not tool_calls and content.strip():
        return tool_calls

    validated = []
    for tc in tool_calls:
        fn = tc.get("function", {}) if isinstance(tc, dict) else {}
        name = fn.get("name", "")
        args = fn.get("arguments", {}) if isinstance(fn.get("arguments"), dict) else {}

        for attempt in range(max_retries):
            name, args, issues = validate_tool_call(name, args, tool_params, required_params, valid_tool_names)

            # Custom validation: datestring format (YYYY-MM-DD) and valid month/day
            if name == "is_holyday" and "datestring" in args:
                ds = str(args["datestring"])
                if not re.match(r"^\d{4}-\d{2}-\d{2}$", ds):
                    digits = re.findall(r"\d+", ds)
                    if len(digits) >= 3:
                        args["datestring"] = f"{digits[0]:0>4}-{digits[1]:0>2}-{digits[2]:0>2}"
                    else:
                        issues.append("datestring must be in YYYY-MM-DD format (default hint: 2021-03-30)")
                m = re.match(r"^(\d{4})-(\d{2})-(\d{2})$", str(args.get("datestring", "")))
                if m:
                    month, day = int(m.group(2)), int(m.group(3))
                    if not (1 <= month <= 12 and 1 <= day <= 31):
                        issues.append(f"datestring has invalid month/day: month={month}, day={day}")

            if not issues:
                break
            name, args = retry_single_tool_call(prompt, name, args, issues, tools, valid_tool_names)

        validated.append({"function": {"name": name, "arguments": args}})

    return validated

# program
{
    "method": "check_enoch_holyday",
    "return": "list",
    "execute": "check_enoch_holyday()"
}

</Example>
<Example>
<Instruction>
Is 64 a power of 8? What is the minimun element in the array [5, 6, 7, 8, 1, 2, 3]? What is the future value of an investment of $5000 at 8% annual interest over 5 years?
```

```
</Instruction>

Available tools:
[{"type": "function", "function": {"name": "is_power", "description": "Checks if a number is a power of a given base.", "parameters": {"type": "object", "properties": {"num": {"description": "The number to check.", "type": "integer"}, "base": {"description": "The base to check against.", "type": "integer"}}, "required": ["num", "base"]}}}, {"type": "function", "function": {"name": "find_minimum_rotated_sorted_array", "description": "Finds the minimum element in a rotated sorted array.", "parameters": {"type": "object", "properties": {"nums": {"description": "The rotated sorted array.", "type": "array"}}, "required": ["nums"]}}}, {"type": "function", "function": {"name": "future_investment_value", "description": "Calculates the future value of an investment.", "parameters": {"type": "object", "properties": {"present_value": {"description": "The present value of the investment.", "type": "number"}, "annual_interest_rate": {"description": "The annual interest rate as a decimal.", "type": "number"}, "years": {"description": "The number of years the investment will grow.", "type": "integer"}}, "required": ["present_value", "annual_interest_rate", "years"]}}}, {"type": "function", "function": {"name": "find_peak_element", "description": "Finds a peak element in a list of integers.", "parameters": {"type": "object", "properties": {"nums": {"description": "The list of integers.", "type": "array"}}, "required": ["nums"]}}}]

def answer_three_questions() -> list:
    # tools is pre-injected  do not redefine it
    valid_tool_names = {"is_power", "find_minimum_rotated_sorted_array", "future_investment_value", "find_peak_element"}
    tool_params = {
        "is_power": {
            "num": {"type": "integer", "required": True},
            "base": {"type": "integer", "required": True},
        },
        "find_minimum_rotated_sorted_array": {
            "nums": {"type": "array", "required": True},
        },
        "future_investment_value": {
            "present_value": {"type": "number", "required": True},
            "annual_interest_rate": {"type": "number", "required": True},
            "years": {"type": "integer", "required": True},
        },
        "find_peak_element": {
            "nums": {"type": "array", "required": True},
        },
    }
    required_params = {
        "is_power": {"num", "base"},
        "find_minimum_rotated_sorted_array": {"nums"},
        "future_investment_value": {"present_value", "annual_interest_rate", "years"},
        "find_peak_element": {"nums"},
    }
    max_retries = 3

    sub_tasks = [
        "Is 64 a power of 8?",
        "What is the minimun element in the array [5, 6, 7, 8, 1, 2, 3]?",
        "What is the future value of an investment of $5000 at 8% annual interest over 5 years?",
    ]

    prompt = "Is 64 a power of 8? What is the minimun element in the array [5, 6, 7, 8, 1, 2, 3]? What is the future value of an investment of $5000 at 8% annual interest over 5 years?"
    result = LLM_call(prompt, tools)

    tool_calls = result.get("tool_calls", []) if isinstance(result, dict) else []
    content = result.get("content", "") if isinstance(result, dict) else ""

    if not tool_calls and content.strip():
        return tool_calls
```

```

    tool_calls = decompose_fallback(tool_calls, sub_tasks, tools, max_retries)

    validated = []
    for idx, tc in enumerate(tool_calls):
        fn = tc.get("function", {}) if isinstance(tc, dict) else {}
        name = fn.get("name", "")
        args = fn.get("arguments", {}) if isinstance(fn.get("arguments"), dict) else {}
        original_task = sub_tasks[idx] if idx < len(sub_tasks) else prompt

        for attempt in range(max_retries):
            name, args, issues = validate_tool_call(name, args, tool_params, required_params, valid_tool_names)
            if not issues:
                break
            name, args = retry_single_tool_call(original_task, name, args, issues, tools, valid_tool_names)

        validated.append({"function": {"name": name, "arguments": args}})

    return validated

# program
{
    "method": "answer_three_questions",
    "return": "list",
    "execute": "answer_three_questions()"
}

</Example>
<Example>
<Instruction>
What is the capital of France?
</Instruction>

Available tools:
[]

def answer_capital_question() -> dict:
    # No tools available respond with text only, no tool calls
    prompt = "What is the capital of France?"
    result = LLM_call(prompt)

    return {"content": result, "tool_calls": []}

# program
{
    "method": "answer_capital_question",
    "return": "dict",
    "execute": "answer_capital_question()"
}

</Example>
<Example>
<Instruction>
What's the weather like on Mars today?
</Instruction>

Available tools:
[{"type": "function", "function": {"name": "get_current_weather", "description": "Get the current weather in a given location on Earth.", "parameters": {"type": "object", "properties": {"location": {"description": "The city and state, e.g. San Francisco, CA", "type": "string"}, "unit": {"description": "The temperature unit.", "type": "string", "enum": ["celsius", "fahrenheit"]}}, "required": ["location"]}}}]

def check_mars_weather() -> list:
    # tools is pre-injected  do not redefine it

```

```
    prompt = "What's the weather like on Mars today?"
    result = LLM_call(prompt, tools)

    tool_calls = result.get("tool_calls", []) if isinstance(result, dict) else []

    # The question is about Mars  no Earth weather tool is relevant.
    # If the model incorrectly called a tool, return empty.
    if tool_calls:
        tool_calls = []

    return tool_calls

# program
{
    "method": "check_mars_weather",
    "return": "list",
    "execute": "check_mars_weather()"
}

</Example>
<Example>
<Instruction>
Get current weather in Sacramento, CA in celsius, and search for Italian restaurants nearby that have vegetarian options.
</Instruction>

Available tools:
[{"type": "function", "function": {"name": "get_current_weather", "description": "Get the current weather in a given location.", "parameters": {"type": "object", "properties": {"location": {"description": "The city and state, e.g. San Francisco, CA", "type": "string"}, "unit": {"description": "The temperature unit.", "type": "string", "enum": ["celsius", "fahrenheit"]}}, "required": ["location"]}}}, {"type": "function", "function": {"name": "search_restaurants", "description": "Search for restaurants in a given area.", "parameters": {"type": "object", "properties": {"location": {"description": "The city to search in.", "type": "string"}, "cuisine": {"description": "Type of cuisine.", "type": "string", "enum": ["italian", "chinese", "mexican", "japanese", "indian", "french"]}, "has_vegetarian_options": {"description": "Filter for vegetarian options.", "type": "boolean"}}, "required": ["location", "cuisine"]}}}]

import re

def weather_and_restaurants() -> list:
    # tools is pre-injected  do not redefine it
    valid_tool_names = {"get_current_weather", "search_restaurants"}
    tool_params = {
        "get_current_weather": {
            "location": {"type": "string", "required": True},
            "unit": {"type": "string", "required": False, "enum": ["celsius", "fahrenheit"]},
        },
        "search_restaurants": {
            "location": {"type": "string", "required": True},
            "cuisine": {"type": "string", "required": True, "enum": ["italian", "chinese", "mexican", "japanese", "indian", "french"]},
            "has_vegetarian_options": {"type": "boolean", "required": False},
        },
    }
    required_params = {
        "get_current_weather": {"location"},
        "search_restaurants": {"location", "cuisine"},
    }
    max_retries = 3

    sub_tasks = [
        "Get current weather in Sacramento, CA in celsius.",
        "Search for Italian restaurants in Sacramento that have vegetarian options.",
    ]

```

```
    prompt = "Get current weather in Sacramento, CA in celsius, and search for Italian restaurants nearby that have vegetarian options."
    result = LLM_call(prompt, tools)

    tool_calls = result.get("tool_calls", []) if isinstance(result, dict) else []
    content = result.get("content", "") if isinstance(result, dict) else ""

    if not tool_calls and content.strip():
        return tool_calls

    tool_calls = decompose_fallback(tool_calls, sub_tasks, tools, max_retries)

    validated = []
    for idx, tc in enumerate(tool_calls):
        fn = tc.get("function", {}) if isinstance(tc, dict) else {}
        name = fn.get("name", "")
        args = fn.get("arguments", {}) if isinstance(fn.get("arguments"), dict) else {}
        original_task = sub_tasks[idx] if idx < len(sub_tasks) else prompt

        for attempt in range(max_retries):
            name, args, issues = validate_tool_call(name, args, tool_params, required_params, valid_tool_names)

            # Custom validation: location must be "City, ST" (per get_current_weather description hint)
            if name == "get_current_weather" and "location" in args:
                loc = str(args["location"])
                if not re.match(r"^.+,\s*[A-Z]{2}$", loc):
                    issues.append(f"location must be in 'City, ST' format (e.g. San Francisco, CA), got '{loc}'")

            if not issues:
                break
            name, args = retry_single_tool_call(original_task, name, args, issues, tools, valid_tool_names)

        validated.append({"function": {"name": name, "arguments": args}})

    return validated

# program
{
    "method": "weather_and_restaurants",
    "return": "list",
    "execute": "weather_and_restaurants()"
}

</Example>
<Example>
<Instruction>
My friends and I are taking our yearly trip and I would like some help finding three hotel rooms in Philadelphia please.
</Instruction>

Available tools:
[{"type": "function", "function": {"name": "Flights_4_SearchOnewayFlight", "description": "Search for one-way flights to the destination, allowing users to select various options such as departure date, seating class, and preferred airlines.", "parameters": {"type": "object", "properties": {"origin_airport": {"type": "string", "description": "The IATA code or name of the airport or city to depart from, such as 'SFO' for San Francisco International Airport."}, "destination_airport": {"type": "string", "description": "The IATA code or name of the airport or city to arrive at, such as 'LAX' for Los Angeles International Airport."}, "departure_date": {"type": "string", "description": "The start date of the trip in the format 'YYYY-MM-DD', e.g., '2023-07-15'."}, "seating_class": {"type": "string", "description": "The cabin seating class option for the flight.", "enum": ["Economy", "Premium Economy", "Business"], "default": "Economy"}, "number_of_tickets": {"type": "integer", "description": "The number of flight tickets required for the trip.", "default": 1}, "airlines": {"type": "
```

```
    string", "description": "The preferred airline company for air transport
    services. Use 'dontcare' to indicate no preference.", "enum": ["United Airlines"
    , "American Airlines", "Delta Airlines", "Southwest Airlines", "Alaska Airlines"
    , "British Airways", "Air Canada", "Air France", "South African Airways", "LOT
    Polish Airlines", "LATAM Brasil", "dontcare"], "default": "dontcare"}}, "
    required": ["origin_airport", "destination_airport", "departure_date"]}}}, {"
    type": "function", "function": {"name": "Hotels_4_SearchHotel", "description": "
    Searches for available accommodation in a specified location with optional
    filters such as star rating, smoking policy, and number of rooms.", "parameters"
    : {"type": "object", "properties": {"location": {"type": "string", "description"
    : "The city or town where the accommodation is searched for, in the format of '
    City, State' or 'City, Country'. State has to be abbreviated."}, "star_rating":
    {"type": "string", "description": "The star rating for the accommodation ranging
     from 1 to 5 stars.", "enum": ["1", "2", "3", "4", "5", "dontcare"], "default":
    "dontcare"}, "smoking_allowed": {"type": "string", "description": "Indicates if
    smoking is allowed. Choose 'True' for smoking rooms, 'False' for non-smoking
    rooms, or 'dontcare' if no preference.", "enum": ["True", "False", "dontcare"],
    "default": "dontcare"}, "number_of_rooms": {"type": "integer", "description": "
    The number of rooms required for the reservation.", "default": 1}}, "required":
    ["location"]}}}, {"type": "function", "function": {"name": "
    Travel_1_FindAttractions", "description": "Browse attractions in a given city,
    filtering by various criteria such as free entry, category, and suitability for
    children.", "parameters": {"type": "object", "properties": {"location": {"type":
     "string", "description": "The name of the city or town where the attraction is
    located, in the format of 'City, State' or 'City, Country', such as 'Paris,
    France' or 'New York, NY'."}, "free_entry": {"type": "boolean", "description": "
    Flag indicating whether the attraction offers free entry.", "enum": ["True", "
    False", "dontcare"], "default": "dontcare"}, "category": {"type": "string", "
    description": "Category of the attraction.", "enum": ["Place of Worship", "Theme
     Park", "Museum", "Historical Landmark", "Park", "Tourist Attraction", "Sports
    Venue", "Shopping Area", "Performing Arts Venue", "Nature Preserve", "dontcare"
    ], "default": "dontcare"}, "good_for_kids": {"type": "boolean", "description": "
    Flag indicating whether the attraction is suitable for children.", "enum": ["
    True", "False", "dontcare"], "default": "dontcare"}}, "required": ["location"
    ]}}}]

import re

def search_philadelphia_hotels() -> list:
    # tools is pre-injected  do not redefine it
    valid_tool_names = {"Flights_4_SearchOnewayFlight", "Hotels_4_SearchHotel", "
        Travel_1_FindAttractions"}
    tool_params = {
        "Flights_4_SearchOnewayFlight": {
            "origin_airport": {"type": "string", "required": True},
            "destination_airport": {"type": "string", "required": True},
            "departure_date": {"type": "string", "required": True},
            "seating_class": {"type": "string", "required": False, "enum": ["Economy
                ", "Premium Economy", "Business"]},
            "number_of_tickets": {"type": "integer", "required": False},
            "airlines": {"type": "string", "required": False, "enum": ["United
                Airlines", "American Airlines", "Delta Airlines", "Southwest
                Airlines", "Alaska Airlines", "British Airways", "Air Canada", "Air
                France", "South African Airways", "LOT Polish Airlines", "LATAM
                Brasil", "dontcare"]},
        },
        "Hotels_4_SearchHotel": {
            "location": {"type": "string", "required": True},
            "star_rating": {"type": "string", "required": False, "enum": ["1", "2",
                "3", "4", "5", "dontcare"]},
            "smoking_allowed": {"type": "string", "required": False, "enum": ["True"
                , "False", "dontcare"]},
            "number_of_rooms": {"type": "integer", "required": False},
        },
        "Travel_1_FindAttractions": {
            "location": {"type": "string", "required": True},
            "free_entry": {"type": "boolean", "required": False, "enum": ["True", "
                False", "dontcare"]},
            "category": {"type": "string", "required": False, "enum": ["Place of
                Worship", "Theme Park", "Museum", "Historical Landmark", "Park", "
```

```
                Tourist Attraction", "Sports Venue", "Shopping Area", "Performing Arts Venue", "Nature Preserve", "dontcare"]},
            "good_for_kids": {"type": "boolean", "required": False, "enum": ["True", "False", "dontcare"]},
        },
    }
    required_params = {
        "Flights_4_SearchOnewayFlight": {"origin_airport", "destination_airport", "departure_date"},
        "Hotels_4_SearchHotel": {"location"},
        "Travel_1_FindAttractions": {"location"},
    }
    max_retries = 3

    prompt = "My friends and I are taking our yearly trip and I would like some help finding three hotel rooms in Philadelphia please."
    result = LLM_call(prompt, tools)

    tool_calls = result.get("tool_calls", []) if isinstance(result, dict) else []
    content = result.get("content", "") if isinstance(result, dict) else ""

    if not tool_calls and content.strip():
        return tool_calls

    validated = []
    for tc in tool_calls:
        fn = tc.get("function", {}) if isinstance(tc, dict) else {}
        name = fn.get("name", "")
        args = fn.get("arguments", {}) if isinstance(fn.get("arguments"), dict) else {}

        for attempt in range(max_retries):
            name, args, issues = validate_tool_call(name, args, tool_params, required_params, valid_tool_names)

            # Custom validation: location must be "City, ST"
            if name == "Hotels_4_SearchHotel" and "location" in args:
                loc = str(args["location"])
                if not re.match(r"^.+,\s*[A-Z]{2}$", loc):
                    issues.append(f"location must be in 'City, ST' format, got '{loc}'")

            if not issues:
                break
            name, args = retry_single_tool_call(prompt, name, args, issues, tools, valid_tool_names)

        validated.append({"function": {"name": name, "arguments": args}})

    return validated

# program
{
    "method": "search_philadelphia_hotels",
    "return": "list",
    "execute": "search_philadelphia_hotels()"
}

</Example>
<Instruction>
Can Pikachu learn Thunderbolt?
</Instruction>

Available tools:
[{"type":"function","function":{"name":"get_pokemon_move_info","description":"Retrieves information about a Pok mon's move using the Pok API.","parameters":{"type":"object","properties":{"pokemon_name":{"description":"The name of the Pok mon.","type":"str","default":"run"},"move_name":{"description":"The name of the move to check. Defaults to \"run\".","type":"str, optional","default":"run"}},"required":[]}}},{"type":"function","function":{"name":"get_ip_zipcode","
```

```
description":"Retrieves the ZIP code of a given IP address using the ip-api.com
API.","parameters":{"type":"object","properties":{"ip":{"description":"The IP
address to locate.","type":"str"}},"required":[]}}}]
```


Listing 1: Full user prompt content.

C SFT training recipe

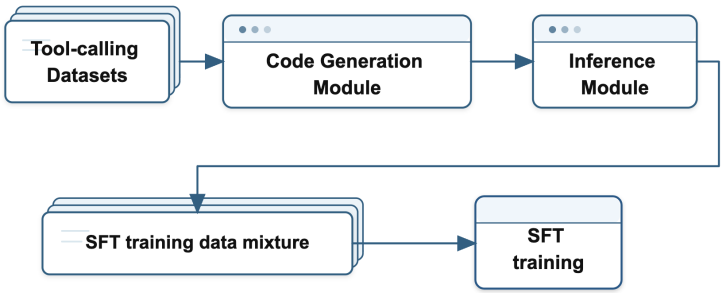

Figure C.1: SFT process

Figure C.1 outlines the pipeline used to develop a supervised fine-tuned (SFT) model for generating high-quality tool-calling programs. The process begins with curating a diverse collection of tool-calling datasets that represent a wide range of tool-use cases and scenarios. These datasets serve as a comprehensive foundation for SFT training. Next, the datasets are augmented with paired tool-calling programs generated by a larger LLM as program generator. The generated tool-calling programs are subsequently evaluated within the inference module by executing them with multiple LLM(s) of various sizes. These executors run the programs, obtain tool-calling outputs and then we evaluate those outputs. We filter out tool-calling programs when multiple executors obtained incorrect outputs, and retain only those with correct execution results. The validated tool-calling programs from this stage are then compiled into a specialized training mixture, which is used to fine-tune small models. By training these smaller, task-specific models, we retain the performance improvements typically associated with larger models, while significantly reducing computational costs.

C.1 SFT data mixture details

In Listing 2, we present a sample of the SFT data point.

Listing 2: An example of the SFT training data point


```
{
  "messages": [
    {
      "role": "system",
      "content": "You are provided a natural language description of a task and a set of available tools. Your task is to
            generate a Python program that uses LLM_call() with tools and standard Python to solve the task. Use clear,
            direct, pythonic constructs. Explicitly handle all stated constraints. Keep the solution minimal, correct, and
            robust. The helpers LLM_call(prompt, tools), validate_tool_call, retry_single_tool_call, and the variable tools
            are pre−defined  call them, don't redefine. End with result = your_function(...)."
    },
    {
      "role": "user",
      "content": "<Instruction>\nCan Pikachu learn Thunderbolt?\n</Instruction>\n\nAvailable tools:\n[\n {\n \"type\":
            \"function\",\n \"function\": {\n \"name\": \"get_pokemon_move_info\",\n \"description\": \"Retrieves
            information about a Pokmon's move using the PokAPI.\",\n \"parameters\": {\n \"type\": \"object\",\n
            \"properties\": {\n \"pokemon_name\": {\n \"description\": \"The name of the Pokmon.\",\n \"type\": \"str\",\n
            \"default\": \"run\"\n },\n \"move_name\": {\n \"description\": \"The name of the move to check. Defaults to
            \\\"run\\\".\",\n \"type\": \"str, optional\",\n \"default\": \"run\"\n }\n },\n \"required\": []\n }\n }\n },\n {\n \"type\":
            \"function\",\n \"function\": {\n \"name\": \"get_ip_zipcode\",\n \"description\": \"Retrieves the ZIP code of a
            given IP address using the ip−api.com API.\",\n \"parameters\": {\n \"type\": \"object\",\n \"properties\": {\n
            \"ip\": {\n \"description\": \"The IP address to locate.\",\n \"type\": \"str\"\n }\n },\n \"required\": []\n }\n }\n]"
```

```
    },
    {
      "role": "assistant",
      "content": "‘‘‘python\ndef check_pikachu_thunderbolt() -> list:\n # tools is pre-injected - do not redefine it\n
            valid_tool_names = {\"get_pokemon_move_info\", \"get_ip_zipcode\"}\n\n tool_params = {\n
            \"get_pokemon_move_info\": {\n \"pokemon_name\": {\"type\": \"str\", \"required\": False, \"default\":
            \"run\"},\n \"move_name\": ... (truncated)‘‘‘"
    }
  ],
  "uuid": "21f0a6c2c5144bc8",
  "dataset_name": "xlam_from_pavan",
  "source_model": "gpt-oss-120b"
}
```


And we present the pool of data mixture we used for SFT training in Table C.1.

Pattern	**Count**	**%**
has_tools	62,119	99.86%
empty_tools	85	0.14%
— call count —		
no_tool_call	10,472	16.83%
single_call	26,873	43.20%
multi_different_fn	14,408	23.16%
multi_same_fn	9,383	15.08%
multi_mixed_fn	1,068	1.72%
— argument usage —		
only_required_args	9,300	14.95%
uses_optional_arg	42,432	68.21%
uses_non_default_optional	41,700	67.04%
uses_default_matching_optional	5,356	8.61%
uses_array_arg	6,352	10.21%
uses_boolean_arg	1,140	1.83%
uses_enum_arg	133	0.21%
uses_nested_object	173	0.28%
— other —		
has_canonical_string_value	19,772	31.79%
spec_has_default	37,162	59.74%
spec_has_enum	137	0.22%

Table C.1: Summary of observed tool-calling patterns in the SFT tool-calling datasets.

D Implementation Details

Inference. All open-weight executors and adapters are served through a high-throughput inference engine (e.g., vLLM) on a multi-GPU node; closed-weight models are queried through their official APIs. Decoding uses greedy sampling with a fixed maximum generation length, and all models share the same tool schema and system prompt within a given condition to ensure a fair comparison.

E A worked example

User prompt. *"Could you please find out when Shishir's birthday is from our previous conversations and then congratulate him by sending a 'Happy Birthday!' message regardless?"*

Direct (Granite 4.1 3B). Returns 1 call where ground truth requires 2; BFCL flags parallel_function_checker:wrong_count.

```
[ recall_memory_search(query="Shishir's birthday", page=0, request_heartbeat=False) ]
```

Program (C4 scaffold, Granite 4.1 3B). Single-shot returns 1 call; `decompose_fallback` re-issues the second sub-task; `validate_tool_call` + `retry_single_tool_call` converge in 3 LLM calls (`resolution_stage=llm_retry`, `llm_calls=3`). BFCL: ✓.

```
[ recall_memory_search(query="Shishir's birthday", page=0, request_heartbeat=False),
  send_message(recipient="Shishir", message="Happy Birthday!") ]
```

Generated m-program (case-specific function only; tool metadata abbreviated).

```
def congratulate_shishir() -> list:
    valid_tool_names = {"send_message", "recall_memory_search",
                        "core_memory_append", ...}            # 9 tools
    tool_params = {
        "send_message": {
            "message":   {"type": "string",  "required": True},
            ...
        },
        # ... 8 more tools elided
    }
    required_params = {"send_message": {"message"},
                       "recall_memory_search": {"query"}, ...}
    max_retries = 3

    sub_tasks = [
        "Find Shishir's birthday from previous conversation history.",
        "Send a 'Happy Birthday!' message to Shishir.",
    ]
    prompt = "Could you please find out when Shishir's birthday ..."
    result = LLM_call(prompt, tools)

    tool_calls = result.get("tool_calls", []) if isinstance(result, dict) else []
    content    = result.get("content",    "") if isinstance(result, dict) else ""
    if not tool_calls and content.strip():
        return tool_calls                                # respect refusal

    tool_calls = decompose_fallback(tool_calls, sub_tasks, tools, max_retries)

    validated = []
    for idx, tc in enumerate(tool_calls):
        fn   = tc.get("function", {}) if isinstance(tc, dict) else {}
        name = fn.get("name", "")
        args = fn.get("arguments", {}) if isinstance(fn.get("arguments"), dict) else {}
        original_task = sub_tasks[idx] if idx < len(sub_tasks) else prompt
        for attempt in range(max_retries):
            name, args, issues = validate_tool_call(
                name, args, tool_params, required_params, valid_tool_names)
            if not issues:
                break
            name, args = retry_single_tool_call(
                original_task, name, args, issues, tools, valid_tool_names)
        validated.append({"function": {"name": name, "arguments": args}})
    return validated

result = congratulate_shishir()
```

Figure E.1: A program-rescues-Direct example from BFCL v3 `live_parallel_multiple`.

F Full Results

We present comprehensive experimental results for BFCL v3 in Table F.1 and CCTU in Table F.2, respectively. Granite 8B(SFT) means Granite 4.1 8B fine-tuned with SFT data mixture filtered only by GPT-OSS-120B execution correctness, and Granite 8B(SFT-UNION) means Granite 4.1 8B fine-tuned with SFT data mixture filtered by GPT-OSS-120B or Claude Haiku execution correctness.

Table F.1: BFCL v3 AST accuracy (%) of seven executors under direct prompting (**Direct**) and our program-mediated pipeline (**Program**) with four adapters. Δ is the absolute gain of the best adapter over Direct. Bold marks the best adapter per row.

		Program (ours)				
Executor	**Direct**	**GPT-OSS-120B**	**Granite 8B**	**Granite 8B (SFT)**	**Granite 8B (SFT-Union)**	**Δ**
Granite 4.1 3B	82.38%	**83.31%**	81.03%	82.93%	82.94%	+0.93
Granite 4.1 8B	84.70%	**84.94%**	83.33%	84.72%	84.47%	+0.24
Qwen 2.5 3B	73.38%	**77.27%**	74.58%	76.70%	76.41%	+3.89
Qwen 2.5 7B	77.79%	**81.39%**	78.12%	79.90%	79.68%	+3.60
GPT-OSS-20B	69.09%	**79.73%**	76.60%	78.85%	78.39%	+10.64
GPT-OSS-120B	68.92%	**77.89%**	74.73%	76.81%	76.80%	+8.97
Claude Haiku 4.5	83.75%	**83.99%**	80.31%	83.62%	83.50%	+0.24

Table F.2: CCTU accuracy (%) of seven executors under direct prompting (**Direct**) and our program-mediated pipeline (**Program**) with three adapters. Δ is the absolute gain of the best adapter over Direct. Bold marks the best adapter per row.

		Program (ours)				
Executor	**Direct**	**GPT-OSS-120B**	**Granite 8B**	**Granite 8B (SFT)**	**Granite 8B (SFT-Union)**	**Δ**
Granite 4.1 3B	24.50%	33.50%	31.00%	34.50%	**35.50%**	+11.00
Granite 4.1 8B	37.50%	**39.00%**	34.00%	38.00%	38.50%	+1.50
Qwen 2.5 3B	31.00%	**34.00%**	25.00%	33.50%	32.66%	+3.00
Qwen 2.5 7B	**40.00%**	37.50%	31.50%	39.00%	37.00%	-1.00
GPT-OSS-20B	22.00%	**29.00%**	20.50%	27.00%	28.50%	+7.00
GPT-OSS-120B	22.00%	**34.00%**	21.00%	**34.00%**	32.50%	+12.00
Claude Haiku 4.5	**42.00%**	37.00%	33.50%	36.50%	38.50%	-3.50

Table F.3: Error dynamics from direct prompting (**Direct**) to program-mediated execution with the best adapter per executor (**Program**). Counts are sample-level errors over BFCL v3 single-turn (n=4,251); raw `error_type` from the BFCL v3 checker.

	Granite 4.1 3B		Granite 4.1 8B		Qwen 2.5 3B		Qwen 2.5 7B		GPT-OSS-20B		GPT-OSS-120B	
Error Category	**Direct**	**M-Prog**	**Direct**	**M-Prog**	**Direct**	**M-Prog**	**Direct**	**M-Prog**	**Direct**	**M-Prog**	**Direct**	**M-Prog**
value_error:string	166	164	165	159	198	174	179	168	164	152	180	174
value_error:list/tuple	12	13	7	9	14	16	9	12	10	11	11	10
value_error:dict_key	5	5	4	3	5	5	4	5	2	4	3	5
value_error:dict_value	2	1	1	1	2	2	1	1	0	0	1	1
value_error:others	58	57	56	57	67	65	78	76	76	76	70	76
irrelevance:called_tool	173	128	166	140	364	231	357	225	194	162	179	156
relevance:no_tool_call	8	8	6	6	4	3	6	6	5	5	7	7
chatable:empty_response	0	0	0	0	0	0	0	0	0	2	1	0
missing:required_arg	4	1	1	0	3	0	2	0	12	0	11	0
wrong:function_name	19	17	16	12	45	38	19	19	15	17	22	16
hallucinated:function	0	0	0	0	7	1	3	2	21	1	11	1
wrong:type	13	16	18	19	31	30	29	26	32	24	42	33
extra:argument	1	0	3	4	10	5	15	7	5	2	3	2
simple_function_checker:wrong_count	38	39	10	13	22	22	14	18	48	43	68	70
simple_function_checker:missing_optional	33	31	22	21	82	74	12	9	18	19	13	17
parallel_function_checker:wrong_count	15	8	20	18	56	59	26	15	440	68	440	94
parallel_function_checker:cannot_find_match	40	42	29	32	58	57	39	43	0	49	0	54
multiple_function_checker:wrong_count	75	94	42	61	37	66	52	67	88	111	71	93
exec:mismatch	56	58	58	59	81	79	61	60	127	76	135	91

Table F.4: Per-BFCL-category accuracy (%) under direct prompting (**D**) and program-mediated execution with the best adapter (**P**). Green cells indicate improvements from D to P, while red cells indicate degradations.

	Granite 4.1 3B		Granite 4.1 8B		Qwen 2.5 3B		Qwen 2.5 7B		GPT-OSS-20B		GPT-OSS-120B	
BFCL Category	**D**	**P**	**D**	**P**	**D**	**P**	**D**	**P**	**D**	**P**	**D**	**P**
simple	93.8	93.5	96.2	96.2	94.5	93.2	95.5	95.2	87.2	87.2	85.5	84.5
multiple	92.5	92.0	97.0	97.0	91.0	90.0	95.0	95.0	83.5	82.0	84.0	82.5
parallel	90.5	92.0	91.5	91.0	73.0	74.0	89.0	90.5	0.0	77.7	0.0	71.5
parallel_multiple	89.0	89.4	90.5	90.5	79.0	76.4	85.5	86.4	0.0	70.2	0.0	62.3
live_simple	70.5	69.0	79.5	77.1	69.0	69.0	72.9	71.7	75.6	77.1	74.4	74.8
live_multiple	73.9	73.4	77.5	76.8	66.9	69.8	74.2	75.0	73.2	74.4	72.2	73.5
live_parallel	75.0	75.0	81.2	81.2	68.8	68.8	62.5	68.8	0.0	68.8	0.0	62.5
live_parallel_multiple	58.3	62.5	58.3	58.3	45.8	50.0	66.7	70.8	0.0	62.5	0.0	58.3
irrelevance	89.6	90.4	86.2	89.2	63.8	84.2	57.9	82.5	87.0	90.0	86.7	88.7
live_irrelevance	83.2	88.0	84.9	87.0	68.6	78.1	71.0	79.3	81.5	84.3	83.3	85.4
live_relevance	55.6	55.6	64.7	66.7	77.8	83.3	66.7	66.7	72.2	72.2	61.1	61.1
chatable	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.0	99.0	99.5	100.0
java	76.0	78.0	74.0	75.0	63.0	63.0	66.0	70.0	60.0	72.6	62.0	61.6
javascript	78.0	72.0	84.0	80.0	64.0	66.0	72.0	70.0	54.0	56.5	60.0	58.3
exec_simple	84.0	84.0	85.0	84.0	76.0	77.0	85.0	85.0	77.0	83.0	75.0	73.0
exec_multiple	80.0	78.0	78.0	76.0	72.0	72.0	74.0	76.0	72.0	68.0	60.0	62.0
exec_parallel	72.0	72.0	72.0	74.0	52.0	56.0	70.0	72.0	0.0	62.0	0.0	58.0
exec_parallel_multiple	60.0	57.5	55.0	55.0	52.5	50.0	55.0	52.5	0.0	40.0	0.0	40.0

Table F.5: Per-executor distribution of repair mechanisms among prog-wins (Direct ×, Program ✓). Each cell is n (%) where the percentage is normalized by that executor's prog-win total (last row). Bold marks the dominant mechanism for each executor.

Mechanism	**Granite 3B**	**Granite 8B**	**Qwen 7B**	**GPT-OSS 20B**
`decompose+retry`	5 (6.0%)	2 (3.5%)	8 (4.1%)	**365 (71.0%)**
`force_empty`	**46 (55.4%)**	**27 (47.4%)**	**146 (74.9%)**	50 (9.7%)
`force_tool_call`	1 (1.2%)	0 (0.0%)	5 (2.6%)	26 (5.1%)
`decompose_only`	1 (1.2%)	0 (0.0%)	2 (1.0%)	0 (0.0%)
`retry+rename_function`	1 (1.2%)	0 (0.0%)	2 (1.0%)	14 (2.7%)
`validate+rename_function`	0 (0.0%)	0 (0.0%)	0 (0.0%)	8 (1.6%)
`retry+enum/format snap`	10 (12.0%)	18 (31.6%)	9 (4.6%)	6 (1.2%)
`validate+enum/format snap`	4 (4.8%)	1 (1.8%)	5 (2.6%)	11 (2.1%)
`retry+type_coerce`	0 (0.0%)	2 (3.5%)	0 (0.0%)	5 (1.0%)
`validate+type_coerce`	1 (1.2%)	0 (0.0%)	0 (0.0%)	1 (0.2%)
`retry+add_arg`	5 (6.0%)	1 (1.8%)	1 (0.5%)	2 (0.4%)
`validate+add_arg`	2 (2.4%)	1 (1.8%)	1 (0.5%)	4 (0.8%)
`retry+drop_unknown_arg`	0 (0.0%)	1 (1.8%)	0 (0.0%)	2 (0.4%)
`validate+drop_unknown_arg`	1 (1.2%)	1 (1.8%)	9 (4.6%)	11 (2.1%)
`retry+value_rewrite`	4 (4.8%)	3 (5.3%)	5 (2.6%)	2 (0.4%)
`validate+value_rewrite`	0 (0.0%)	0 (0.0%)	2 (1.0%)	3 (0.6%)
`no_visible_change`	2 (2.4%)	0 (0.0%)	0 (0.0%)	4 (0.8%)
Total prog-wins	83 (100%)	57 (100%)	195 (100%)	514 (100%)

Table F.6: Per-executor distribution of failure modes among direct-wins (Direct ✓, Program ×). Each cell is n (%) where the percentage is normalized by that executor's direct-win total (last row). Bold marks the dominant failure mode for each executor.

Failure mode	**Granite 3B**	**Granite 8B**	**Qwen 7B**	**GPT-OSS 20B**
`force_empty_wrong`	8 (16.7%)	7 (13.7%)	8 (16.0%)	**26 (26.8%)**
`force_tool_call_wrong`	1 (2.1%)	1 (2.0%)	11 (22.0%)	18 (18.6%)
`over_decomposed`	7 (14.6%)	8 (15.7%)	7 (14.0%)	7 (7.2%)
`under_returned`	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
`program_error`	2 (4.2%)	2 (3.9%)	1 (2.0%)	7 (7.2%)
`retry+renamed_wrong`	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
`validate+renamed_wrong`	0 (0.0%)	0 (0.0%)	0 (0.0%)	2 (2.1%)
`retry+rewrote_string`	**13 (27.1%)**	**18 (35.3%)**	7 (14.0%)	7 (7.2%)
`validate+rewrote_string`	6 (12.5%)	6 (11.8%)	5 (10.0%)	12 (12.4%)
`retry+type_coerce_wrong`	2 (4.2%)	1 (2.0%)	3 (6.0%)	2 (2.1%)
`validate+type_coerce_wrong`	0 (0.0%)	1 (2.0%)	2 (4.0%)	3 (3.1%)
`retry+added_arg`	0 (0.0%)	0 (0.0%)	1 (2.0%)	0 (0.0%)
`validate+added_arg`	1 (2.1%)	1 (2.0%)	0 (0.0%)	5 (5.2%)
`retry+dropped_arg`	1 (2.1%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
`validate+dropped_arg`	1 (2.1%)	0 (0.0%)	0 (0.0%)	2 (2.1%)
`retry+rewrote_value`	5 (10.4%)	4 (7.8%)	4 (8.0%)	1 (1.0%)
`validate+rewrote_value`	1 (2.1%)	2 (3.9%)	1 (2.0%)	3 (3.1%)
`ground_truth_quirk`	0 (0.0%)	0 (0.0%)	0 (0.0%)	2 (2.1%)
Total direct-wins	48 (100%)	51 (100%)	50 (100%)	97 (100%)

Table F.7: What the program tried on samples where neither condition succeeded (Direct $\times$, Program $\times$). Each cell is n (%) where the percentage is normalized by that executor's both-fail total (last row). Bold marks the dominant behavior for each executor.

What the program did	Granite 3B	Granite 8B	Qwen 7B	GPT-OSS 20B
passthrough (no diff)	**427 (66.7%)**	**414 (72.9%)**	**506 (71.1%)**	311 (41.5%)
retry+no_change	18 (2.8%)	18 (3.2%)	21 (2.9%)	20 (2.7%)
both_empty	104 (16.2%)	40 (7.0%)	42 (5.9%)	114 (15.2%)
force_empty	8 (1.2%)	8 (1.4%)	9 (1.3%)	25 (3.3%)
force_tool_call	1 (0.2%)	0 (0.0%)	1 (0.1%)	13 (1.7%)
decompose+retry	7 (1.1%)	5 (0.9%)	18 (2.5%)	**117 (15.6%)**
decompose_only	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
under_returned	0 (0.0%)	0 (0.0%)	1 (0.1%)	0 (0.0%)
program_error	3 (0.5%)	2 (0.4%)	1 (0.1%)	17 (2.3%)
retry+rename_function	3 (0.5%)	2 (0.4%)	11 (1.5%)	7 (0.9%)
validate+rename_function	2 (0.3%)	0 (0.0%)	1 (0.1%)	3 (0.4%)
retry+enum/format snap	24 (3.8%)	30 (5.3%)	28 (3.9%)	12 (1.6%)
validate+enum/format snap	11 (1.7%)	17 (3.0%)	25 (3.5%)	49 (6.5%)
retry+type_coerce	2 (0.3%)	4 (0.7%)	4 (0.6%)	6 (0.8%)
validate+type_coerce	6 (0.9%)	4 (0.7%)	4 (0.6%)	2 (0.3%)
retry+add_arg	4 (0.6%)	1 (0.2%)	6 (0.8%)	5 (0.7%)
validate+add_arg	3 (0.5%)	2 (0.4%)	7 (1.0%)	13 (1.7%)
retry+drop_unknown_arg	3 (0.5%)	4 (0.7%)	2 (0.3%)	4 (0.5%)
validate+drop_unknown_arg	2 (0.3%)	2 (0.4%)	2 (0.3%)	10 (1.3%)
retry+value_rewrite	10 (1.6%)	11 (1.9%)	16 (2.2%)	8 (1.1%)
validate+value_rewrite	2 (0.3%)	4 (0.7%)	7 (1.0%)	14 (1.9%)
Total both-fails	640 (100%)	568 (100%)	712 (100%)	750 (100%)

Table F.8: Flip partition between Direct and the C4 program at fixed denominator $n = 4079$. *P win* = Direct $\times$, Program $\checkmark$; *D win* = the reverse. Fix rate = P-wins / Direct failures.

Executor	both ($\checkmark/\times$)	P win	D win	fix rate
Granite 4.1 3B	3309 / 639	83	48	11.5%
Granite 4.1 8B	3404 / 567	57	51	9.1%
Qwen 2.5 7B	3123 / 711	195	50	21.5%
GPT-OSS-20B	2719 / 749	514	97	40.7%

G Repair-mechanism taxonomy

We attribute each **prog-win** (Direct wrong, Program correct) and each **direct-win** (Direct correct, Program wrong) to a specific scaffold mechanism by combining two signals:

- the program's `resolution_stage` trace (`llm_call`, `llm_retry`, or `no_llm_call`); and
- a per-call diff between Direct's raw output and the program's final output over function name, argument keys, argument types, and argument values.

Let n_d and n_p denote the number of tool calls returned by Direct and by the program. Mechanisms are decided in priority order: structural rules (n_p vs. n_d, stage) are checked first; only when call counts match are per-call diff tags considered.

G.1 Prog-win mechanisms

Each rule fires only if the rules above it did not.

`force_empty`
Rule: $n_p = 0$ and $n_d > 0$, *or* `stage = no_llm_call`.
Description: The program returned `[]` on a prompt whose ground truth is `[]`. The executor hallucinated a call (or the program's pre-helper short-circuited the LLM entirely); the program forced the output back to empty.

`decompose+retry`
Rule: $n_p > n_d$ and `stage = llm_retry`.
Description: Validator flagged the under-decomposed first attempt; the executor was re-prompted with the typed `issues` list and the retry produced the missing parallel call(s).

`force_tool_call`
Rule: $n_d = 0$, $n_p = 1$, and `stage = llm_call`.
Description: Direct emitted prose or refused to call a tool; the program's prompt template forced exactly one tool call on the first attempt, with no retry needed.

`decompose_only`
Rule: $n_p > n_d$, `stage = llm_call`, and not the `force_tool_call` case (i.e., $n_p \geq 2$).
Description: The program elicited additional parallel calls on the first attempt without any validator retry — the decomposition came from the program's prompt scaffolding alone.

`retry+`⟨tag⟩
Rule: $n_p = n_d$, `stage = llm_retry`, and at least one per-call diff tag fires.
Description: Same call count as Direct, but the re-prompted LLM produced different (correct) function names, keys, types, or values. The fix came from the LLM under validator feedback.

`validate+`⟨tag⟩
Rule: $n_p = n_d$, `stage = llm_call`, and at least one per-call diff tag fires.
Description: Same call count as Direct, no retry happened, but the program's deterministic post-processing (fuzzy matching, canonicalization, default-fill, type coercion) rewrote the call without re-prompting the LLM.

`no_visible_change`
Rule: $n_p = n_d$ and no diff tag fires.
Description: Direct and Program emit identical calls yet the grader flips. Reflects evaluator non-determinism or grader artifacts; rare.

G.2 Per-call diff tags

When $n_p = n_d$, calls are zipped position-wise and compared. For each pair (d_i, p_i) the following tags fire:

`rename_function` function names differ.

`add_arg` program has argument keys absent from Direct.

`drop_unknown_arg` Direct has argument keys absent from program.

`type_coerce` for a shared key, the Python type of the value changed (e.g., `"5"` $\rightarrow$ `5`).

`enum_or_format_snap` for a shared key, a *string* value changed (canonicalization, enum match).

`value_rewrite` for a shared key, a *non-string* value changed.

If multiple tags fire, the primary tag is selected by priority:

`rename_function` $\succ$ `type_coerce` $\succ$ `enum_or_format_snap` $\succ$ `add_arg` $\succ$ `drop_unknown_arg` $\succ$ `value_rewrite`.

The mechanism label is then `retry+`$\langle$primary$\rangle$ if `stage = llm_retry`, else `validate+`$\langle$primary$\rangle$.

G.3 Direct-win (breakage) modes

Symmetric attribution for samples where Direct is correct and the program breaks them.

`force_empty_wrong`
Rule: $n_p = 0$ and $n_d > 0$, *or* `stage = no_llm_call`.
Description: Program returned `[]` on a prompt that genuinely needed a call. Symmetric loss-side counterpart of `force_empty`.

`force_tool_call_wrong`
Rule: $n_d = 0$, $n_p = 1$.
Description: Direct correctly emitted no call, but the program's template forced a (wrong) tool call.

`over_decomposed`
Rule: $n_p > n_d$ and not the `force_tool_call_wrong` case.
Description: Program added a redundant call where fewer were sufficient.

`under_returned`
Rule: $0 < n_p < n_d$.
Description: Program dropped a needed call without going to empty.

`retry+`$\langle$tag$\rangle$`_wrong`, `validate+`$\langle$tag$\rangle$`_wrong`
Rule: $n_p = n_d$ with diff tags `renamed_wrong`, `type_coerce_wrong`, `rewrote_string`, `added_arg`, `dropped_arg`, or `rewrote_value`, with `retry+` / `validate+` prefix set by stage.
Description: Same per-call signals as the win side, misfiring on samples where Direct was already correct.

`ground_truth_quirk`
Rule: $n_p = n_d$ and no diff tag fires.
Description: Direct and Program emit identical calls yet the grader marks one correct and the other wrong; indicates a grader-level artifact rather than a real repair.

G.4 Reading the labels

The `retry+` prefix means the LLM produced the fix on a validator-driven re-prompt. The `validate+` prefix means the program's own code path (fuzzy match, canonicalization, default-fill) produced the fix without re-invoking the LLM. The classifier observes only the structural diff and the `resolution_stage` trace; it does not inspect which specific code path inside the program performed a `validate+` rewrite.

H Case studies: `retry+rewrote_string` losses

To make the `retry+rewrote_string` failure mode concrete, we walk through four representative direct-wins where Direct is correct but the program's validator-driven retry rewrites a string argument and the grader rejects the rewrite. The four cases are sampled from Granite 4.1 8B's 18 `retry+rewrote_string` losses (the executor with the largest share of this mode at 35.3% of its direct-wins).

In three of the four cases, the program's rewrite is a faithful execution of an explicit instruction in the tool's parameter description; only the fourth case (date canonicalization) is a judgment call where the spec is silent.

Case 1: `cmd_controller.execute` — spec/grader contradiction.

- **User prompt:** "list file in desktop using dir Desktop"
- **Argument:** `unit`
- **Direct:** `'N/A'` (accepted)
- **Program (after retry):** `'seconds'` (rejected)
- **Spec excerpt:** *"The unit of measurement for the command execution time, such as 'seconds' or 'milliseconds'."* `enum: ['seconds', 'milliseconds']`, `default: 'N/A'`.
- **Grader expected:** `[", 'N/A']` (case insensitive).

The spec is internally inconsistent: the default value (`'N/A'`) is not in the enum (`['seconds', 'milliseconds']`). Direct happened to copy the default and was accepted; the program's validator flagged the off-enum value and re-prompted the executor, which selected an in-enum value (`'seconds'`). The program followed the enum constraint; the grader followed the default.

Case 2: `flight.status.check` — spec instructs the rewrite.

- **User prompt:** "Hi, I am Karan. Can you tell me the current status of my Indigo flight? My ticket number is IND4567."
- **Argument:** `person_name`
- **Direct:** `'Karan'` (accepted)
- **Program (after retry):** `'Karan LNU'` (rejected)
- **Spec excerpt:** *"The full name or first name of the passenger as it appears on the ticket, in the format 'First Last'. If only one name mentioned assume last name is not specified and instead use LNU."*
- **Grader expected:** `'Karan'`.

The parameter description explicitly tells the model to use `LNU` when only a first name is given. The retry produced `'Karan LNU'` per this rule; the grader rejected it in favor of the bare prompt token.

Case 3: `Hotels_2_SearchHouse` — spec mandates a format the grader rejects.

- **User prompt:** "Search for a house accommodation in Delhi that has a review rating of at least 4.6 for two?"
- **Argument:** `where_to`
- **Direct:** `'Delhi'` (accepted)
- **Program (after retry):** `'Delhi, DL'` (rejected)

- **Spec excerpt:** *"The destination where the house is searched for, in the format of 'City, State', such as 'Austin, TX' or 'San Francisco, CA'."*
- **Grader expected:** `'Delhi'`.

The spec mandates `'City, State'` format with two explicit examples. The retry complied (`'Delhi, DL'`); the grader expected the literal prompt token.

Case 4: `concert.find_nearby` — spec is silent, judgment call.

- **User prompt:** "Find me all the classical concerts near Berlin and Paris happening next Friday, and I am interested only in those with available parking."
- **Argument:** `date`
- **Direct:** `'next Friday'` (accepted)
- **Program (after retry):** `'2023-11-03'` (rejected)
- **Spec excerpt:** *"The date on which the user wants to attend a concert."*
- **Grader expected:** `'next Friday'`.

The spec does not mandate a format. The retry resolved the relative date to ISO; the grader preferred literal pass-through of the prompt phrase. Unlike the previous three cases, both answers are defensible.

Summary. Three of the four cases are scaffold-faithful executions of the tool spec that the benchmark grader penalizes (spec/grader contradiction, explicit format instruction ignored by the grader, mandated format ignored by the grader). The `retry+rewrote_string` failure mode therefore conflates two distinct phenomena: genuine over-canonicalization (Case 4) and benchmark-spec mismatch (Cases 1–3). A grader that respected the parameter descriptions would flip Cases 1–3 from direct-wins back to neutral.

I Archived Program Versions

Table I.1: Feature presence across pipeline versions V1–V3 for the same input. ✓= present, ×= absent. *Silent* = the rule overwrites the model's output before scoring; *flag* = the rule records an `issue` and lets the retry loop fix it.

Feature	**V1**	**V2**	**V3**
`decompose_fallback` (recover when #calls $\neq$ #sub-tasks)	×	✓	✓
Validate function name + args (fuzzy-match, type coerce)	✓	✓	✓
Per-call retry loop (`max_retries=3`)	✓	✓	✓
`issues` list accumulated per attempt	✓	✓	✓
Unknown args	silent delete	silent delete	flag
Missing required args	flag	fill from default	flag
Optional args (unrequested)	keep	silent strip	flag
Enum snap (close-match)	✓	✓	✓