Switchcraft: AI Model Router for Agentic Tool Calling

Sharad Agarwal
Microsoft Research
sagarwal@microsoft.com

Pooria Namyar
Microsoft Research
namyarpooria@microsoft.com

Alec Wolman
Microsoft Research
alecw@microsoft.com

Rahul Ambavat
Microsoft
raambava@microsoft.com

Ankur Gupta
Microsoft
angup@microsoft.com

Qizheng Zhang*
Stanford
qizhengz@stanford.edu

Abstract

Agentic AI systems that invoke external tools are powerful but costly, leading developers to default to large models and overspend inference budgets. Model routing can mitigate this, but existing routers are designed for chat completion rather than tool use. We present **Switchcraft**, the first (to the best of our knowledge) model router optimized for agentic tool calling. Switchcraft operates inline, selecting the lowest-cost model subject to correctness. We construct an evaluation framework on five function-calling benchmarks and train a DistilBERT-based classifier, deployed under a latency budget. Switchcraft achieves 82.9% accuracy—matching or exceeding the best individual model—while reducing inference cost by 84%, saving over $3,600 per million queries. We find that larger models do not consistently outperform smaller ones on tool-use tasks, and that nominally cheaper models can incur higher total cost due to token-intensive reasoning. Our work enables cost-aware agentic AI deployment without sacrificing correctness.

1 Introduction

Agentic AI systems – where LLMs invoke external tools and APIs to perform complex tasks – are emerging as powerful solutions for multiple domains [1, 15], but they incur substantial costs. In interviews with a dozen enterprise teams, we found that selecting models for tool-assisted queries remains challenging: teams default to large, widely-used models, leading to significant overspending—a problem for customers who overpay and for service providers facing over-subscribed GPU infrastructure even when smaller models would suffice.

Model routing addresses this by dynamically selecting an appropriate LLM per query, routing simple requests to smaller models and reserving large models for harder ones. Prior work shows substantial savings; e.g., an AWS service reports $\sim$ 43.9% cost reduction through intelligent routing [9]. However, existing routers [22, 7, 17] target chat completion and do not address agentic tool calling, where models must generate precise tool invocations across multiple steps—requirements that differ fundamentally from chat tasks and make existing routers ill-suited.

To support agentic tool-calling, we curate a diverse set of benchmarks for agentic workloads and use them to fine-tune a specialized routing model. We aggregate five public function-calling benchmarks—Berkeley Function Calling Leaderboard (BFCL v3) [24], AWS ConFETTI [3], Salesforce xLAM-60K [36], Glaive Function Calling [11], and Hermes [20]—covering tool use, multi-turn interaction, and parallel API calls. We build a unified evaluation framework that normalizes these datasets, executes each query across candidate LLMs, and uses an abstract syntax tree (AST)-based checker to score tool invocations robustly. Using these signals, we fine-tune **Switchcraft**, a DistilBERT-based

*worked performed while an intern at Microsoft Research

Available tools: `get_symbol_by_name`, `add_to_watchlist`, `get_stock_info`, `place_order`, `get_order_details`, `send_message`, ...

Turn 1 – User: "Integrate stock for 'Omega Industries' into my watchlist effectively."

```
Correct: get_symbol_by_name(name="Omega Industries") → {"symbol": "OMEG"}
         add_to_watchlist(stock="OMEG")
```

Turn 3 – User: "Execute a transaction for 150 shares at the present market value for the stock we just added."

```
Correct: get_stock_info(symbol="OMEG") → {"price": 457.23, ...}
         place_order(order_type="Buy", symbol="OMEG", price=457.23, amount=150)
```

```
Wrong function:   place_order(order_type="Buy", symbol="OMEG", price=557.23, amount=150)   ×
```
Skipping `get_stock_info` and using a hallucinated or stale price causes the order to be placed at wrong price.

```
Wrong parameters: place_order(order_type="Sell", symbol="OMEG", price=457.23, amount=150)   ×
```
Selling instead of buying: a single wrong parameter value causes the *opposite* of the intended action.

```
Wrong value:      place_order(order_type="Buy", symbol="OMEG", price=457.23, amount=1500)   ×
```
An order-of-magnitude error in `amount` causes a 10× larger financial commitment.

Figure 1: Motivating example from BFCL v3 [24]. Agentic queries demand *correct functions* and *precise parameter values* at every step; small errors (wrong type, wrong value, wrong order) produce consequential failures.

router (66M parameters) that takes an agent's query and context as input and predicts the most suitable model for execution.

Key Results. Switchcraft achieves 82.9% accuracy—matching or exceeding the best individual model (GPT-5.3-chat at 82.3%)—while reducing inference cost by 84%, saving over $3,600 per million queries. Relative to an oracle that always selects the cheapest correct model (89.4%), Switchcraft closes 37% of the accuracy gap. We further find that costlier models do not consistently outperform cheaper ones (GPT-5.4 trails GPT-5.3-chat at 81.3% vs. 82.3%), nominally cheaper models can incur higher cost due to verbose output, and a chat-fine-tuned router of the same architecture significantly underperforms Switchcraft (Appendix O).

To the best of our knowledge, this is the first system to address LLM model selection for tool-augmented, multi-turn tasks. We make three main contributions:

- We formulate model routing for agentic use-cases and identify key challenges.
- We build a unified evaluation framework for agentic routing, including multi-benchmark normalization, corrected ground-truth annotations, and an AST-based comparison framework for robust tool-calling evaluation.
- We develop Switchcraft, a DistilBERT-based model router that significantly improves cost versus quality trade-offs, achieving near-oracle accuracy while reducing inference cost by 84%.

2 Motivation

Figure 1 illustrates why routing for agentic tool calling differs from routing for chat completion. A user asks an AI agent to add "Omega Industries" to a watchlist and place a market order; fulfilling this request requires a *sequence* of tool invocations (resolve ticker, add to watchlist, fetch price, place order) where each step depends on the previous one. Unlike chat, where a paraphrase may be acceptable, agentic errors compound and can be irreversible: hallucinating a price puts the order at the wrong value; confusing `"Buy"` with `"Sell"` executes the *opposite* of the user's intent; a single digit error in `amount` causes a ten-fold larger financial commitment. At the same time, not all variation is an error—independent parallel calls can be issued in any order, and string parameters often admit semantically equivalent forms (e.g., `"Illinois"` vs. `"IL"`); a correct evaluator must accept these (full discussion in Appendix A). These properties—sequential dependencies, strict parameter precision, and selective tolerance for variation—motivate a router specifically fine-tuned on agentic data with evaluation metrics that capture the unique correctness criteria of tool-calling.

3 Design

We explored a large space of architectures, input representations, scoring methods, and routing strategies (frozen-embedding MLPs, larger encoders, LLM-as-a-router, similarity routing, cost-

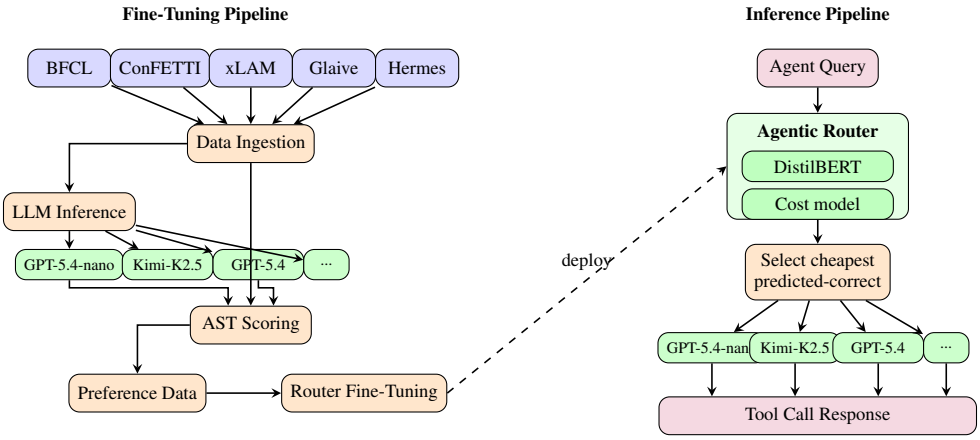

Figure 2: System architecture. **Left:** fine-tuning pipeline (ingest benchmarks, run inference across LLMs, score via AST, fine-tune router). **Right:** inference-time routing (DistilBERT classifier and cost model select the cheapest predicted-correct LLM).

weighted losses, BLEU and LLM-as-judge scoring); rejected alternatives are detailed in Appendix B. Figure 2 shows the final architecture: a fine-tuning pipeline that ingests function-calling benchmarks, runs every query against every candidate LLM, scores outputs via AST comparison, and distils the resulting preference data into a lightweight DistilBERT classifier; and an inference pipeline that runs the classifier plus a cost model to select the cheapest predicted-correct LLM. We build a router specialized for function-calling to avoid diluting it across heterogeneous workloads such as chat. Dispatching to the correct router is trivial— function-calling requests are identified by the presence of the `tools` parameter in the request body.

3.1 Input representation

A key design choice is how to pack an agentic query—multi-turn conversation, tool definitions, and metadata—into DistilBERT's 512-token window. Our packing operates in five steps:

1. **Latest user turn** (immediate intent, always included).
2. **Tool signatures** converted to compact `func_name(param1, param2)` form (descriptions and JSON-schema boilerplate stripped, etc.; capped at 100 subword tokens with `[truncated]`).
3. **Earlier turns** added greedily in reverse chronological order until the budget is exhausted, prioritizing recent context.
4. **Metadata preamble** Three numeric features: `length, num_tools, num_turns` for complexity-aware routing without relying solely on text.
5. **Concatenate and tokenize** (512-token truncation, dynamic padding).

3.2 Scoring the tool-calls

Executing each LLM-issued tool call is infeasible at our scale: queries span thousands of distinct, often private third-party APIs. Instead, we label each call statically by comparing its abstract syntax tree (AST)—function name, argument names, types, and values—to that of the ground-truth call.

The AST checker determines which (model, query) pairs the router sees as "correct". However, designing a general scorer is surprisingly hard: (1) the same call admits many semantically equivalent forms (set- vs. list-valued arguments, omitted default values, alternate string encodings), (2) tool-call arguments can be arbitrarily nested objects that must be compared structurally not by direct equality, and (3) the same syntactic shape across benchmarks (e.g., a list of lists) can carry different meanings.

For these reasons, the existing AST checker from BFCL [24] handles its native benchmark well but breaks on every other one: scoring the same GPT-5.3-chat outputs yields much lower accuracies on the four non-BFCL datasets (Table 1). Through a systematic study of rejected calls across all five

Table 1: AST checker comparison on GPT-5.3-chat results. *FN*: our checker accepts but BFCL rejects; *FP*: inverse. † Single-turn BFCL categories only (Simple, Multiple, Parallel, Parallel+Multiple, and their *Live* variants); multi-turn categories are excluded because BFCL's checker does not run on per-turn evaluation records.

Dataset	# Entries	Accuracy (%)		FN	FP
		BFCL	Ours		
BFCL [24]†	2,351	84.13	**86.73**	67	6
Glaive [11]	83,814	43.99	**88.46**	37,277	2
ConFETTI [3]	506	0.00	**53.16**	269	0
xLAM-60K [36]	60,000	17.35	**82.32**	39,302	323
Hermes [20]	8,940	59.73	**86.49**	2,531	139
Overall	155,611	35.08	**85.84**	79,446	470

datasets, we identify recurring classes of bias in BFCL's AST checker; we summarize each below, with examples in Appendix D (Table 9).

1. **Array order sensitivity.** BFCL compares arrays element-wise, rejecting set-valued parameters when the model emits them in a different order than the ground truth. We compare flat arrays as multisets and lists of dictionaries as sets.
2. **No default-parameter awareness.** BFCL treats every parameter listed in the ground truth as required. We parse defaults from the tool description and accept an omission when the documented default matches the ground truth.
3. **Brittle string matching.** Differences in case, whitespace, punctuation, and ISO-8601 timestamp formatting cause spurious mismatches. We canonicalize each string and fall back to DistilBERT cosine similarity (threshold 0.85) for multi-word strings.
4. **No nested-structure handling.** BFCL does not recognize the JSON-Schema `"object"` type and aborts the entry; even when fixed, it has no recursive comparator. We add the missing types and recursively validate each nested object against its sub-schema.
5. **Ambiguous array semantics.** A list-of-lists in the ground truth can mean either a list of *alternative* valid values or a true *nested* array. We use the tool-call schema to disambiguate.

With our AST checker, the same models show much more consistent accuracy across all five datasets (Table 1). We validate it via unit tests, manual review of random entries, and LLM-as-judge scoring.

3.3 Routing logic

At inference time, the router selects a single LLM for each incoming query in two stages:

Stage 1: multi-label classification. The DistilBERT classifier outputs a probability for each of the K candidate models, indicating the likelihood that the model will answer the query correctly. These probabilities are thresholded at 0.5 to produce a binary vector of predicted-correct models.

Stage 2: cost-aware selection. Given the set of predicted-correct models, the router selects the one with the **lowest profiled cost**: the actual dollar cost computed from the input/output token counts observed when each candidate model answered training queries, multiplied by its per-million-token list prices (Table 3; full formula in Appendix C). This naturally captures *chattiness*: a model that generates extensive reasoning or verbose output accumulates a higher profiled cost than a concise model at the same per-token rate. We analyze chattiness in detail in Section 4.8.

If no model is predicted correct (all probabilities below 0.5), the router falls back to the model with the highest probability (argmax). This ensures graceful degradation: even when the classifier is uncertain, it routes to its best guess rather than refusing to route.

Oracle router. Our oracle upper bound uses the same two-stage logic with perfect information: for all queries, including the test set, it knows which models answer correctly and selects the cheapest

correct one. When no model is correct, the oracle defaults to the most expensive model, providing a tight upper bound on any router that does not change the underlying models' answers.

4 Evaluation

We evaluate Switchcraft on five function-calling benchmarks (below), comparing it against individual LLMs, heuristic baselines, and an oracle upper bound.

4.1 Datasets

We combine five function-calling benchmarks spanning a broad range of tool-calling complexity, totaling 157,101 examples (122,267 after deduplication) across 14 categories (Table 2); diversity is essential to avoid overfitting to any single benchmark's distribution.

Table 2: Datasets and splits used in our evaluation. *ST* = single-turn, *MT* = multi-turn, *par* = parallel calls.

Source	Split	Examples	Type
BFCL v3 [24]	Simple	400	ST
	Multiple	200	ST
	Parallel	200	ST, par
	Parallel+Multiple	200	ST, par
	Live Simple	258	ST
	Live Multiple	1,053	ST
	Live Parallel	16	ST, par
	Live Par.+Multiple	24	ST, par
	Multi-turn Base (per turn)	745	MT
	Multi-turn Long Ctx (per turn)	745	MT
ConFETTI [3]	Conversations (cleaned)	506	MT
Glaive [11]	Function Calling v2	83,814	MT
xLAM-60K [36]	Function Calling 60K	60,000	ST
Hermes [20]	Function Calling v1	8,940	ST/MT

The corpus spans synthetic and human-authored data, single- and multi-turn conversations. We decompose multi-turn conversations into per-turn evaluation records, which is reflected in Table 2. Several datasets required substantial cleaning—most notably ConFETTI, where we corrected 27% of entries (Appendix E)—and format normalization to a common BFCL-style JSONL schema (Appendix F). All datasets pass through a unified pipeline: deduplication on (query, tools), per-dataset stratified 80/10/10 splits, and multi-label annotation by running each query through all candidate LLMs and recording correctness per our AST framework (Section 3.2).

4.2 Experimental setup

We route among eight LLMs spanning four model families (Table 3), accessed through API endpoints (default temperature; all values in USD). For each of the 14 dataset splits, we run every query against all eight models using the OpenAI-compatible function-calling API (tool definitions via the native `tools` parameter, no system prompt) and score outputs with our AST framework. Per-dataset stratified 80/10/10 splits yield a 12,267-example validation set and a 12,282-example held-out test set; we select the best seed on validation and report final numbers on test. We fine-tune DistilBERT-base-uncased [26] (66M parameters) as a multi-label classifier with eight output heads using the input representation in Section 3.1, with 20 random seeds (hyperparameters in Appendix G; fine-tuning curves in Appendix H). We compare against (i) **single-model** routers (eight baselines); (ii) **heuristic** routers using a single feature (input token length, number of tool definitions, or conversation turns) with thresholds profiled on training data; and (iii) an **oracle** that selects the cheapest correct model per query (most expensive when none is correct).

Table 3: Accuracy and average cost per query on the held-out test set (12,282 examples). In/Out columns are list prices. For our routers we report best-seed accuracy over 20 random seeds (±std); per-seed details in Appendix I.

Type	Entity	In ($/M)	Out ($/M)	Accuracy (%)	Avg Cost (10^{-4} $)
Single LLM	GPT-5.3-chat	1.75	14.00	82.29	43.1
	GPT-5.4	2.50	15.00	81.26	64.2
	GPT-5.4-mini	0.75	4.50	80.25	21.1
	GPT-5-nano	0.05	0.40	79.15	1.9
	GPT-5.4-nano	0.20	1.25	78.00	5.4
	GPT-5-mini	0.25	2.00	77.33	8.6
	Qwen-3.5-9B	0.05	0.15	72.40	2.1
	Kimi-K2.5	0.60	3.00	60.88	8.2
Heuristic Router	Num. Turns		—	80.41	41.2
	Length		—	78.75	6.0
	Num. Tools		—	75.63	53.6
Ours	ModernBERT (149M)		—	83.02 (±0.38)	6.1
	DistilBERT (66M)		—	**82.94 (±0.41)**	**6.8**
	DeBERTa-v3 (86M)		—	82.89 (±0.41)	6.1
Upper Bound	Oracle		—	89.39	9.6

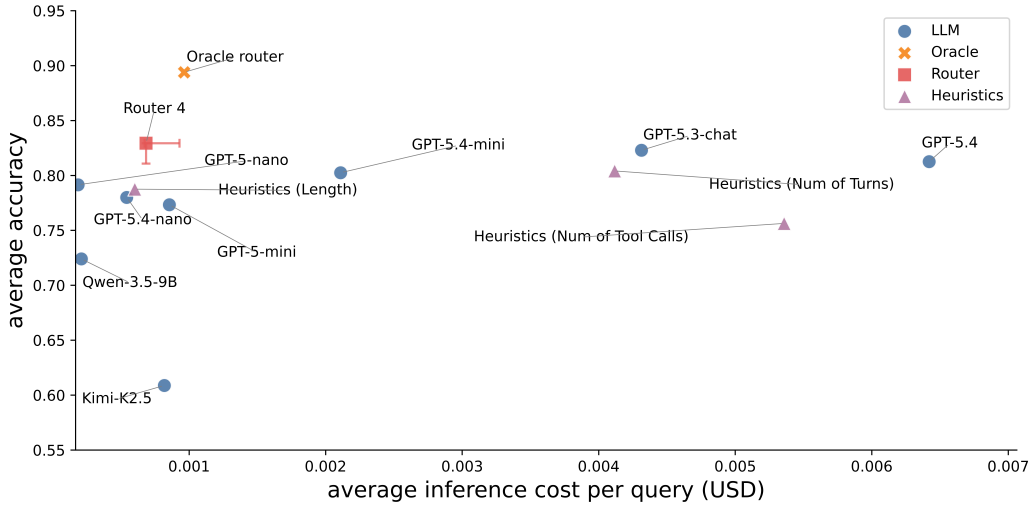

Figure 3: Accuracy–cost Pareto plot on the held-out test set (12,282 examples). Switchcraft (red square, with seed-range error bars) lies on the Pareto frontier between the cheapest single models and the oracle upper bound.

4.3 Main results: accuracy–cost trade-offs

Table 3 and Figure 3 present the headline results. The cost column reports the actual API-billed dollar cost per query; a token-level decomposition of these costs and an analysis of model chattiness are given in Section 4.8.

Switchcraft occupies a region of the Pareto frontier no single model reaches. Switchcraft achieves **82.94%** accuracy at 6.8×10^{-4} **$** per query—matching the best individual model (GPT-5.3-chat, 82.29%) while reducing cost by **84%**. The two are within seed variance; the salient claim is that *no individual model in the pool simultaneously achieves ≥80% accuracy and cost* $\leq 10\times10^{-4}$ *$*. GPT-5.3-chat reaches the accuracy at 6× the cost, while GPT-5-nano reaches the cost ceiling at 79.15% accuracy. Switchcraft uniquely occupies this region, saving approximately $3,630 per million queries over GPT-5.3-chat at matched accuracy. The threshold can be tuned to other Pareto points

(Appendix J). ModernBERT and DeBERTa-v3 are accuracy-equivalent within seed variance (± 0.41 and ± 0.38 pp); we prefer DistilBERT for its smaller footprint.

Gap to the oracle. The oracle's 89.39% at 9.6×10^{-4} \$ upper-bounds any router; Switchcraft closes $(82.94 - 79.15)/(89.39 - 79.15) = 37\%$ of the gap from the cheapest single model. A per-dataset breakdown (Appendix F) shows the advantage is largest on Glaive and xLAM-60K. We analyze the gap to the oracle in Section 4.7.

Heuristic routers are limited. The number-of-turns heuristic performs best (80.41%), still 2.53 pp below Switchcraft; input length and number of tools score 78.75% and 75.63%—no single surface-level feature captures the complexity of agentic routing.

4.4 Key findings

Table 3 reveals two counterintuitive inversions.

Finding 1: costlier is not always better. GPT-5.3-chat (82.29%) outperforms the newer, more expensive GPT-5.4 (81.26%): GPT-5.4 over-elaborates, producing verbose chain-of-thought that occasionally corrupts the structured tool-call output. Similarly, GPT-5-nano (79.15%) outperforms GPT-5-mini (77.33%): the nano model follows tool-calling instructions more directly, while mini paraphrases arguments or adds explanations. Blindly routing to the newest or most expensive model—a common enterprise default—can therefore decrease both accuracy and cost-efficiency.

Finding 2: open-weight models lag behind on tool calling. The two worst-performing models—Kimi-K2.5 (60.88%) and Qwen-3.5-9B (72.40%)—are both open-weight. Their dominant failure modes are: (i) **format violations** (markdown wrappers, preamble text, malformed JSON); (ii) **argument hallucination** (parameter values invented from training data rather than provided context); and (iii) **refusal to call tools**, returning a textual answer instead. These modes are largely absent from the proprietary instruction-tuned models, which were fine-tuned for structured output.

4.5 Router inference latency

A practical router must add minimal latency to LLM dispatch. Table 4 shows single-query P99 latency on a commodity NVIDIA T4 GPU (~\$0.35/hr spot): DistilBERT adds only 3–17 ms depending on sequence length—over an order of magnitude below typical LLM generation latency—and reaches 722 queries/sec peak throughput. ModernBERT is 2.7–5.6× slower with no meaningful accuracy gain. More details are in Appendix L.

Table 4: Single-query router latency (batch = 1) on NVIDIA T4, FP32.

Seq length	DistilBERT P99	ModernBERT P99	Ratio
64 tokens	3.2 ms	17.8 ms	5.6×
200 tokens	7.4 ms	34.6 ms	4.7×
512 tokens	17.1 ms	46.1 ms	2.7×

4.6 Robustness and ablations

Seed stability and encoder choice. Across 20 random seeds, DistilBERT router accuracy varies by only 1.86 pp (81.08%–82.94%, mean 81.89 ± 0.41 pp). Even the worst seed beats six of eight individual LLMs and all heuristic baselines. ModernBERT-base (149M, 8K context) and DeBERTa-v3-base (86M) achieve best-seed accuracy within 0.13 pp of DistilBERT (Table 3) with comparable seed stability (± 0.38, ± 0.41 pp). We prefer DistilBERT for its smaller footprint (Appendix I).

Token packing ablation. Our input representation uses intelligent token packing (Section 3.1) to fit decision-relevant content into the encoder's 512-token window. Table 5 isolates its contribution: replacing packing with naive right-truncation drops accuracy by 1.66 pp and increases routing cost by 21% (full details in Appendix K).

Table 5: Token packing ablation (DistilBERT, seed 4, test set).

Input strategy	**Accuracy (%)**	**Avg cost (10^{-4} \$)**
Token packing (ours)	82.94	6.8
Naive truncation	81.28	8.2
Δ	-1.66 pp	$+21\%$

Table 6: Routing outcomes on the validation set (12,267 examples, best-seed DistilBERT router).

Outcome	**Count**	**%**	**Meaning**
Correct–optimal	5,018	40.9	Cheapest correct model selected
Correct–suboptimal	5,156	42.0	A correct but costlier model selected
Wrong model	902	7.4	A correct model exists, but router picks one that fails
No correct model	1,191	9.7	Every model in the pool fails

4.7 Error analysis

We classify every validation query into four mutually exclusive **routing outcomes** (Table 6). On the validation set (12,267 examples), Switchcraft selects a correct model for **82.94%** of queries; only **7.4%** are avoidable mistakes (*wrong model*) where a different prediction would have succeeded; the remaining 9.7% are *irreducible* (no model in the pool answers correctly). The implied oracle gap on the validation set (7.4 pp) is slightly larger than the 6.45 pp gap reported on the test set in Table 3; both are dominated by the same "wrong model" failure mode and the difference reflects split composition, not a metric inconsistency.

Among correctly routed queries, 50.7% go to a non-cheapest model, but the practical impact is small (median overhead 0.18×10^{-4} \$, total \$2.19 across the validation set); 91% of suboptimal cases route to GPT-5-nano where the oracle would have chosen Qwen. Two robust difficulty patterns emerge (Appendix M): fewer correct models in the pool drives error rates up (67.9% misrouted with 1 correct model vs. 0% with 8), and more tool definitions hurt (15.3% at 1 tool vs. 39.8% at 11–50 tools), likely because complex schemas exceed the 512-token context window. This points to an opportunity for future improvement via loss shaping, probabilistic correctness, and richer embeddings.

4.8 Chattiness: when cheaper models become more expensive

A recurring theme is that **per-token pricing is a misleading proxy for per-query cost**. We define **chattiness** as the ratio of actual per-query cost to an *expected cost* assuming each model consumes the cross-model average number of tokens (above $1.0\times$ = more expensive than expected). Table 7 summarizes per-model results; full derivation in Appendix N. Kimi-K2.5 is the chattiest ($1.31\times$, output dominates 69% of its cost). Qwen-3.5-9B generates the most tokens (228 avg) yet its chattiness is only $1.10\times$ because input cost dominates (81%)—the large output surplus barely moves the ratio. GPT-5.3-chat is most concise ($0.78\times$, 47 avg tokens), making it the best accuracy–cost trade-off among single models despite its high list price. A router selecting on list price would systematically misjudge verbose budget models and concise mid-tier ones; using profiled per-query cost (Section 3.3) avoids this trap.

Table 7: Per-model chattiness on the validation set.

Model	**Chattiness**	**Pattern**
Kimi-K2.5	$1.31\times$	Verbose budget
GPT-5-nano	$1.13\times$	Hidden reasoning cost
Qwen-3.5-9B	$1.10\times$	Verbose but cheap rate
GPT-5-mini	$1.00\times$	Near average
GPT-5.4	$0.91\times$	Concise
GPT-5.4-mini	$0.90\times$	Concise
GPT-5.4-nano	$0.88\times$	Concise
GPT-5.3-chat	$0.78\times$	Most concise

4.9 Comparison to chat router and other model pools

With an earlier eight-model basket (Appendix O), the same pattern holds: Switchcraft achieves 82.73% accuracy at 88% lower cost than the best individual model, while a chat-fine-tuned variant of the same architecture (trained on 64 public chat benchmarks) reaches only 77.47% (5.26 pp below Switchcraft)—routing decisions learned from chat completions do not transfer to agentic tool calling.

5 Limitations

Oracle gap. Switchcraft's 6.45 pp gap to the oracle is the primary improvement opportunity. Promising directions include probabilistic correctness modeling (Appendix P), asymmetric loss shaping, and richer input representations that capture more context within the token budget.

In-distribution evaluation and contamination. Our test set is drawn from the same distribution as training (stratified 80/10/10 splits per benchmark)—a *known agentic workload mix* typical of enterprise settings with representative production logs available for fine-tuning. All public datasets predate the 2025-08-31 GPT-5.3/5.4 training cutoff, so we cannot rule out memorization inflating single-model accuracies. However, Switchcraft learns from *realized model behavior*, so its advantage holds regardless. Quantifying degradation under benchmark shift remains future work.

Generalization to new models. Switchcraft is fine-tuned for a fixed pool of eight models and requires retraining when models are added or updated. We validate that the same architecture generalises across baskets (Appendix O). We evaluate cold-start approaches (MIRT: Appendix Q), but combining those with agentic specialization is future work.

Additional considerations—per-turn vs. end-to-end success, cost-model assumptions, prompt caching, reasoning effort—are in Appendix R.

6 Related work

Tool calling extends LLMs with access to external data and computation [4], and prior work has proposed methods that teach individual LLMs to invoke tools [23, 27]. Scoring those calls, however, remains the bottleneck. Prior work either execute each call [36], or match its abstract syntax tree against a reference using BFCL's checker [24, 3]. The former is infeasible at our scale as it requires implementing all the APIs, and the latter, as we show in §3.2, is significantly biased on every non-BFCL benchmark we test. Prior work also focus on synthetic data generation for tool calls [5].

Model routing reduces inference cost by dispatching each query to the cheapest model in a pool that can answer it with a reasonable quality. A growing body of work learns this routing policy from per-query quality signals, using neural networks [7, 25, 6, 2], k-nearest neighbors [12, 28, 31], matrix factorization [22, 38], graph neural networks [10], k-means clustering [14], item response theory [30], or constrained optimization [18]. Complementary lines target cost estimation directly [29, 19], enrich the action space with token-budget control [34] or best-of-n sampling [8], learn more expressive query representations [33], or improve explainability [21]. Two recent benchmarks also target model routing [16, 17]. None of these works, however, targets tool calling or treats correctness as a first-class objective. As we argue in §2, assuming partial responses are acceptable does not work well in tool calling workloads, where a single incorrect argument can have consequences.

We bridge these two threads with Switchcraft—to the best of our knowledge, the first model router explicitly designed and evaluated for agentic tool calling: fine-tuned on agentic benchmarks and optimized for strict, AST-level correctness. A separate line of work attacks agent inference cost through orthogonal mechanisms [37, 32] which compose with model routing rather than replacing it.

7 Conclusions

We presented Switchcraft, an agentic model router using a lightweight DistilBERT classifier (66M parameters) to route tool-calling queries among eight candidates, matching the best individual model's accuracy (82.9% vs. 82.3%) at 84% lower cost ($3,600+ saved per million queries). Key enablers are

agentic training data with an improved AST framework, a chattiness-aware cost model, and sub-20 ms inference. As available LLMs proliferate and their cost–capability trade-offs diversify, intelligent routing becomes critical for scalable AI deployment. Code generation and multi-step planning are natural future targets.

8 Acknowledgments

We are grateful to Vijay Aski, Rupeshkumar Mehta, Sethu Raman and Steve Sweetman for supporting and enabling deep collaboration between Microsoft Research and Microsoft Foundry on this effort.

References

[1] Mohamad Abou Ali, Fadi Dornaika, and Jinan Charafeddine. Agentic AI: A Comprehensive Survey of Architectures, Applications, and Future Directions. *Artificial Intelligence Review*, 59(11), 2025. doi: 10.1007/s10462-025-11422-4. URL https://link.springer.com/article/10.1007/s10462-025-11422-4.

[2] Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam . Automix: Automatically mixing language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[3] Tamer Alkhouli, Katerina Margatina, James Gung, Raphael Shu, Claudia Zaghi, Monica Sunkara, and Yi Zhang. CONFETTI: Conversational Function-Calling Evaluation Through Turn-Level Interactions. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7993–8006, Vienna, Austria, jul 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.394. URL https://aclanthology.org/2025.acl-long.394/. Creative Commons Attribution 4.0 International License.

[4] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Validation of modern json schema: Formalization and complexity. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi: 10.1145/3632891. URL https://doi.org/10.1145/3632891.

[5] Vibha Belavadi, Tushar Vatsa, Dewang Sultania, Suhas Suresha, Ishita Verma, Cheng Chen, Tracy Holloway King, and Michael Friedrich. Routenator: A router-based multi-modal architecture for generating synthetic training data for function calling llms, 2025. URL https://arxiv.org/abs/2505.10495.

[6] Shuhao Chen, Weisen Jiang, Baijiong Lin, James Kwok, and Yu Zhang. Routerdc: Query-based router by dual contrastive learning for assembling large language models. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-2120. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/7a641b8ec86162fc875fb9f6456a542f-Paper-Conference.pdf.

[7] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *The Twelfth International Conference on Learning Representations*, 2024.

[8] Dujian Ding, Ankur Mallick, Shaokun Zhang, Chi Wang, Daniel Madrigal, Mirian Del Carmen Hipolito Garcia, Menglin Xia, Laks V. S. Lakshmanan, Qingyun Wu, and Victor Rühle. BEST-route: Adaptive LLM routing with test-time optimal compute. In *Forty-second International Conference on Machine Learning*, 2025.

[9] Aosong Feng, Balasubramaniam Srinivasan, Yun Zhou, Zhichao Xu, Kang Zhou, Sheng Guan, Yueyan Chen, Xian Wu, Ninad Kulkarni, Yi Zhang, Zhengyuan Shen, Dmitriy Bespalov, Soumya Smruti Mishra, Yifei Teng, Darren Yow-Bang Wang, Haibo Ding, and Lin Lee Cheong. IPR: Intelligent Prompt Routing with User-Controlled Quality-Cost Trade-offs. *arXiv preprint arXiv:2509.06274*, 2025. URL https://arxiv.org/abs/2509.06274.

[10] Tao Feng, Yanzhen Shen, and Jiaxuan You. Graphrouter: A graph-based router for llm selections, 2025. URL https://arxiv.org/abs/2410.03834.

[11] Glaive AI. Glaive Function Calling v2. Dataset available from HuggingFace, 2023. URL https://huggingface.co/datasets/glaiveai/glaive-function-calling-v2. Apache 2.0 License.

[12] Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. Routerbench: A benchmark for multi-LLM routing system. In *Agentic Markets Workshop at ICML 2024*, 2024.

[13] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770*, 2024.

[14] Wittawat Jitkrittum, Harikrishna Narasimhan, Ankit Singh Rawat, Jeevesh Juneja, Congchao Wang, Zifeng Wang, Alec Go, Chen-Yu Lee, Pradeep Shenoy, Rina Panigrahy, Aditya Krishna Menon, and Sanjiv Kumar. Universal Model Routing for Efficient LLM Inference. *arXiv preprint arXiv:2502.08773*, 2025.

[15] Pedro Las-Casas, Alok Gautum Kumbhare, Rodrigo Fonseca, and Sharad Agarwal. LLexus: An AI Agent System for Incident Management. *ACM SIGOPS Operating Systems Review*, 58 (1):23–36, 2024. doi: 10.1145/3689051.3689056.

[16] Hao Li, Yiqun Zhang, Zhaoyan Guo, Chenxu Wang, Shengji Tang, Qiaosheng Zhang, Yang Chen, Biqing Qi, Peng Ye, Lei Bai, Zhen Wang, and Shuyue Hu. LLMRouterBench: A Massive Benchmark and Unified Framework for LLM Routing. *arXiv preprint arXiv:2601.07206*, 2026.

[17] Yifan Lu, Rixin Liu, Jiayi Yuan, Xingqi Cui, Shenrun Zhang, Hongyi Liu, and Jiarong Xing. RouterArena: An Open Platform for Comprehensive Comparison of LLM Routers. *arXiv preprint arXiv:2510.00202*, 2025.

[18] Kai Mei, Wujiang Xu, Minghao Guo, Shuhang Lin, and Yongfeng Zhang. OmniRouter: Budget and Performance Controllable Multi-LLM Routing. *arXiv preprint arXiv:2502.20576*, 2025.

[19] Quang H. Nguyen, Thinh Dao, Duy C. Hoang, Juliette Decugis, Saurav Manchanda, Nitesh V. Chawla, and Khoa D. Doan. Metallm: A high-performant and cost-efficient dynamic framework for wrapping llms, 2025. URL https://arxiv.org/abs/2407.10834.

[20] Nous Research. Hermes 3 Technical Report. Technical report, Nous Research, 2024. URL https://nousresearch.com/wp-content/uploads/2024/08/Hermes-3-Technical-Report.pdf. Apache 2.0 License.

[21] Mika Okamoto, Ansel Kaplan Erol, and Mark Riedl. Explainable model routing for agentic workflows, 2026. URL https://arxiv.org/abs/2604.03527.

[22] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs from preference data. In *The Thirteenth International Conference on Learning Representations*, 2025.

[23] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems*, volume 37. Curran Associates, Inc., 2024. doi: 10.52202/079017-4020.

[24] Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, pages 48371–48392, 2025. Apache 2.0 License.

[25] Marija Sakota, Maxime Peyrard, and Robert West. Fly-swat or cannon? cost-effective language model choice via meta-modeling. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*, WSDM '24, page 606–615, 2024. URL https://doi.org/10.1145/3616855.3635825.

[26] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *arXiv preprint arXiv:1910.01108*, 2019. URL https://arxiv.org/abs/1910.01108.

[27] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf.

[28] Tal Shnitzer, Anthony Ou, Mírian Silva, Kate Soule, Yuekai Sun, Justin Solomon, Neil Thompson, and Mikhail Yurochkin. Large language model routing with benchmark datasets, 2023. URL https://arxiv.org/abs/2309.15789.

[29] Seamus Somerstep, Felipe Maia Polo, Allysson Flavio Melo de Oliveira, Prattyush Mangal, Mírian Silva, Onkar Bhardwaj, Mikhail Yurochkin, and Subha Maity. Carrot: A cost aware rate optimal router, 2025. URL https://arxiv.org/abs/2502.03261.

[30] Wei Song, Zhenya Huang, Cheng Cheng, Weibo Gao, Bihan Xu, GuanHao Zhao, Fei Wang, and Runze Wu. IRT-Router: Effective and Interpretable Multi-LLM Routing via Item Response Theory. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025.

[31] Dimitris Stripelis, Zijian Hu, Jipeng Zhang, Zhaozhuo Xu, Alay Dilipbhai Shah, Han Jin, Yuhang Yao, Salman Avestimehr, and Chaoyang He. Tensoropera router: A multi-model router for efficient llm inference, 2024. URL https://arxiv.org/abs/2408.12320.

[32] vLLM Semantic Router Team. vllm semantic router. https://github.com/vllm-project/semantic-router, 2025.

[33] Chenxu Wang, Hao Li, Yiqun Zhang, Linyao Chen, Jianhao Chen, Ping Jian, Peng Ye, Qiaosheng Zhang, and Shuyue Hu. Icl-router: In-context learned model representations for llm routing. In *AAAI Conference on Artificial Intelligence*, 2025. URL https://api.semanticscholar.org/CorpusID:282057625.

[34] Jiaqi Xue, Qian Lou, Jiarong Xing, and Heng Huang. R2-router: A new paradigm for llm routing with reasoning. *arXiv preprint arXiv:2602.02823*, 2026.

[35] Shunyu Yao, Jian Pei, Yue Ma, and Howard Chen. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. *arXiv preprint arXiv:2406.12045*, 2024.

[36] Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, Zhiwei Liu, Yihao Feng, Tulika Awalgaonkar, Rithesh Murthy, Eric Hu, Zeyuan Chen, Ran Xu, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. xLAM: A Family of Large Action Models to Empower AI Agent Systems. *arXiv preprint arXiv:2409.03215*, 2024. URL https://arxiv.org/abs/2409.03215. Creative Commons Attribution 4.0 International License.

[37] Qizheng Zhang, Michael Wornow, and Kunle Olukotun. Cost-Efficient Serving of LLM Agents via Test-Time Plan Caching. In *International Conference on Machine Learning Workshops*, 2025.

[38] Richard Zhuang, Tianhao Wu, Zhaojin Wen, Andrew Li, Jiantao Jiao, and Kannan Ramchandran. Embedllm: Learning compact representations of large language models, 2024. URL https://arxiv.org/abs/2410.02223.

Available tool: `calculate_sales_tax(purchase_amount, city, state)`

User: "Calculate the amount of sales tax to be added on a purchase amount of $30.45 in Chicago, Illinois, $52.33 in Sacramento, California and $11.23 in Portland, Oregon."

`Response A` ✓ (three parallel calls):

```
1. calculate_sales_tax(purchase_amount=30.45, city="Chicago",    state="Illinois")
2. calculate_sales_tax(purchase_amount=52.33, city="Sacramento", state="California")
3. calculate_sales_tax(purchase_amount=11.23, city="Portland",   state="Oregon")
```

`Response B` ✓ (also correct—reordered calls with abbreviated parameter values):

```
1. calculate_sales_tax(purchase_amount=11.23, city="Portland",   state="OR")
2. calculate_sales_tax(purchase_amount=30.45, city="CHI",      state="IL")
3. calculate_sales_tax(purchase_amount=52.33, city="Sacramento", state="CA")
```

Both responses are correct. Because these three calls are *independent* (no data flows between them), any ordering is valid. Additionally, semantically equivalent parameter values—"Illinois" vs. "IL", "Chicago" vs. "CHI"—are equally acceptable.

Figure 4: Acceptable variation in an agentic parallel tool-calling scenario (BFCL v3 [24]). The user's request requires three independent `calculate_sales_tax` calls. Response A and Response B differ in (i) the *order* of the parallel calls and (ii) the surface form of string parameters (e.g., "Illinois" vs. "IL"), yet both are semantically correct. An agentic evaluation framework must recognise such benign variation rather than penalising it as error.

A Acceptable variations in tool calls

While agentic tool calls demand strict precision, not all variation constitutes an error. Consider the sales-tax scenario in Figure 4, where the user asks an agent to compute sales tax for purchases in three cities. The agent must issue three independent calls to `calculate_sales_tax`, but because no call depends on the output of another, any permutation of the three calls is equally valid. Furthermore, string parameters admit semantically equivalent surface forms: `"Illinois"` and `"IL"` refer to the same state, as do `"Chicago"` and `"CHI"`. A correct evaluation must treat both orderings and both spellings as acceptable, rather than penalizing responses that happen to differ from a single canonical reference. These flexibility requirements directly motivate the AST-checker design choices in Section 3.2.

B Design space exploration

The architecture described in Section 3 emerged from systematic exploration of a substantially larger design space. Table 8 summarizes the principal alternatives we evaluated and the reasons we did not retain them in the final system.

This exploration consumed the majority of our effort and informed several key decisions: (i) a fine-tuned encoder outperforms frozen embeddings; (ii) latency constraints eliminate larger backbones and LLM-based routers; (iii) AST-based scoring is essential for reliable labels in the function-calling domain; and (iv) separating correctness prediction from cost-aware selection (two-stage routing) dominates single-objective alternatives such as cost-weighted losses or α/β trade-off parameters.

Table 8: Design alternatives explored during development. Each row is an approach we prototyped or carefully evaluated before discarding.

Alternative	Outcome and rationale for discarding
OpenAI embedding + MLP	Replaced the DistilBERT encoder with frozen OpenAI text embeddings fed to a small MLP (1–4M parameters). Accuracy was comparable on single-turn data but degraded on multi-turn benchmarks because the frozen embedding could not be fine-tuned to attend to conversation structure. Also requires an external API call at inference time, adding latency and a dependency.
Larger encoders (RoBERTa-base/large, BERT-large)	Marginal accuracy gains ($<$0.5 pp) over DistilBERT but doubled or tripled inference latency, violating our sub-5 ms latency budget for a production router.
LLM-as-a-router	Using a language model (e.g., GPT-4o-mini) to select the target model per query. Latency is 100–500$\times$ higher than an encoder classifier, cost scales linearly with traffic, and accuracy was not meaningfully better. Dismissed early.
RouteLLM-style similarity routing	Cosine-similarity-weighted scores using LMSys preference data. Informative for error analysis but did not improve over the fine-tuned classifier: plain cosine similarity could not separate confusable cases in our function-calling domain.
Ensemble / heuristic hybrids	Combining rule-based heuristics (e.g., tool count, conversation length) with learned classifiers. The classifier alone subsumed the heuristic signals via the metadata preamble (Section 3.1).
Cost-weighted loss	Asymmetric loss that penalizes misrouting to expensive models more heavily. Improved cost slightly but reduced accuracy; the two-stage predict-then-select architecture already separates correctness from cost.
BLEU / text-similarity scoring	Evaluated as an alternative to AST comparison for labeling correctness. Unsuitable: high BLEU scores can co-occur with functionally incorrect tool calls (e.g., wrong argument order), and low BLEU can accompany correct calls with cosmetic differences.
LLM-as-judge labeling	Used a strong LLM to label correctness instead of AST comparison. Expensive, non-deterministic, and surprisingly unreliable on structured output: the judge often marked calls as correct when parameter values were subtly wrong.
RL / contextual-bandit online routing	Learning routing policy from live agent success signals. Promising in principle but requires a deployed agent generating reward signal; incompatible with our offline benchmark evaluation setting. Identified as future work.
Distillation from oracle router	Training a fast student router to mimic oracle routing decisions. The oracle is already the implicit supervision signal in our multi-label formulation (correct/incorrect labels), so explicit distillation would provide no additional benefit.

C Profiled cost model

For every candidate model m, we accumulate its actual dollar cost across all training queries:

$$C_{\text{profiled}}(m) = \sum_{q \in \mathcal{D}_{\text{train}}} \text{cost}(q, m) \quad \text{where} \quad \text{cost}(q, m) = \frac{n_{\text{in}}(q) \cdot r_{\text{in}}(m) + n_{\text{out}}(q, m) \cdot r_{\text{out}}(m)}{10^6}.$$

n_{in} and n_{out} are the actual input and output token counts observed when model m answered query q; r_{in} and r_{out} are the per-million-token list prices from Table 3. For multi-turn conversations, costs are summed across all turns. This profiled cost captures *chattiness*: a model that generates extensive reasoning tokens or verbose explanations accumulates a higher profiled cost than a concise model at the same per-token rate.

D AST checker examples

Table 9 presents representative cases where the BFCL AST checker produces false negatives, along with our corresponding fixes.

Table 9: Representative cases where BFCL AST checker fails with our corresponding fix. Each row is a false-negative entry selected from parsing the GPT-5.3-chat responses. *out* shows the model output, and *GT* is the ground truth.

Limitation	**Failing example**	**Why BFCL fails**	**Our resolution**
Array order sensitivity	*out:* `search_books( keywords=["haunted house", "mystery"], genre="mystery")` *GT:* `keywords=["mystery", "haunted house"]`	BFCL compares the model's list to the ground-truth list with Python's == after exact-membership lookup, which is order-sensitive. The two lists contain the same two strings but in different order, so equality is false and the call is marked wrong.	Order-insensitive multiset comparison with per-element string normalization.
No default-parameter awareness	*out:* `top_players_by_matchmaking(limit=50)` *GT:* `limit=50, page=0` *schema:* `page` described as *"Default is 0"*	BFCL treats every parameter that appears in the ground truth as required. Because the model omitted `page`, BFCL reports it missing; it never reads the schema description to learn that the documented default (`0`) is exactly the expected ground-truth value.	Default value parsed from the schema description; an omission is accepted iff the documented default matches the GT.
Brittle string matching	*out:* `birthdate= "1990-05-15T00:00:00"` *GT:* `"1990-05-15T00:00:00Z"` *out:* `date="15 June", location="Office conference room"` *GT:* `date="15th June", location="office conference room"`	BFCL does only case-insensitive byte equality on strings. The trailing `Z` (UTC marker) makes the two ISO-8601 timestamps unequal even though they denote the identical instant; likewise, `"15 June"` vs. `"15th June"` differs only by an ordinal suffix and `"Office conference room"` differs only in capitalization. BFCL has no whitespace, punctuation, ISO-8601, or paraphrase normalization.	Canonicalize case, whitespace, punctuation, and ISO-8601 (`Z`, `+00:00`); a DistilBERT cosine ≥ 0.85 serves as a paraphrase fallback when GT contains spaces.
No nested-structure comparison	*out:* `sync_salesforce_data(authentication_details={ salesforce:{client_id, ...}, pega:{api_key, ...}})` *GT:* byte-identical nested object *schema:* `authentication_details:{type:"object", properties:{...}}`	BFCL maps each schema type string to a Python class through a hard-coded lookup table that has no entry for the JSON-Schema standard type `"object"` (and also omits `"number"`). The first time the checker meets such a parameter it raises `KeyError: 'object'` and aborts the entire entry, marking it wrong regardless of content. Even if the lookup succeeded, BFCL has no recursive comparator and could not have validated the nested sub-objects.	Add `"object":dict` (and `"number":int`) to the type table; recursively validate each nested key against the corresponding sub-schema.

Continued on next page

Table 9 continued from previous page

Limitation	Failing example	Why BFCL fails	Our resolution
Ambiguous array semantics	*out:* `find_common_elements(arrays=[[1,2,3,4,5],[2,3,4,6,7],[2,3,8,9,10]])` *GT:* byte-identical 2-D matrix *schema:* `arrays:{type:"array", items:{type:"array"}}`	BFCL ground truth wraps every value in a list, which makes a true list-of-lists indistinguishable in shape from a list of alternative flat answers. BFCL has no schema-aware branch: it always interprets the outer list as alternatives. Here the parameter's actual type is `array of arrays`, so BFCL silently re-reads the GT as *three alternative flat integer lists* and demands the model's output equal one of them. It can never match.	Consults `items.type`: alternatives only when the items are scalars; a true 2-D structure when items are themselves `array`.
List-of-dict handling and ordering	*out:* `calculate_gpa(grades=[{course:"Math", grade:"B"}, {course:"Science", grade:"A"}, ...])` *GT:* same dicts reordered *schema:* `grades:{type:"array", items:{type:"object",...}}`	BFCL has no entry for `"object"` in its type table, so any `array of object` parameter is dispatched into a code path that immediately raises `KeyError: 'object'` and aborts. Even if the crash were fixed, BFCL would still compare the two lists element-wise in order, so any reordering of the per-course dictionaries would also be rejected.	Each inner dict is normalized and the two lists are compared as multisets, so neither key order within a dict nor element order across the list affects the verdict.

E ConFETTI ground-truth corrections

The ConFETTI dataset [3] provides 506 multi-turn conversational function-calling entries that we adopted for use in our evaluation (Section 4). When manually examining results from preliminary experiments, we observed that some ground-truth labels contained errors—incorrect function calls or parameter values that do not match the conversation context. Because our routing evaluation scores every model against these labels, even a small number of incorrect labels can unfairly penalize correct model behavior and distort accuracy comparisons. We therefore conducted a systematic review and corrected the ground truth where necessary.

E.1 Correction methodology

We performed two review rounds using an LLM-assisted human-in-the-loop process. In each round, every entry was submitted to a verifier model (GPT-5) that received the full conversation history, the available function definitions, and the current ground-truth label. The model assessed whether the ground truth correctly represents the next function call given the conversational context and flagged entries it considered incorrect, along with a suggested correction and explanation.

Entries flagged by the model were then presented to a human reviewer in an interactive terminal interface. For each flagged entry the reviewer could *accept* the correction (updating the ground-truth file), *decline* it (keeping the original label), or *skip* it so that the human can manually edit that entry, typically because the suggested correction was not quite correct or the input needed correction rather than the ground truth. Entries the model deemed correct were automatically marked as reviewed and not surfaced for human inspection. A second round re-examined entries that were skipped and manually edited in round 1 and served as a consistency check.

Across two rounds, a total of **69 ground-truth corrections** (13.6% of the dataset) were made. Combined with the 86 input disambiguations described below, 137 of 506 entries (27%) were modified in total. We will open-source all corrections as a PR against the ConFETTI repository, and separately open-source the LLM-assisted correction tool used to produce them.

E.2 Error categories

Table 10 classifies the 69 corrections into six categories. Each correction was assigned a single primary category; when multiple issues co-occurred, we categorised by the most impactful error (e.g., calling the wrong function takes precedence over a casing mismatch in one of its parameters). These categories matter for function-calling evaluation because scoring typically relies on exact or near-exact matching of function names and parameter values, so even minor mismatches in the ground truth can cause correct model outputs to be marked as failures.

Table 10: Categories of ground-truth errors corrected in the ConFETTI dataset (69 corrections out of 506 entries).

Category	Count	Description	Example
Wrong function	14	Ground truth invokes a function that does not match the user's intent or the available conversational context.	User asks what their insurance *covers*; ground truth calls `get_policy` (returns general policy info) instead of `get_available_coverage`.
Incorrect parameter value	21	Correct function, but a structured parameter value is wrong—e.g., wrong entity ID, swapped coordinates, or incorrect reference from prior tool results.	User asks to check their *savings* account balance; ground truth passes the *checking* account number.
Incorrect content	7	Free-text content in the function call does not match what the user actually said.	User says "white elephant RSVP"; ground truth sets the email subject to "Secret Santa RSVP."
Format / schema error	16	Value violates the function's schema constraints: wrong enum casing, incorrect data type, invalid date format, trailing whitespace, or invalid enum value.	Function expects date as `YYYY-MM-DD`; ground truth passes `"2024-04-09T24:00:00Z"` (invalid date-time with hour 24).
Temporal error	5	Date or time value is incorrect or unresolvable from the conversation—e.g., wrong UTC offset, unspecified year defaulting to an arbitrary past date.	User requests a 4:20 PM Eastern appointment; ground truth books `09:20 UTC` instead of `20:20 UTC`.
Logical / state-tracking error	6	Ground truth reflects flawed reasoning about the conversation state or skips a prerequisite step.	Assistant computed distances to JFK and EWR but not LGA; ground truth searches flights to LGA without first computing its distance.

E.3 Input disambiguation

In addition to the 69 ground-truth corrections, we modified the *conversation input* of 86 entries (17% of the dataset) to resolve temporal ambiguity. Many ConFETTI conversations reference dates without specifying a year (e.g., "fly on the 25th of December" or "check hotel availability for June"). The ground-truth function calls in these entries use specific dates with hardcoded years (e.g., `2023-12-25`), yet the conversation text provides no basis for inferring which year was intended. This creates an unfair evaluation setup: a model that produces the correct function call with a different year—or that reasonably asks for clarification—would be scored as incorrect.

To resolve this, we appended the year (or month and year where needed) to the user utterance so that the conversation unambiguously specifies the date referenced by the ground-truth label. For example, "fly on the 25th of December" becomes "fly on the 25th of December 2023." Three additional entries received non-temporal disambiguation (e.g., adding a location qualifier when the ground truth assumed a specific region). These input changes do not alter the ground-truth labels; they ensure that the information needed to produce the expected function call is present in the conversation.

In total, 137 of 506 entries (27%) received either an input disambiguation, a ground-truth correction, or both (18 entries required both).

F Dataset details and per-dataset breakdown

This appendix provides detailed descriptions and cleaning procedures for each dataset, followed by a per-dataset accuracy and cost breakdown.

F.1 Dataset descriptions

BFCL v3. The Berkeley Function Calling Leaderboard v3 [24] is the de facto standard for evaluating tool-calling capabilities. We use all single-turn categories (`simple`, `multiple`, `parallel`, `parallel_multiple`) and their "live" counterparts, which test against real-world API signatures. For the multi-turn categories (`multi_turn_base`, `multi_turn_long_context`), we decompose each multi-turn conversation into per-turn evaluation records using a custom script that replays ground-truth function executions to build the conversation history progressively. This yields 745 per-turn examples per category, providing fine-grained multi-turn evaluation.

ConFETTI. ConFETTI [3] provides 109 human-simulated multi-turn conversations spanning 86 unique APIs, totaling 506 turns after decomposition. Unlike BFCL's synthetic multi-turn data, ConFETTI conversations are authored by human annotators who simulate realistic dialogue flows, including clarifications, corrections, and multi-step task completions. We apply a format normalization step that restructures conversation turns into the nested list format expected by our evaluation framework. The ConFETTI dataset required substantial data cleaning: we disambiguated 86 conversation inputs (17%) where temporal references lacked a year, and corrected 69 ground-truth annotations (13.6%) where the expected function call was incorrect or incomplete—137 entries (27%) in total (Appendix E). We frame these corrections as *benchmark infrastructure*, not a methodological contribution: we will open-source the corrected ConFETTI annotations, the LLM-assisted correction tool used to produce them, and our improved AST checker, so that any third party can reproduce, audit, or reject individual edits without re-deriving the corrections.

Glaive Function Calling v2. The Glaive Function Calling v2 dataset [11] contributes the largest number of examples (83,814) and covers multi-turn function-calling conversations with diverse tool definitions. We convert the dataset to BFCL format and apply two cleaning steps: (i) replacing `None` placeholder values in ground-truth arguments with appropriate defaults (e.g., empty objects for functions like `get_current_time` that take no arguments), and (ii) removing empty-string optional parameters from ground-truth annotations that would cause spurious evaluation failures.

xLAM-60K. The Salesforce xLAM Function Calling 60K dataset [36] provides 60,000 single-turn function-calling examples. Each example consists of a user query, a set of tool definitions, and ground-truth function calls. This dataset contributes scale and diversity of tool schemas, serving as the bulk of our single-turn training data.

Hermes Function Calling v1. The Hermes Function Calling v1 dataset [20] from Nous Research contributes 8,940 examples drawn from three subsets: `func_calling_singleturn`, `func_calling`, and `glaive_func_calling`. This provides a mix of single-turn and multi-turn tool-calling scenarios, including both tool-call prediction (selecting the right function and arguments) and tool-response understanding (interpreting the result of a prior tool call).

F.2 Per-dataset accuracy breakdown

Table 11 breaks down accuracy by dataset group on the seed-4 validation split (12,267 examples).

Observations.

- **The router outperforms every single model on Glaive and Hermes** (85.7% and 88.9%), the two largest dataset groups. On Glaive, it exceeds the best single model (GPT-5.3-chat, 84.0%) by 1.7 percentage points, confirming that routing can improve accuracy, not just reduce cost.
- **Multi-turn (MT) is the hardest category for all models.** Even the best single model (GPT-5.4, 73.0%) falls far short of the oracle (90.5%), leaving a 17.5 pp gap. The router

Table 11: Per-dataset-group accuracy (%) on the validation set. Dataset groups aggregate the sources in Table 2: *Single* = BFCL single-turn (simple, multiple, parallel, parallel_multiple), *Live* = BFCL live variants, *MT* = BFCL multi-turn (base + long context), *ConF.* = ConFETTI. Sample sizes (n) shown below each column header. Models sorted by overall accuracy; best single-model result per column in **bold**.

	Single	**Live**	**MT**	**ConF.**	**Glaive**	**Hermes**	**xLAM**	**All**
n	100	133	148	50	5093	782	5961	12267
GPT-5.3-chat	**94.0**	**83.5**	67.6	**56.0**	**84.0**	87.7	**83.1**	83.6
GPT-5.4	**94.0**	82.0	**73.0**	34.0	**84.0**	**89.4**	80.1	82.2
GPT-5.4-mini	92.0	79.7	60.8	40.0	83.4	88.2	78.7	81.0
GPT-5-nano	89.0	81.2	55.4	34.0	82.9	87.1	77.4	80.0
GPT-5.4-nano	**94.0**	77.4	60.8	32.0	80.2	86.7	77.2	78.8
GPT-5-mini	92.0	70.7	62.8	36.0	79.3	83.6	77.9	78.5
Qwen-3.5-9B	89.0	76.7	41.2	50.0	81.5	85.2	66.1	73.7
Kimi-K2.5	82.0	70.7	61.5	32.0	82.5	79.8	40.6	61.4
DistilBERT	90.0	81.2	61.5	44.0	85.7	88.9	80.6	82.9
Oracle	99.0	91.7	90.5	74.0	90.4	95.4	89.6	90.3

(61.5%) does not close this gap, matching the performance of lower-tier models. This suggests that multi-turn context is difficult for the router to capture within its 512-token window.

- **ConFETTI is challenging** due to its small size (50 queries) and conversational nature. The oracle itself reaches only 74.0%, meaning 26% of ConFETTI queries defeat all eight models. The router's 44.0% is between the best (GPT-5.3-chat, 56.0%) and the median single model.
- **xLAM-60K shows the largest absolute model spread**: Kimi-K2.5 (40.6%) vs. GPT-5.3-chat (83.1%), a 42.5 pp gap. The router achieves 80.6%, close to the best model, indicating effective routing on this large synthetic dataset.
- **BFCL-Single** queries are relatively easy (82–94% across all models), so routing adds less value. The router achieves 90.0%, near the oracle's 99.0%.

Table 12 shows the corresponding per-query cost breakdown. Costs are reported in milli-dollars ($\times 10^3$).

Table 12: Per-dataset-group average cost per query on the validation set (12,267 examples), in $\times 10^3$ \$ (milli-dollars). Multi-turn queries are 100–200$\times$ more expensive than single-turn due to extended conversation context. Cheapest single-model result per column in **bold**. Costs here are computed as the per-query average of API-billed dollar amounts (sum of per-query costs divided by number of queries with valid token-count records); they may differ slightly from the cost-decomposition view in Table 28, which computes $\bar{t}_{\text{in}} \cdot r_{\text{in}} + \bar{t}_{\text{out}} \cdot r_{\text{out}}$ from rounded average token counts and so does not preserve per-query variance.

	Single	**Live**	**MT**	**ConF.**	**Glaive**	**Hermes**	**xLAM**	**All**
GPT-5.3-chat	1.37	1.74	280.8	4.42	0.80	1.09	1.35	4.23
GPT-5.4	1.96	2.74	455.9	7.20	1.11	1.52	2.02	6.75
GPT-5.4-mini	0.56	0.80	119.4	2.13	0.33	0.46	0.57	1.79
GPT-5-nano	0.08	0.09	9.2	0.19	**0.05**	0.06	0.09	0.17
GPT-5.4-nano	0.14	0.21	37.8	0.54	0.09	0.12	0.15	0.54
GPT-5-mini	0.32	0.35	51.1	0.87	0.19	0.23	0.32	0.82
Qwen-3.5-9B	**0.07**	**0.09**	**14.7**	**0.20**	**0.04**	**0.05**	**0.06**	**0.21**
Kimi-K2.5	0.92	1.07	15.0	2.67	0.42	0.63	0.83	0.82
DistilBERT	0.11	0.25	41.5	3.11	0.14	0.14	0.28	0.68
Oracle	0.06	0.15	13.4	0.81	0.04	0.06	0.10	0.23

Cost observations.

- **Multi-turn queries dominate cost.** MT queries cost 100–200$\times$ more than single-turn queries for the same model (e.g. GPT-5.4: \$0.456 vs. \$0.002 per query) due to extended

conversation context. Although MT comprises only 1.2% of val queries, it accounts for a disproportionate share of total spending.

- **Switchcraft saves 80–92% on most dataset groups** compared to the best single model (GPT-5.3-chat), with savings ranging from 79% (xLAM-60K) to 92% (BFCL-Single). The exception is ConFETTI, where savings are only 30%—the router routes many ConFETTI queries to expensive models, reflecting the difficulty of these conversational queries.
- **Qwen-3.5-9B is the cheapest model across all groups**, yet its accuracy is too low to be the default choice (73.7% overall vs. 83.6% for GPT-5.3-chat). On validation, Switchcraft achieves 82.9% accuracy at a cost (6.8×10^{-4} \$/query) only $3.2\times$ that of Qwen, while closing most of the accuracy gap to GPT-5.3-chat.

G Router fine-tuning configuration

This appendix details the hyperparameters and fine-tuning settings used for all three encoder models evaluated in Section 4.6. All models share the same fine-tuning framework (Hugging Face Transformers), loss function, and data pipeline; they differ only in the base model, sequence length, batch size, and learning rate.

G.1 Training data

Training labels are generated from the evaluation results of all 8 target models on 14 dataset splits (Section 4). For each query, a model is labeled as *correct* if its AST-match score ≥ 0.75 (the same threshold used by BFCL v3 scoring). This produces a **multi-label** binary target vector per query—multiple models may be correct for the same input.

Stratified 80/10/10 splits. Each dataset split (the 14 rows of Table 2) is partitioned *independently* into 80% train / 10% validation / 10% test, then the per-dataset slices are concatenated and shuffled to form the global splits. Stratifying by dataset guarantees that every benchmark and every category within BFCL (Simple, Multiple, Parallel, Live*, Multi-turn, etc.) appears in the train, validation, and test sets in its native 80/10/10 ratio, so no benchmark is held out entirely. Deduplication on the (*query, tools*) pair is performed *within each dataset* before splitting, preventing near-duplicates from leaking across the train/validation/test boundary. The final sizes are approximately 98,000 training, 12,267 validation, and 12,282 test examples.

Seed-dependent splits. The seed value controls *both* the random initialization of the classifier head *and* the per-dataset shuffle that produces the 80/10/10 split. Concretely, each seed s deterministically produces a distinct train/validation/test partition (`train_data_*_`s`.csv`, `val_data_*_`s`.csv`, `test_data_*_`s`.csv`) by seeding `random.seed(s)` before shuffling each dataset. The 80/10/10 ratio and per-dataset stratification are preserved across seeds—only the specific examples assigned to each split change. As a result, the ± 0.41 pp seed variance reported in Section 4.6 reflects variation across both classifier initializations *and* train/validation/test partitions, providing a stronger generalization signal than a fixed-split protocol with only initialization-level seed variation.

To assess seed stability, each model is fine-tuned independently with 20 random seeds (see Appendix I). The full list of seeds is given at the end of this appendix.

G.2 Input representation

Each training example is tokenized using the token packing strategy described in Section 3.1. Table 13 shows the per-model token budgets.

Table 13: Token packing budgets by model.

Setting	DistilBERT	DeBERTa-v3	ModernBERT
Max sequence length	512	512	8,192
Max tool tokens	100	100	1,600

Within the budget, the packing algorithm prioritizes the latest turn, compact tool signatures, and earlier turns newest-first, as detailed in Section 3.1.

G.3 Shared hyperparameters

Table 14 lists settings common to all three models.

Table 14: Shared training hyperparameters.

Hyperparameter	Value
Optimizer	AdamW
LR scheduler	Linear decay with warmup
Warmup ratio	0.1
Weight decay	0.01
Max epochs	30
Early stopping patience	3 epochs (on val macro-F1)
Loss function	BCEWithLogitsLoss
Class balancing (pos_weight)	Disabled
Prediction threshold	0.5
Mixed precision	FP16 (when GPU available)
Best-model selection	Validation macro-F1

G.4 Per-model hyperparameters

Table 15 lists settings that differ between models. The learning rate and batch size were selected via preliminary experiments; DistilBERT tolerates a higher learning rate due to its smaller depth, while DeBERTa-v3 and ModernBERT fine-tune stably at 2×10^{-5}. ModernBERT uses a smaller batch size to accommodate its longer sequences within GPU memory.

Table 15: Per-model training hyperparameters.

Hyperparameter	DistilBERT	DeBERTa-v3	ModernBERT
Base model	`distilbert-base-uncased`	`deberta-v3-base`	`ModernBERT-base`
Parameters	66M	86M	149M
Hidden size	768	768	768
Layers	6	12	22
Attention heads	12	12	12
Learning rate	5×10^{-5}	2×10^{-5}	2×10^{-5}
Train batch size	16	32	8
Eval batch size	16	64	8

G.5 Classification head

All models use a single linear classification head mapping from hidden dimension (768) to the number of target classes (8 models). No intermediate layers or dropout beyond what is built into the pretrained encoder are added. The loss is binary cross-entropy with logits, computed independently for each class, enabling multi-label prediction.

G.6 Inference-time selection

At inference time, the model outputs 8 sigmoid probabilities (one per target model). The **lowest-cost correct** selection strategy is applied: among all classes whose probability exceeds the threshold (0.5), the model with the lowest profiled inference cost is selected. If no class exceeds the threshold, the class with the highest probability (`argmax`) is chosen as a fallback.

G.7 Compute environment

All fine-tuning was conducted on a node with 4× NVIDIA A100 80 GB PCIe GPUs. The parallelism strategy differs by model: DistilBERT uses PyTorch DataParallel across all 4 GPUs per seed (effective

batch size $16 \times 4 = 64$; seeds run sequentially); DeBERTa-v3 uses DistributedDataParallel (DDP) via `torchrun` across all 4 GPUs per seed (effective batch size $32 \times 4 = 128$; seeds run sequentially); ModernBERT does not support multi-GPU parallelism, so seeds are distributed across GPUs (5 seeds per GPU, run sequentially in parallel across devices; batch size 8 on a single GPU). Table 16 lists the software stack.

Table 16: Software environment for router fine-tuning.

Component	Version
Python	3.11
PyTorch	2.6.0 (CUDA 12.4)
Transformers	4.55.2
Accelerate	1.10.0

Wall-clock time. DistilBERT fine-tunes in ∼50 min per seed on 4 GPUs via DataParallel (17 hr total for 20 seeds, run sequentially). DeBERTa-v3 requires ∼4.5 hr per seed using 4-GPU DDP. ModernBERT requires ∼6 hr per seed on a single GPU (30 hr wall-clock with 5 seeds per GPU × 4 GPUs in parallel). Total compute across all 60 runs (3 models × 20 seeds): ∼70 (DistilBERT, 4 GPUs × 17 hr) + ∼360 (DeBERTa, 4 GPUs × 90 hr) + ∼120 (ModernBERT) ≈ 550 A100 GPU-hours. No gradient accumulation is used; the per-device batch sizes in Table 15 are multiplied by the number of GPUs to obtain the effective global batch size.

Seeds. The 20 seeds used across all models are: `9999, 1, 2, 3, 4, 5, 6, 7, 8, 9, 997, 420, 1234, 2025, 2024, 666, 247, 42, 31415, 27182`.

H Fine-tuning curves

Figure 5 shows the fine-tuning and validation loss curves for all 20 DistilBERT seeds. Fine-tuning uses binary cross-entropy loss with AdamW (lr=5×10^{-5}, linear schedule with 10% warmup, weight decay 0.01) for up to 30 epochs, with early stopping (patience 3) on macro F1.

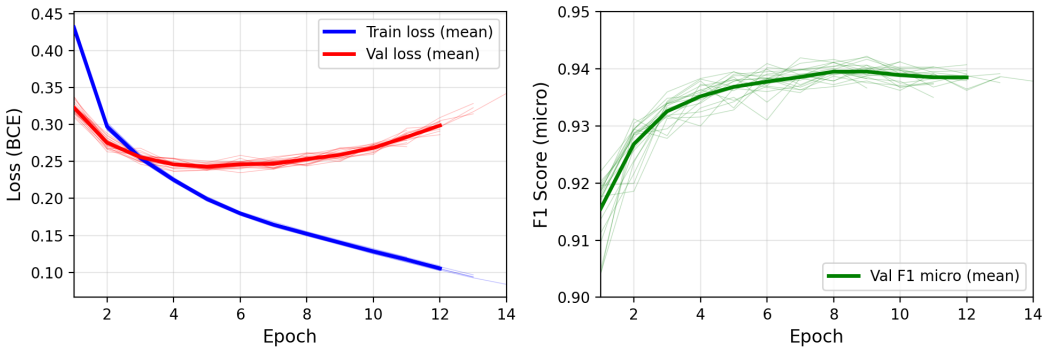

Figure 5: DistilBERT fine-tuning curves across 20 seeds. Left: training and validation loss. Right: validation micro-F1. Faint lines show individual seeds; bold lines show the mean (over seeds still fine-tuning at each epoch). Early stopping triggers between epochs 9 and 14 depending on the seed.

Training loss decreases steadily from 0.43 (epoch 1) to ∼0.10 (epoch 12). Validation loss reaches its minimum at epoch 5 (0.243 on average) and rises thereafter, indicating mild overfitting. Despite the rising validation loss, validation micro-F1 continues to improve slightly, plateauing around 0.939–0.940 from epoch 7 onward. This divergence between loss and F1 is expected for multi-label classification: the loss penalizes calibration (sigmoid probability), while F1 rewards ranking (threshold at 0.5). Early stopping monitors macro F1 and triggers between epochs 9 and 14 across seeds.

The narrow spread across all 20 seeds—standard deviation of F1 is only 0.0015 at epoch 9—confirms that fine-tuning is stable and the final checkpoint quality is not sensitive to random initialization (see also Section 4.6).

I Router seed stability

Figure 6 visualizes the distribution of Switchcraft's accuracy across 20 random seeds (DistilBERT encoder) on the validation set.

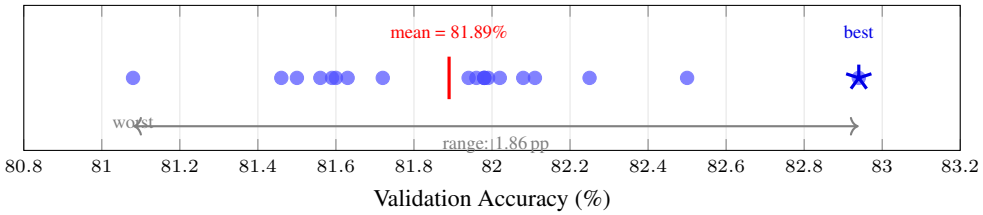

Figure 6: DistilBERT router accuracy across 20 random seeds. Each dot is one seed; the red bar marks the mean. The total range is 1.86 percentage points, confirming stability.

Table 17 reports the validation accuracy and average inference cost for all 20 seeds of each router encoder model, evaluated on the 12,267-example validation set.

Table 17: Validation accuracy (%) and average cost per query (10^{-4} \$) for each of the 20 random seeds across DistilBERT, ModernBERT, and DeBERTa-v3 routers (1× binary AST score, 12,267 examples). The best seed per model is **bolded**.

	DistilBERT		**ModernBERT**		**DeBERTa-v3**	
Seed	**Acc**	**Cost**	**Acc**	**Cost**	**Acc**	**Cost**
1	81.46	7	81.42	8	81.43	7
2	81.08	8	81.36	7	81.80	9
3	81.94	9	82.10	7	81.83	8
4	**82.94**	**7**	**83.02**	**6**	**82.89**	**6**
5	81.98	8	81.82	9	82.68	8
6	81.56	8	81.58	7	81.43	8
7	81.98	8	82.28	7	81.97	7
8	81.63	9	81.69	8	81.83	7
9	81.72	7	81.80	7	81.55	7
42	82.02	8	81.86	7	81.99	7
247	81.60	7	81.76	8	81.45	8
420	82.08	7	82.19	7	82.09	8
666	81.99	8	81.85	8	81.67	7
997	81.96	7	81.81	8	81.77	7
1234	82.25	8	82.26	7	82.11	7
2024	81.50	8	81.45	8	81.42	7
2025	81.59	9	81.65	7	81.53	7
9999	81.98	7	81.92	6	81.75	7
27182	82.50	7	82.17	6	82.54	7
31415	82.11	9	82.10	7	82.18	7
Mean	81.89	8	81.91	7	81.90	7
Std	±0.41	±1	±0.38	±1	±0.41	±1

All 20 DistilBERT seeds achieve accuracy between 81.08% and 82.94%, a range of only 1.86 percentage points, with a mean of 81.89% (±0.41). ModernBERT shows a similar pattern: its 20 seeds span 81.36%–83.02% (1.66 pp range, mean 81.91% ±0.38), indicating slightly tighter clustering. DeBERTa-v3 is comparable: 81.42%–82.89% (1.47 pp range, mean 81.90% ±0.41). For all three models, seed 4 is the top performer. Average inference cost is similarly stable across seeds for all architectures, varying between 6 and 9 × 10^{-4} \$. This narrow spread confirms that Switchcraft's advantage over single-model baselines is not an artifact of seed selection: even the

worst seed across all three models (81.08%, DistilBERT) outperforms six of the eight individual LLMs and all three heuristic routers reported in Table 3.

J Adaptive thresholding

J.1 Motivation

Our default routing pipeline in Switchcraft applies a fixed sigmoid threshold ($\theta = 0.5$) to convert multi-label logits into binary predictions, then selects the *cheapest* model among those predicted positive. When no class exceeds the threshold, the router falls back to the single highest-probability class (argmax). This cost-minimizing strategy is optimal when the goal is maximum savings at acceptable accuracy—the regime reported in Section 4.3.

However, production deployments may face different operating points: some workloads tolerate higher cost if it brings meaningful accuracy gains, while others demand the absolute lowest cost floor. A natural question is whether the sigmoid threshold can be *tuned* to trade off between accuracy and cost—and whether alternative decision rules expose a broader Pareto frontier.

J.2 Strategies evaluated

We evaluate two families of thresholding strategies:

Constant threshold (θ). The same approach as the default pipeline, but sweeping $\theta \in \{0.5,\ 0.75\}$. A higher threshold shrinks the set of "positive" classes, forcing more predictions through the argmax fallback (which picks the highest-probability model regardless of cost). This trades cost for accuracy: the router becomes more willing to choose an expensive-but-confident model.

Max-probability drift (δ). An alternative rule that ignores the binary threshold entirely. For each query, we identify all classes whose predicted probability is within δ of the maximum probability:

$$\mathcal{C}_\delta = \left\{c : p_c \geq \max_j p_j - \delta\right\}$$

and then select the cheapest class in $\mathcal{C}_\delta$. Small δ (e.g., 0.001) effectively acts as argmax (only the single top-scoring class qualifies), while large δ (e.g., 0.2) permits aggressive cost optimization by considering near-ties.

We sweep $\delta \in \{0.001,\ 0.01,\ 0.1,\ 0.2\}$.

J.3 Results

We evaluate all six strategies across all 20 random seeds of the DistilBERT router and report test-set accuracy and cost (Table 18, Figure 7).

Table 18: Accuracy–cost trade-off under different thresholding strategies (mean $\pm$ std across 20 seeds, test set).

Strategy	**Parameter**	**Accuracy (%)**	**Cost (10^{-4} \$/query)**
Constant threshold	$\theta = 0.5$	81.90 ± 0.30	7.3 ± 0.8
Constant threshold	$\theta = 0.75$	82.82 ± 0.28	10.8 ± 2.1
Max-prob drift	$\delta = 0.2$	82.75 ± 0.32	8.4 ± 1.1
Max-prob drift	$\delta = 0.1$	83.53 ± 0.32	11.7 ± 2.7
Max-prob drift	$\delta = 0.01$	84.79 ± 0.32	31.4 ± 6.3
Max-prob drift	$\delta = 0.001$	85.16 ± 0.29	41.7 ± 6.3

J.4 Discussion

The results reveal a smooth accuracy–cost Pareto frontier spanning from 81.9% accuracy at 7.3×10^{-4} \$/query (aggressive cost minimization) to 85.2% at 41.7×10^{-4} \$/query (accuracy maximization):

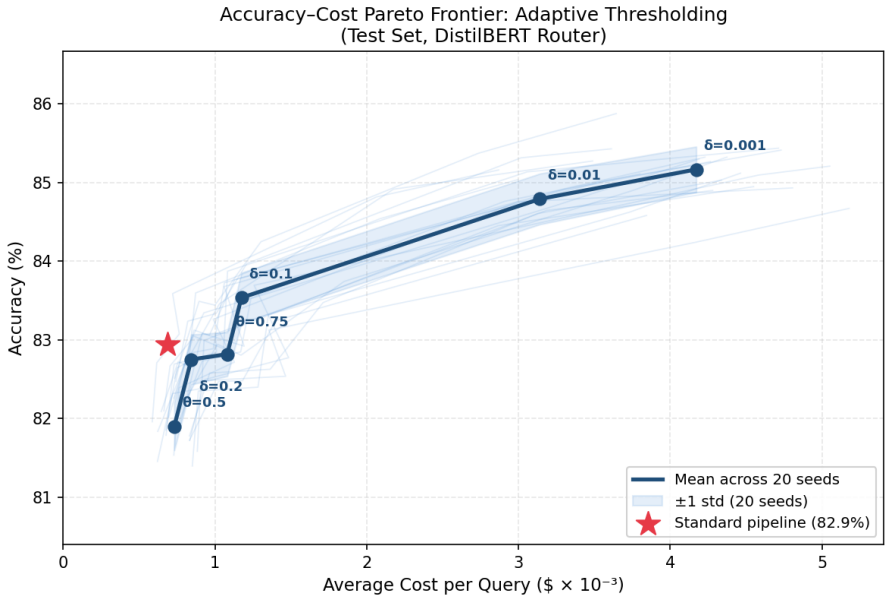

Figure 7: Accuracy–cost Pareto frontier for different thresholding strategies. Each faint curve traces one seed's trade-off across six strategies; the bold curve connects means across 20 seeds with a ± 1 std shaded band. The standard pipeline operating point (82.9%, 6.8×10^{-4} \$) is shown as a star.

- **Max-prob drift dominates constant thresholds.** At comparable cost, drift-based strategies consistently achieve higher accuracy (e.g., $\delta = 0.1$ achieves 83.5% at 11.7×10^{-4} \$ vs. $\theta = 0.75$ at 82.8% and 10.8×10^{-4} \$).
- **Over 3 percentage points of accuracy are available** by relaxing the cost budget; even the sweet spot ($\delta = 0.1$ at 11.7×10^{-4} \$) is a 3.7× cost reduction compared to the best single LLM (43.1×10^{-4} \$ for GPT-5.3-chat).
- **The sweet spot is** $\delta = 0.1$, which gains +1.6 pp accuracy over the default at only 4.4×10^{-4} \$ additional cost per query ($440 per million queries).
- **Seed variance is low** across all strategies ($\pm$0.3 pp), confirming that the trade-off is stable and not an artifact of seed selection.

These results suggest that deployments with moderate cost tolerance should consider $\delta \in [0.01, 0.1]$ for a favourable accuracy–cost balance, while the default $\theta = 0.5$ remains optimal for strict cost minimization.

K Ablation: token packing

Switchcraft's input representation uses an intelligent *token packing* strategy (Section 3.1) to fit the most decision-relevant information into the encoder's context window.

When using ModernBERT's 8,192-token context window, we allocate up to 1,600 tokens for tool definitions (vs. 100 for DistilBERT).

To isolate the contribution of this design, we fine-tune an ablation model that replaces the packing strategy with *naive right-truncation*: the raw conversation text is fed directly to the tokenizer, which truncates at 512 tokens with no prioritization of recent turns, tool signatures, or metadata. All other hyperparameters (learning rate, epochs, class weighting, early stopping) remain identical.

Results. Table 5 (in Section 4.6) reports the test-set performance (12,282 examples) for a single seed. Without token packing, accuracy drops by 1.66 percentage points and average routing cost increases by 21%. Switchcraft more frequently selects expensive models when it lacks the structured context that packing provides—particularly tool signatures and metadata that signal query complexity.

Implications. The accuracy gap confirms that token packing is a meaningful contributor to router quality—not merely a convenience for handling long inputs. Because agent conversations often exceed 512 tokens (median length in our dataset is ∼800 tokens), naive truncation discards the most recent user turn in roughly half of all examples, depriving Switchcraft of the single strongest routing signal. The cost increase further indicates that without structured packing, Switchcraft defaults to routing queries to larger, more expensive models as a hedge against uncertainty.

L Router inference latency

A practical router must not only minimize its own prediction cost but also provide a prediction with minimal latency since it would add to TTFT of the subsequent LLM call it dispatches. We benchmark both router encoder models—DistilBERT (66 M parameters) and ModernBERT (149 M parameters)—to demonstrate that DistilBERT is deployable on commodity inference hardware with low cost, low latency and high throughput, and that the marginal accuracy gain from ModernBERT (+0.08 pp) comes at a significant inference cost.

Infrastructure choice. We benchmark on a single NVIDIA T4 GPU (16 GB, Turing architecture), one of the most widely deployed inference accelerators in public clouds (AWS `g4dn`, Azure `NC_T4_v3`, GCP `n1-standard + T4`). The T4 represents the *cheapest* GPU option (~$0.35/hr spot[2]); strong performance here demonstrates minimal cost overhead with no specialised hardware required.

Methodology. We load each model's best-seed checkpoint and run inference on synthetic inputs at three sequence lengths: short (64 tokens), medium (200 tokens), and long (512 tokens—DistilBERT's maximum context). For each configuration we perform 50 warmup iterations followed by 200 timed iterations, reporting wall-clock latency percentiles (P50, P95, P99) and sustained throughput (queries/sec). All measurements use `torch.cuda.synchronize()` barriers for accurate timing. GPU utilisation is the mean SM (streaming multiprocessor) activity sampled at 10 ms intervals via NVML during each measurement window. We report FP32 precision with PyTorch 2.6 and CUDA 12.4.

Results. Tables 19 and 20 present the full latency and throughput measurements for DistilBERT and ModernBERT respectively.

Table 19: DistilBERT router inference (67 M params, 284 MB GPU memory) on NVIDIA T4, FP32.

Batch	Seq Len	P50 (ms)	P95 (ms)	P99 (ms)	Throughput (qps)	Mean GPU Util (%)
1	64	3.1	3.2	3.2	325	88
8	64	12.6	13.1	13.3	633	98
32	64	48.0	49.4	49.7	665	100
64	64	88.7	90.1	90.3	722	100
128	64	194.8	199.8	200.3	656	100
1	200	7.2	7.3	7.4	139	97
8	200	40.5	41.0	41.1	198	100
32	200	167.0	169.6	170.3	192	100
64	200	344.5	348.8	350.8	186	100
128	200	683.0	691.3	692.9	188	100
1	512	16.6	16.9	17.1	60	98
8	512	121.2	121.7	122.0	66	100
32	512	501.8	505.4	506.6	64	100
64	512	994.2	1001.5	1002.6	64	100
128	512	1989.9	2008.4	2013.0	64	100

Discussion. For single-query routing at intermediate prompt lengths (200 tokens), DistilBERT adds only **7.3 ms at P95**—over an order of magnitude faster than the cheapest LLM in our pool (GPT-5-nano averages ~200 ms per call). ModernBERT, despite its marginal accuracy advantage

[2]Based on AWS `g4dn.xlarge` spot pricing as of April 2026: `https://aws.amazon.com/ec2/spot/pricing/`.

Table 20: ModernBERT router inference (149 M params, 871 MB GPU memory) on NVIDIA T4, FP32.

Batch	Seq Len	P50 (ms)	P95 (ms)	P99 (ms)	Throughput (qps)	Mean GPU Util (%)
1	64	17.5	17.6	17.8	57	42
8	64	33.0	33.5	33.6	243	98
32	64	138.2	141.1	142.3	231	100
64	64	288.2	298.2	301.2	222	100
128	64	578.7	598.1	600.5	222	100
1	200	22.5	23.1	34.6	44	92
8	200	141.5	143.3	144.2	57	99
32	200	569.1	572.5	573.6	56	100
64	200	1013.3	1021.2	1022.4	63	100
128	200	2028.8	2043.3	2045.5	63	100
1	512	45.0	46.0	46.1	22	98
8	512	367.1	370.6	371.3	22	100
32	512	1362.4	1372.8	1375.4	24	100
64	512	2758.2	2772.0	2774.2	23	100
128	512	5531.4	5557.4	5563.7	23	100

(+0.08 pp), is **3.2× slower** at the same operating point (23.1 ms P95). At maximum sequence length (512 tokens), DistilBERT remains under 17 ms P95 while ModernBERT requires 46 ms.

Peak throughput tells an even starker story: DistilBERT achieves **722 queries/sec** versus ModernBERT's 243—a **3.0**× advantage. ModernBERT also consumes **3.1× more GPU memory** (871 MB vs. 284 MB). These differences matter in production: at DistilBERT's throughput, a single $0.35/hr T4 can serve over 2.5 million routing decisions per hour, making the per-query router cost effectively zero (~1.4×10^{-7} $).

These results confirm that DistilBERT's combination of near-identical accuracy (82.94% vs. 83.02%) and dramatically better inference efficiency makes it the clear production choice, particularly on low-cost commodity hardware like the T4.

M Misrouted query breakdowns

This appendix provides detailed breakdowns of the 2,093 misrouted queries (17.1% of the 12,267-example validation set) from Switchcraft's best-seed configuration (DistilBERT, seed 4). A query is *misrouted* if the predicted model answers incorrectly (*wrong model*: 902 cases) or if no model in the pool answers correctly (*no correct model*: 1,191 cases).

Misroute rate by number of tool definitions

Table 21: Misroute rate by number of tool definitions.

Tools	Misrouted	Total	Rate (%)
1	1,044	6,817	15.3
2–3	564	3,282	17.2
4–6	358	1,739	20.6
7–10	61	263	23.2
11–50	66	166	39.8

As shown in Table 21, the misroute rate increases monotonically with the number of tool definitions in the query, rising from 15.3% for single-tool queries to 39.8% for queries with 11–50 tools. Queries with many tools tend to have longer function schemas that may exceed the router's 512-token context window, causing information loss.

Table 22: Misroute rate by number of conversation turns.

Turns	Misrouted	Total	Rate (%)
1	1,577	9,037	17.5
3–4	210	1,385	15.2
5–8	239	1,690	14.1
9–20	65	153	42.5
21+	2	2	100.0

Misroute rate by number of conversation turns

Table 22 shows that the relationship with turn count is non-monotonic: moderate multi-turn conversations (3–8 turns) are actually slightly easier to route than single-turn queries (14–15% vs. 17.5%). Very long conversations ($\geq$9 turns) spike sharply to 42.5%, though the sample size is small (153 queries).

Misroute rate by text length

Table 23: Misroute rate by text length (in tokens).

Text Length (tokens)	Misrouted	Total	Rate (%)
4–15	269	2,673	10.1
15–22	598	3,092	19.3
22–44	666	3,406	19.6
44–68	272	1,861	14.6
68–3,137	288	1,235	23.3

Table 23 shows that very short queries ($<$15 tokens) are the easiest to route (10.1%), likely because they are simple single-function calls. Mid-range and long queries are harder, with the longest bucket (68+ tokens) reaching 23.3%.

Misroute rate by number of correct models

Table 24: Misroute rate by number of correct models in the pool.

Models Correct	Misrouted	Total	Rate (%)
0	1,191	1,191	100.0
1	203	299	67.9
2	147	321	45.8
3	115	298	38.6
4	124	335	37.0
5	138	443	31.2
6	128	1,003	12.8
7	47	2,480	1.9
8	0	5,897	0.0

Table 24 reveals the strongest predictor of routing difficulty. When all 8 models answer correctly, the misroute rate is 0%—every prediction yields a correct answer. When only 1 model is correct, the router must identify that specific model among 8 candidates, and the error rate climbs to 67.9%. The 1,191 queries where no model is correct are misrouted by definition (100%) and account for 56.9% of all misroutes.

Predicted vs. Oracle model confusion

Among the 902 *wrong-model* errors (where a correct model existed but the router chose one that failed), Table 25 shows the predicted and oracle model distributions:

Table 25: Predicted vs. oracle model distribution for the 902 wrong-model errors.

Model	Predicted (%)	Oracle (%)
GPT-5.3-chat	49.9	4.1
GPT-5-nano	38.8	8.3
Qwen-3.5-9B	5.3	44.7
GPT-5.4-mini	1.6	6.7
Kimi-K2.5	1.5	7.6
GPT-5-mini	1.3	9.6
GPT-5.4-nano	1.1	17.6
GPT-5.4	0.4	1.3

Switchcraft over-predicts GPT-5.3-chat (49.9% of wrong predictions vs. 4.1% of oracle selections) and GPT-5-nano (38.8% vs. 8.3%), while under-predicting Qwen (5.3% vs. 44.7%) and GPT-5.4-nano (1.1% vs. 17.6%). The top confusion flows are:

- GPT-5-nano → Qwen (320 cases): the router predicts nano but it fails, while Qwen would have been the cheapest correct model.
- GPT-5-nano → GPT-5.4-nano (126 cases): nano fails, but the slightly more expensive 5.4-nano succeeds.
- GPT-5-nano → GPT-5-mini (63 cases): nano fails on queries that require the mini model's capability.

N Per-model token usage and chattiness analysis

Token counts are extracted from each model's API response metadata. Every result record includes `input_token_count`, `output_token_count`, and (where applicable) `reasoning_token_count`. The output token count corresponds to the API's `completion_tokens` field, which *includes* reasoning tokens; the reasoning token count is the subset reported via `completion_tokens_details.reasoning_tokens`. Thus the visible (non-reasoning) output is output − reasoning tokens. For multi-turn conversations, token counts are stored as nested lists (one entry per turn); we sum across all turns to obtain the per-example total. Reasoning tokens are billed at the output token rate but are not returned in the response text.

Table 26 reports the average input, output, and reasoning token counts per query for each model on the 12,267-example validation set, alongside per-token pricing.

Table 26: Per-model token consumption on the validation set (12,267 examples). Avg In = average input (prompt) tokens per query. Avg Out = average total output (completion) tokens per query, which *includes* reasoning tokens. Avg Reas = average reasoning tokens per query (a subset of Avg Out). % Reas = percentage of queries that consume any reasoning tokens. Pricing is in USD per million tokens. See the caveat on input token counts at the end of this section.

Model	In $/M	Out $/M	Avg In	Avg Out	Avg Reas	% Reas
GPT-5.4	2.50	15.00	2,215	77	27	44.4
GPT-5.4-mini	0.75	4.50	2,143	71	22	36.8
GPT-5.4-nano	0.20	1.25	2,185	62	14	20.0
GPT-5.3-chat	1.75	14.00	1,755	47	2	2.3
GPT-5-mini	0.25	2.00	2,222	121	67	49.3
GPT-5-nano	0.05	0.40	2,083	170	118	71.3
Kimi-K2.5	0.60	3.00	414	183	0	0.0
Qwen-3.5-9B	0.05	0.15	3,003	228	0	0.0

Several patterns emerge. First, **reasoning token usage varies dramatically**: GPT-5-nano uses reasoning tokens on 71% of queries (118 avg), while GPT-5.3-chat uses them on only 2%. This hidden cost substantially inflates GPT-5-nano's per-query expense beyond what its ultra-low per-token price ($0.05/M input) would suggest. Second, **output verbosity and reasoning are distinct phenomena**: Qwen-3.5-9B produces the most total output tokens (228 avg, all visible) but zero

reasoning tokens, whereas GPT-5-nano's 170 avg output tokens include 118 reasoning tokens—leaving only 52 visible output tokens, the fewest of any model.

Profiled per-query cost. The router's cost-based tie-breaking uses *profiled costs*: the actual dollar cost of each model computed from its observed token consumption and per-token pricing, rather than from list price alone. Table 27 reports the average profiled cost per query for each model, computed over all 157,101 raw examples (before the pipeline's deduplication step that yields 122,267 records for splitting; see Section 4.1). The same per-model cost will differ modestly across splits because the share of expensive multi-turn queries varies: e.g. GPT-5.3-chat averages 32.8×10^{-4} \$ on the full raw dataset, 37.3×10^{-4} \$ on validation (Table 28), and 43.1×10^{-4} \$ on the test set (Table 3). These differences reflect split composition, not pricing changes.

Table 27: Profiled average inference cost per query (10^{-4} \$), computed from observed token consumption across all 157,101 raw examples (pre-deduplication; see Section 4.1). Models are sorted by ascending cost. This ordering determines which model the router selects when multiple candidates are predicted correct.

Model	**Avg Cost / Query** (10^{-4} **\$**)
GPT-5-nano	1.4
Qwen-3.5-9B	1.7
GPT-5.4-nano	4.2
GPT-5-mini	6.4
Kimi-K2.5	7.2
GPT-5.4-mini	14.5
GPT-5.3-chat	32.8
GPT-5.4	50.4

The profiled cost ordering matches the list-price ordering (Table 3) for all eight models. This means that, for our current model pool, a simpler price-based ranking would yield the same routing behavior. However, this equivalence is not guaranteed in general: a model with a low list price but high output verbosity or heavy reasoning token usage could be more expensive per query than a model with a higher list price but more concise responses. In earlier versions of Switchcraft where we considered a different basket of models, we did observe such cost-price inversions.

Cost decomposition. Table 28 breaks the per-query cost into its input and output components, revealing how much each contributes to the total. For most models input cost accounts for 60–85% of total cost, meaning that output token differences—the driver of the chattiness metric (Section 4.8)—are compressed when viewed through the lens of total cost.

Table 28: Per-query cost decomposition on the validation set (12,267 examples). Input Cost and Output Cost are the average per-query costs (`avg_input_tokens` $\times$ `input_rate` and `avg_output_tokens` $\times$ `output_rate`, respectively). Input % is the share of total cost attributable to input tokens. Models are sorted by ascending total cost.

Model	**Input Cost** (10^{-4} **\$**)	**Output Cost** (10^{-4} **\$**)	**Total Cost** (10^{-4} **\$**)	**Input %**
GPT-5-nano	1.0	0.7	1.7	60
Qwen-3.5-9B	1.5	0.3	1.8	81
GPT-5.4-nano	4.4	0.8	5.2	85
GPT-5-mini	5.6	2.4	8.0	70
Kimi-K2.5	2.5	5.5	8.0	31
GPT-5.4-mini	16.1	3.2	19.3	83
GPT-5.3-chat	30.7	6.6	37.3	82
GPT-5.4	55.4	11.5	66.8	83

The outlier is Kimi-K2.5, whose input share is only 31%—a combination of its high output rate (\$3.00/M), a more efficient tokenizer, and early termination of failed multi-turn conversations (see caveat below). Because output dominates Kimi's cost, its chattiness of 1.31$\times$ (Section 4.8) reflects the raw output ratio (1.53$\times$) with relatively little compression. Conversely, Qwen-3.5-9B has the

highest raw output ratio (1.90×) but a chattiness of only 1.10× because input accounts for 81% of its cost.

Caveat on input token counts. The Avg In column in Table 26 and the input costs in Table 28 should be interpreted with two caveats. First, different models use different tokenizers, so the same prompt text produces different input token counts—for instance, on single-turn queries all GPT models encode the prompt as ∼213 tokens on average, while Kimi-K2.5's tokenizer produces ≈0.68× as many (144) and Qwen-3.5-9B's tokenizer produces ≈1.76× more. These differences reflect tokenizer design, not meaningful variation in model behavior. Second, on multi-turn conversations Kimi-K2.5 completes far fewer turns than other models—a median of 3 turns versus 21 for GPT models—resulting in more failures, fewer cumulative API calls and correspondingly lower total input token counts. Together, tokenizer efficiency and early conversation termination explain Kimi's unusually low average input count (414 tokens vs. ∼2,000 for GPT models) and its low input cost share (31%). Despite these caveats, the profiled per-query costs in Tables 27 and 28 are computed from each API's own reported token counts and pricing, so they accurately reflect what a deployment would actually be billed.

Chattiness derivation. For each model m, we compute an *expected cost* assuming it generates the cross-model average number of output tokens (120, averaged over all eight models on the validation set) while keeping its actual input cost fixed:

$$\text{expected}(m) = \underbrace{\bar{t}_{\text{in}}^{(m)} \cdot r_{\text{in}}(m)}_{\text{actual input cost}} + \underbrace{\bar{t}_{\text{out}} \cdot r_{\text{out}}(m)}_{\text{avg-output cost}}$$

where $\bar{t}_{\text{in}}^{(m)}$ is m's average input token count, $\bar{t}_{\text{out}} = 120$ is the cross-model average output token count, and $r_{\text{in}}, r_{\text{out}}$ are the per-token rates. **Chattiness** is the ratio of actual to expected cost:

$$\text{chattiness}(m) = \frac{\text{actual cost per query}(m)}{\text{expected cost}(m)}$$

Figure 8 visualizes this metric. Points above the $y{=}x$ diagonal cost more than expected (chattiness >1); points below cost less (chattiness <1).

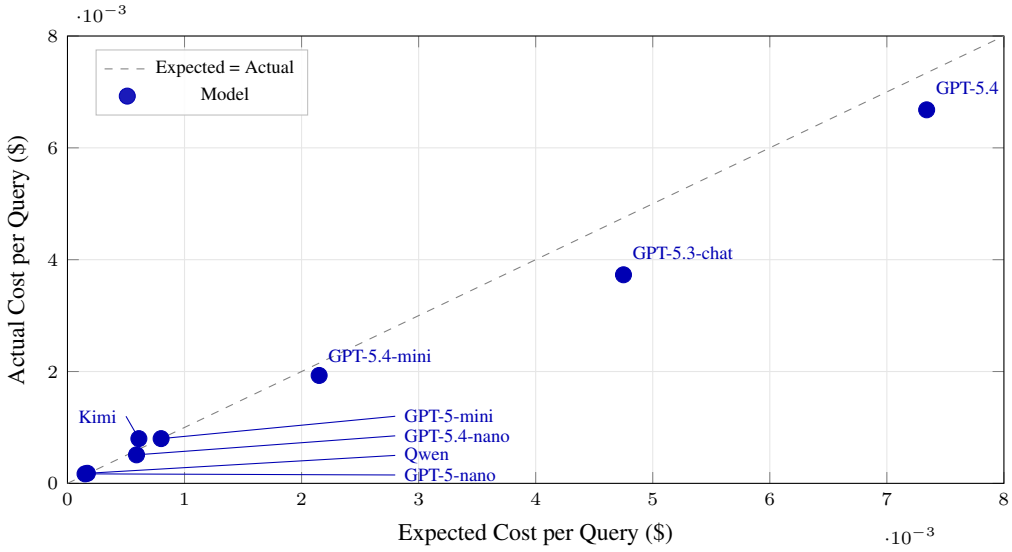

Figure 8: Expected vs. actual average cost per query on the validation set. *Expected cost* uses each model's actual input cost plus the cost of generating the cross-model average output (120 tokens) at that model's output rate. Points above the $y{=}x$ diagonal cost more than expected—indicating above-average output verbosity; points below cost less—indicating concise output.

Verbose budget models. Kimi-K2.5 has the highest chattiness in our pool ($\approx$1.31$\times$): it generates 183 output tokens on average—well above the cross-model mean of 120—making its actual cost 31% higher than expected. Because output accounts for 69% of Kimi's total cost (Table 28), its verbose output translates almost directly into higher total cost. Qwen-3.5-9B is the most verbose model in raw token terms (228 avg), yet its chattiness is only $\approx$1.10$\times$. The explanation is that 81% of its per-query cost is input (Table 28), so even a large output surplus barely moves the overall cost ratio. Despite sharing the cheapest input rate with GPT-5-nano ($0.05/M) and having an even cheaper output rate ($0.15/M vs. $0.40/M), Qwen costs 7% more per query (1.8 vs. 1.7×10^{-4} \$) due to its higher total token consumption—and achieves lower accuracy (72.40% vs. 79.15%).

Concise output vs. verbose output. GPT-5.3-chat has the second-highest output rate in our pool ($14.00/M), yet it is the most concise model (chattiness $\approx$0.78$\times$, 47 avg completion tokens, only 2% with reasoning). Its actual per-query cost (37.3×10^{-4} \$) is 22% below expected, making it the best accuracy–cost trade-off among single models. A router relying on list price alone would rank GPT-5.3-chat among the most expensive models; in practice its frugal output makes it one of the most cost-efficient. The GPT-5.4 family (GPT-5.4, GPT-5.4-mini, GPT-5.4-nano) also falls below the diagonal (chattiness 0.88–0.91$\times$), generating fewer output tokens than the cross-model average. For these premium models, input cost dominates (83–85% of total; Table 28), which compresses chattiness toward 1 even though their output token ratios range from 0.52–0.64$\times$ the cross-model mean.

O Reproduction with a different model basket

To demonstrate that our findings generalise beyond a single set of LLMs, we replicate the core evaluation with an *earlier model basket* comprising eight OpenAI models available in early 2025 (Table 29). This basket pre-dates the GPT-5.4 family and the open-weight models used in the main paper; it includes two reasoning models (o4-mini, GPT-5-chat) and spans a wide cost range from GPT-4.1-nano ($0.10/M input) to GPT-5 ($1.25/M input).

Table 29: Earlier model basket: LLMs available for routing, listed with per-token pricing (USD per million tokens).

Model	Input ($/M)	Output ($/M)
GPT-5	1.25	10.00
GPT-5-chat	1.25	10.00
GPT-4.1	2.00	8.00
o4-mini	1.10	4.40
GPT-5-mini	0.25	2.00
GPT-4.1-mini	0.40	1.60
GPT-5-nano	0.05	0.40
GPT-4.1-nano	0.10	0.40

Experimental setup. The evaluation protocol mirrors Section 4.2 exactly: the same 14 datasets, the same 80/10/10 stratified splits, the same DistilBERT architecture and hyperparameters, and the same 20-seed fine-tuning procedure. The only difference is the pool of candidate LLMs. We report results on the same held-out test set of 12,282 examples.

Results. Table 30 and Figure 9 present the accuracy–cost trade-offs. The key findings from the main paper hold:

Finding 1: the learned router dominates the cost–accuracy frontier. The DistilBERT agent router achieves 82.73% accuracy at 8.7×10^{-4} \$ per query—comparable to the best individual model (GPT-4.1 at 83.22%) while reducing cost by **88%** ($73.6 \rightarrow 8.7 \times 10^{-4}$ \$). This mirrors the main result in Section 4.3: a lightweight classifier can match frontier-model accuracy at a fraction of the cost.

Finding 2: a chat-fine-tuned router is insufficient for agentic workloads. This experiment additionally includes a *Chat Router*—a DistilBERT model fine-tuned primarily on a diverse set of

Table 30: Accuracy and average inference cost per query on the test set (12,282 examples) with the earlier model basket. Entities are grouped by type and sorted by accuracy.

Type	Entity	Accuracy (%)	Avg Cost (10^{-4} $)
Single LLM	GPT-4.1	83.22	73.6
	GPT-4.1-mini	82.18	10.5
	GPT-5	79.18	49.9
	GPT-5-nano	78.80	1.8
	GPT-5-mini	77.24	8.0
	GPT-4.1-nano	76.67	3.2
	GPT-5-chat	62.76	22.7
	o4-mini	60.93	23.6
Heuristic Router	Length	78.95	12.8
	Num. Tool Calls	67.84	66.9
	Num. Turns	62.67	54.0
Chat Router	Chat-fine-tuned	77.47	7.7
Agent Router	**DistilBERT (ours)**	**82.73**	**8.7**
Upper Bound	Oracle	89.65	12.7

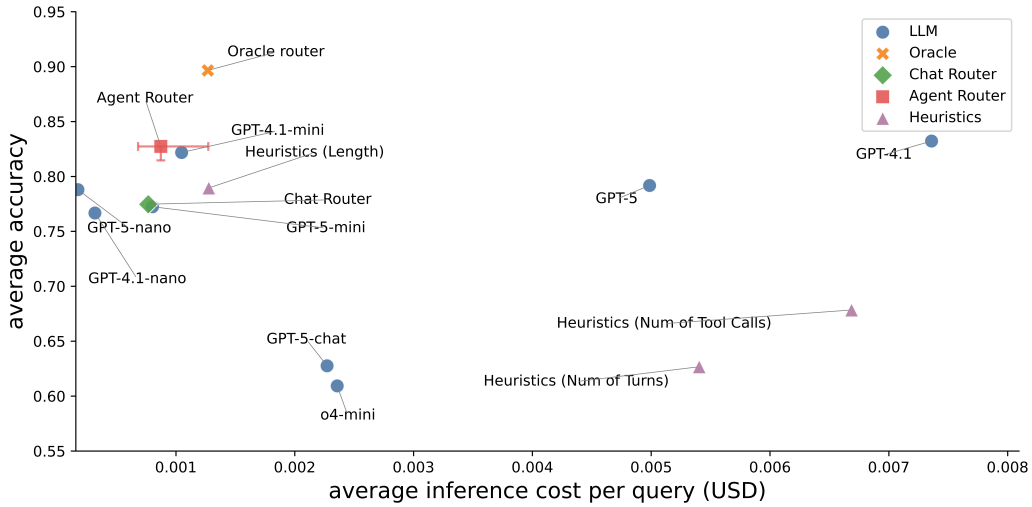

Figure 9: Accuracy–cost Pareto plot for the earlier model basket on the held-out test set (12,282 examples). Our agent-fine-tuned DistilBERT router (■) achieves 82.73% accuracy while the chat-fine-tuned router (♦) reaches only 77.47% despite similar cost, demonstrating the importance of agentic training data.

public chat completion benchmarks (non-agentic workloads; see Table 31 for the full list). Despite achieving low cost (7.7×10^{-4} $), the chat router reaches only **77.47%** accuracy—5.26 percentage points below our agent router (82.73%) and even below the cheapest heuristic baseline (Length at 78.95%).

This gap demonstrates that routing decisions learned from chat completions do not transfer effectively to agentic tool-calling scenarios. Agentic queries involve multi-turn conversations, complex tool schemas, and interleaved reasoning that differ fundamentally from single-turn chat. The techniques presented in this paper—agentic evaluation data, multi-label fine-tuning, and cost-aware inference—are essential for effective routing in this domain.

Finding 3: heuristic performance varies with the model pool. With the earlier model basket, the Length heuristic performs best (78.95%), while Num. Turns drops to 62.67%—a reversal from the main results where Num. Turns leads (80.41%). This suggests that heuristic effectiveness is

tightly coupled to the specific models available and does not generalise, whereas the learned router consistently performs well regardless of the underlying model pool.

Table 31: 64 public datasets used to train the chat-fine-tuned router baseline. HF = HuggingFace, GH = GitHub.

Dataset	Source	Dataset	Source
AGIEval	GH (ruixiangcui/AGIEval)	mgsm	GH (google-research/url-nlp)
ai2_arc	HF (allenai/ai2_arc)	mlqa	HF (facebook/mlqa)
anli	HF (facebook/anli)	mmlu	HF (cais/mmlu)
Belebele	HF (facebook/2M-Belebele)	MMLU-Pro	HF (TIGER-Lab/MMLU-Pro)
bigcodebench	HF (bigcode/bigcodebench)	narrativeqa	HF (deepmind/narrativeqa)
BioInstructQA	HF (BioMistral/BioInstructQA)	Natural-Questions	GH (google-research-datasets)
boolq	HF (google/boolq)	OpenBookQA	GH (allenai/OpenBookQA)
BoolQ_robustness	HF (ibm-research/BoolQ_rob.)	pawsx	GH (google-research-datasets/paws)
cnn_dailymail	HF (abisee/cnn_dailymail)	piqa	GH (ybisk/ybisk.github.io)
code_generation	HF (livecodebench/code_gen.)	PubMedQA	HF (qiaojin/PubMedQA)
code_gen._lite	HF (livecodebench/code_gen._l.)	qasper	HF (allenai/qasper)
commonsense_qa	HF (tau/commonsense_qa)	QMSum	GH (Yale-LILY/QMSum)
CyberMetric	GH (cybermetric/CyberMetric)	quac	HF (allenai/quac)
d2n	HF (NTaylor/d2n)	race	HF (ehovy/race)
function-calling	HF (gorilla-llm/BFCL)	RCT-summarization	GH (bwallace/RCT-summ.-data)
gpqa	GH (Idavidrein/gpqa)	samsum	HF (Samsung/samsum)
gpt4_judge_battles	HF (routellm/gpt4_judge_b.)	social_i_qa	HF (allenai/social_i_qa)
gsm8k	HF (openai/gsm8k)	squad_v2	HF (rajpurkar/squad_v2)
HeadQA	HF (dvilares/head_qa)	SQuALITY	GH (nyu-mll/SQuALITY)
HellaSwag	GH (rowanz/hellaswag)	StoryCloze	GH (EleutherAI/lm-eval.-harness)
HumanEval	GH (openai/human-eval)	SummScreen	GH (mingdachen/SummScreen)
humanevalplus	HF (evalplus/humanevalplus)	ToxiGen	HF (toxigen/toxigen-data)
lambada	HF (cimec/lambada)	trivia_qa	HF (mandarjoshi/trivia_qa)
MATH-500	HF (HuggingFaceH4/MATH-500)	TruthfulQA	GH (sylinrl/TruthfulQA)
MathInstruct	HF (TIGER-Lab/MathInstruct)	TyDiQA	GH (google-research-datasets/tydiqa)
mbpp	HF (Muennighoff/mbpp)	WinoGrande	GH (allenai/winogrande)
mbppplus	HF (evalplus/mbppplus)	wmdp	HF (cais/wmdp)
med_qa	HF (bigbio/med_qa)	xcopa	HF (cambridgeltl/xcopa)
MedCalc-Bench	GH (ncbi-nlp/MedCalc-Bench)	xlsum	HF (csebuetnlp/xlsum)
medbullets	GH (HanjieChen/ChallengeClQA)	xnli	HF (facebook/xnli)
medical-o1-ver.	HF (FreedomIntelligence)	xquad	HF (google/xquad)
medmcqa	HF (openlifescienceai/medmcqa)	xstory_cloze	HF (juletxara/xstory_cloze)

P Probabilistic correctness labels

P.1 Motivation

The standard evaluation protocol in this paper treats each LLM response as either correct or incorrect (a binary label). In practice, however, LLMs are *stochastic*: the same model may produce a correct answer on some invocations and an incorrect one on others for the *same* query. A model that answers correctly 95% of the time is fundamentally more reliable than one that answers correctly 55% of the time, yet both receive the same binary label of "correct" if evaluated on a single invocation.

This appendix presents a preliminary investigation into **probabilistic correctness labels**: instead of evaluating each model once per query, we evaluate it **20 times** and record the fraction of correct responses as a probability $p \in [0, 1]$. We then fine-tune Switchcraft using these soft labels and measure whether this richer supervision improves routing quality.

P.2 Methodology

Multi-invocation evaluation. Each of the eight candidate LLMs is called 20 times on every query. Each response is independently scored using the same AST comparison framework described in Section 4.2. The *correctness probability* for model m on query q is:

$$p(m, q) = \frac{\text{correct iterations}}{20}$$

This probability captures the reliability of a model on a given query, not just whether it *can* answer correctly.

Dataset coverage. Due to the 20× cost multiplier, we evaluated 12 of the 14 datasets (omitting Glaive and xLAM-60K), yielding 1,224 test examples. This is a smaller test set than the 12,282 examples used in the main paper, making results noisier but still directionally informative.

Soft-label training. The data generation pipeline stores each model's probability directly as the training label. For example, a query where GPT-4.1 answered correctly on 17/20 invocations and GPT-5-nano on 20/20 receives the label vector:

```
{"gpt-4.1": 0.85, "gpt-5-nano": 1.0, ...}
```

The DistilBERT classifier is then fine-tuned with `BCEWithLogitsLoss` against these continuous targets. Each output head learns to predict the *probability of correctness* for the corresponding model, rather than a hard correct/incorrect label.

Inference procedure. At inference time, the soft-label fine-tuned model still outputs a sigmoid probability for each candidate LLM. The prediction procedure is identical to the binary-label case:

1. Apply a threshold ($\theta = 0.5$) to each sigmoid output to determine which models are predicted to be "reliable" for this query.
2. Among the models above threshold, select the **cheapest** one (cost-aware tie-breaking using profiled per-query costs).
3. If no model exceeds the threshold, fall back to the model with the highest predicted probability (argmax).

Probability distributions. Figure 10 shows the distribution of correctness probabilities per model. The distributions are strongly bimodal: the vast majority of model–query pairs have probability near 0 (always incorrect) or 1 (always correct), with only 10–22% falling in the intermediate range (denoted "mid" in each subplot). This bimodality explains why soft labels do not dramatically change the learning problem—most labels are effectively binary. Figure 11 shows the same analysis grouped by dataset; multi-turn datasets (ConFETTI, multi-turn base/long-context) exhibit higher mid-range fractions (29–31%), indicating greater stochasticity in model responses for complex conversational tasks.

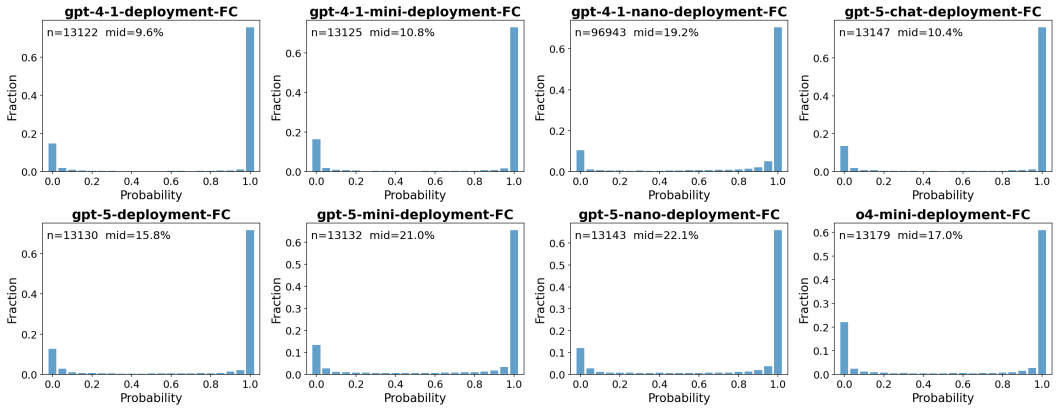

Figure 10: Distribution of correctness probabilities across 20 invocations, grouped by model. Each histogram shows the fraction of queries at each probability level. "mid" denotes the fraction of queries with probability strictly between 0 and 1.

P.3 Results

Table 32 and Figure 12 present the results. The model basket is the same as Appendix O (the earlier basket of eight OpenAI models).

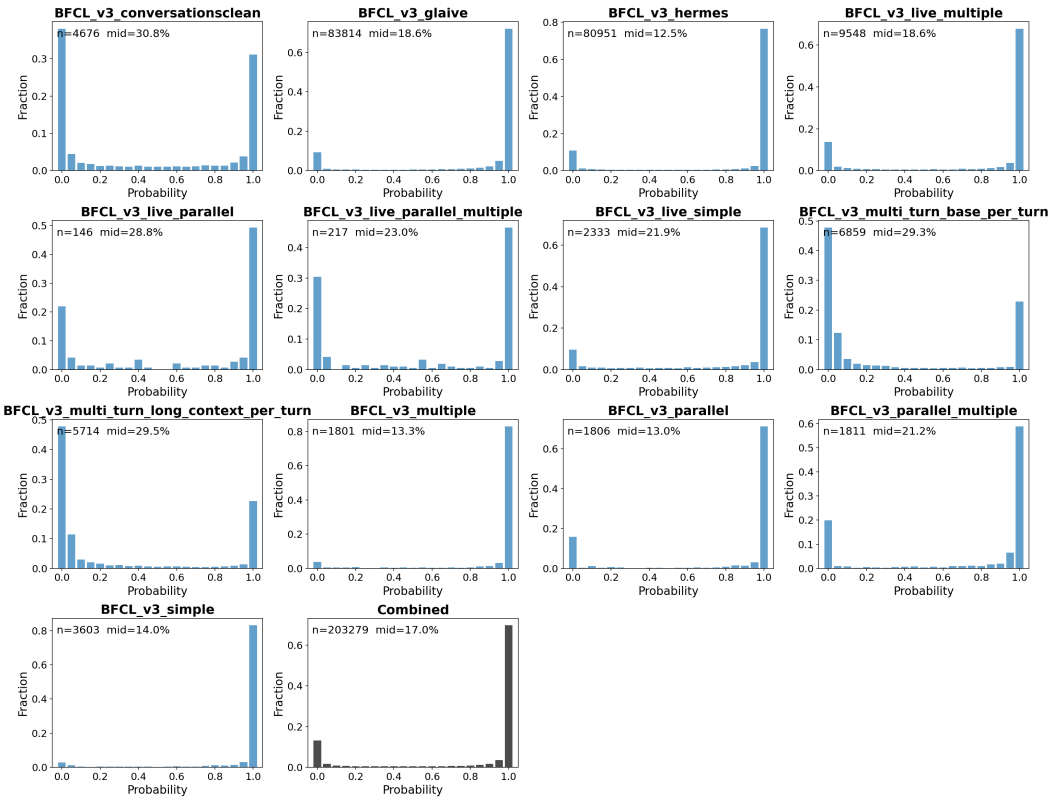

Figure 11: Distribution of correctness probabilities across 20 invocations, grouped by dataset. Multi-turn and conversational datasets show higher fractions of intermediate probabilities, indicating greater response variability.

P.4 Discussion

Soft labels match binary-label performance. The probabilistic router achieves **82.28%** accuracy at 94×10^{-4} \$ per query. Comparing to the binary-label router on the same model basket (Appendix O), which was evaluated on all 14 datasets (12,282 test examples), the binary router achieved 82.73% at 8.7×10^{-4} \$. However, this is not a direct comparison: the probabilistic experiment covers only 12 datasets (1,224 test examples) due to the prohibitive cost of 20× evaluation. The accuracy difference (0.45 pp) is within noise for the smaller test set, and the higher per-query cost (94×10^{-4} \$ vs. 8.7×10^{-4} \$) reflects the different dataset composition rather than the labeling strategy itself—the two omitted datasets (Glaive, xLAM-60K) contain many easy queries that drive down average cost in the full evaluation.

The prediction task is not substantially harder. Our initial hypothesis was that probabilistic labels would make the prediction task *harder* for the router—since a model might be correct only 60% of the time, the router must learn a finer-grained distinction than correct/incorrect. In practice, however, the accuracy distributions are strongly bimodal: most model–query pairs have probability near 0 or near 1, with relatively few in the intermediate range. The mean correctness probability across all 203,279 model–query evaluations is 0.80, and the median is 1.0, indicating that the majority of entries are deterministic.

Cost implications. The 20× evaluation cost is prohibitive at scale: evaluating 8 models × 12,282 queries × 20 iterations would require nearly 2 million API calls. We were only able to complete 12 of the 14 datasets within our budget, resulting in a smaller test set (1,224 examples). The reduced test set makes it difficult to draw definitive conclusions about whether soft labels improve routing quality.

Table 32: Accuracy and average inference cost per query on the test set (1,224 examples) with probabilistic correctness labels. Entities are grouped by type and sorted by accuracy.

Type	Entity	Accuracy (%)	Avg Cost (10^{-4} $)
Single LLM	GPT-5-chat	82.11	240
	GPT-4.1	80.31	506
	GPT-5	79.98	329
	GPT-5-nano	78.84	11
	GPT-4.1-mini	78.10	105
	GPT-5-mini	77.61	65
	GPT-4.1-nano	74.67	25
	o4-mini	73.69	195
Heuristic Router	Num. Tool Calls	80.96	430
	Length	77.94	84
	Num. Turns	74.35	338
Chat Router	Chat-fine-tuned	79.17	118
Agent Router	**DistilBERT (ours)**	**82.28**	**94**
Upper Bound	Oracle	91.01	205

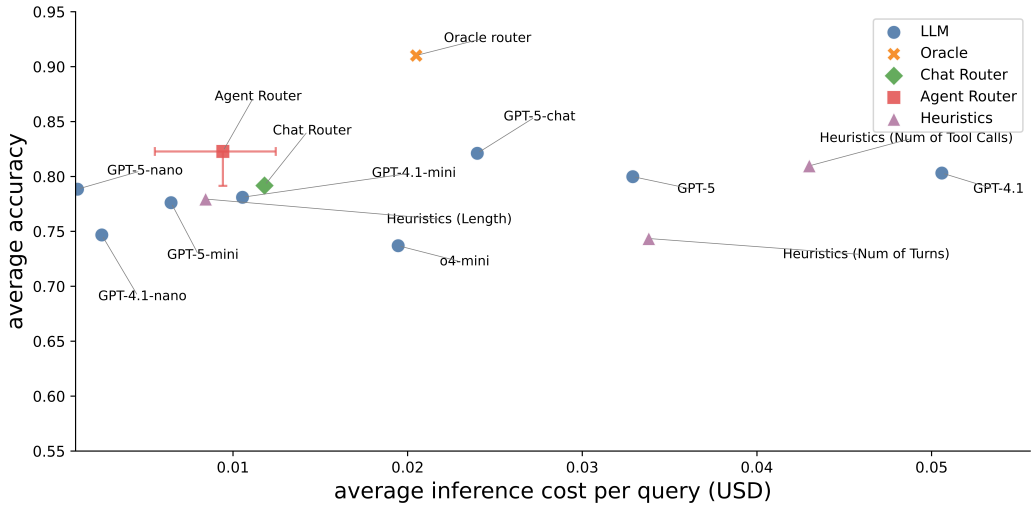

Figure 12: Accuracy–cost Pareto plot with probabilistic correctness labels (1,224 test examples). The agent router fine-tuned on soft probability labels (■, with seed-range error bars) achieves 82.28% accuracy.

Implications and future work. While this preliminary investigation does not demonstrate a clear benefit of probabilistic labels over binary labels, the approach remains promising for settings where:

- Model responses are highly stochastic (e.g., complex multi-step reasoning tasks where correctness varies significantly across runs).
- The cost of incorrect routing is high, and a confidence-calibrated router that can say "this model answers correctly only 60% of the time" would enable risk-aware fallback strategies.
- New evaluation data is expensive to acquire, and soft labels extract more information per evaluation run.

A larger-scale investigation with the full dataset and a proper cost-controlled comparison is an important direction for future work.

Q MIRT-DistilBERT baseline

To assess whether an alternative routing architecture that explicitly models per-LLM capabilities could outperform our multi-label classification approach, we implement and evaluate a router based on **Multidimensional Item Response Theory** (MIRT) [30]. MIRT originates in psychometrics, where it models the probability that a test-taker with latent ability $\boldsymbol{\theta}$ answers an item with discrimination $\boldsymbol{a}$ and difficulty b correctly. Applied to LLM routing, each candidate model plays the role of a test-taker and each query plays the role of a test item.

Q.1 Architecture

The MIRT router uses a two-stage architecture:

Stage 1: embedding extraction (frozen). Both query texts and LLM profile descriptions are encoded by a frozen DistilBERT model into 768-dimensional mean-pooled, L2-normalised embeddings. Each LLM is represented by a short natural-language profile describing its release date, capabilities, and intended use case (8 profiles in total). Query embeddings are computed from the concatenation of the user message and tool signatures.

Stage 2: MIRT head (fine-tuned). A lightweight MIRT head projects query and LLM embeddings into a shared K-dimensional latent space (we use K=25) via three linear projections:

$$\boldsymbol{\theta} = W_\theta \, \mathbf{e}_{\text{llm}} \qquad \text{(LLM ability)} \tag{1}$$

$$\boldsymbol{a} = \text{softplus}\big(W_a \, \mathbf{e}_{\text{query}}\big) \qquad \text{(query discrimination)} \tag{2}$$

$$b = W_b \, \mathbf{e}_{\text{query}} \qquad \text{(query difficulty)} \tag{3}$$

The predicted probability that LLM m answers query q correctly follows the 2PL (two-parameter logistic) IRT response function:

$$P(m, q) = \sigma\Big(\sum_{k=1}^{K} a_k \, \theta_k - b\Big)$$

At inference time, the router scores all eight LLMs for a given query and selects the cheapest one whose predicted probability exceeds a threshold (θ=0.5), falling back to argmax if none qualifies—the same cost-aware selection rule used by our multi-label classifier.

Q.2 Training differences from our multi-label router

Table 33 summarises the key differences between the MIRT router and our DistilBERT multi-label classifier.

Table 33: Architectural and training differences between our multi-label DistilBERT router and the MIRT-DistilBERT baseline.

Aspect	Our Router	MIRT
Encoder	Fine-tuned end-to-end	Frozen (embeddings only)
Head	8 independent sigmoids	Shared IRT 2PL head
Training data format	Multi-label per query	Pairwise (query, LLM, 0/1)
Parameters trained	66M (full model)	$\sim$59K (3 linear layers)
LLM representation	Fixed output heads	Embedded profile text
Learning rate	5×10^{-5}	5×10^{-3}
Weight decay	0.01	0.0
Optimizer	AdamW	Adam
Input representation	Compressed token packing	Vanilla concatenation

The MIRT architecture has two potential advantages over fixed-head classification: (i) it can theoretically generalise to *new* LLMs at test time by computing an embedding from their profile text, without retraining; and (ii) its *trainable* parameter count is orders of magnitude smaller ($\sim$59K vs. 66M), since only the three projection matrices are learned while the encoder remains frozen. Note, however, that the frozen encoder is still required at inference time, so the total model size is comparable; the advantage is reduced fine-tuning cost, not a smaller deployment footprint.

Q.3 Results

Table 34 compares the MIRT router against our DistilBERT multi-label classifier on the same validation set (12,267 examples) used for seed selection, evaluated across 20 random seeds.

Table 34: Routing accuracy and cost comparison: our DistilBERT multi-label router vs. MIRT-DistilBERT (validation set, 12,267 examples, 20 seeds). Best-seed accuracy is reported with mean $\pm$ std across seeds in parentheses.

Router	**Accuracy (%)**	**Avg Cost** (10^{-4} **\$**)
DistilBERT (ours)	**82.94** ($\pm$0.41)	6.8
MIRT-DistilBERT	81.64 ($\pm$0.37)	**5.2**
Gap	-1.30 pp	-23%

The MIRT router achieves a best-seed accuracy of **81.64%** (mean 80.79%, std $\pm$0.37 pp), which is **1.30 percentage points below** Switchcraft's best seed (82.94%) and 1.10 pp below its mean (81.89%). The MIRT router does achieve a slightly lower average cost per query (5.2 vs. 6.8×10^{-4} \$), indicating that it routes more aggressively to cheaper models—but at the expense of accuracy.

Q.4 Discussion

Why does MIRT underperform? We identify two likely factors:

1. **Frozen encoder.** Our multi-label router fine-tunes all 66M DistilBERT parameters end-to-end, allowing the encoder to learn task-specific representations for agentic function-calling queries. The MIRT router uses frozen embeddings from a general-purpose pre-trained model, which may not capture the fine-grained distinctions (e.g., JSON structure validity, tool-schema compliance) that matter for routing.
2. **Vanilla tokenization.** The MIRT router uses simple text concatenation rather than our compressed token-packing strategy (Section 3.1), which prioritises the most recent user turn and tool signatures within the 512-token budget. The ablation in Appendix K shows that token packing alone contributes 1.66 pp of accuracy; this accounts for most of the observed gap.

MIRT's advantage: extensibility. The MIRT architecture represents LLMs via their profile embeddings rather than fixed output heads, which in principle allows zero-shot routing to new models by simply providing a profile description. Our multi-label classifier requires retraining when the model pool changes (Section 5). However, since MIRT's accuracy already trails Switchcraft by 1.3 pp even on the *known* model pool, its zero-shot performance on unseen models—which would lack fine-tuning signal entirely—is unlikely to be competitive in practice without further architectural improvements (e.g., unfreezing the encoder or adopting compressed tokenization).

Implications for router design. This comparison supports our design choice of end-to-end fine-tuning with a simple multi-label head over a more structured IRT formulation. The expressiveness gained by fine-tuning the full encoder outweighs the theoretical elegance of the IRT framework in our setting, where the model pool is fixed and retraining is inexpensive (30 epochs on a single GPU).

Q.5 Deviations from the original IRT-Router

Table 35 lists the differences between our MIRT-DistilBERT implementation and the original IRT-Router [30], along with the rationale for each deviation. All deviations are motivated by the goal of an apples-to-apples comparison with our multi-label router: we isolate the effect of the *routing architecture* (multi-label classification vs. IRT head) by holding the encoder, data, and evaluation protocol constant.

In summary, our implementation faithfully reproduces the MIRT-Router's core architecture (embedding $\rightarrow$ linear projections $\rightarrow$ 2PL response function $\rightarrow$ BCE loss) while substituting the encoder, data domain, model pool, and routing rule to match our experimental setup. This design ensures that the

Table 35: Deviations from the original IRT-Router [30] and justification. Each change is made to enable a controlled comparison with our DistilBERT multi-label router on the same data and model pool.

Aspect	**Original IRT-Router**	**Our MIRT impl.**	**Justification**
Embedding model	BERT-base-uncased (110M)	DistilBERT-base-uncased (66M)	Same encoder as our multi-label router; isolates the IRT head as the only architectural variable.
Task domain	Chat completion (12 datasets, LLM-as-judge scoring)	Agentic function calling (14 datasets, AST scoring)	Our evaluation setting for direct comparison.
Candidate LLMs	20 LLMs (GPT-4o, Llama-3.1, etc.)	8 LLMs (GPT-5.x family + Qwen + Kimi)	Same model pool as our main experiments for direct comparison.
Routing decision	$\arg\max_j[\alpha \cdot \hat{P}(q, M_j) - \beta \cdot C(M_j)]$ with fixed per-model cost $C \in [0, 1]$	Cheapest LLM above threshold 0.5; fallback to argmax	Matches Switchcraft's selection rule; uses profiled per-query cost (Section 4.8) rather than fixed per-model pricing.
Cost model	Fixed output-token price normalised to $[0, 1]$	Profiled realised cost per query ($)	Accounts for model chattiness; same cost model as Switchcraft.
Warm-up mechanism	KNN-based embedding interpolation (k=5) for cold-start queries	Not used	Our test set is drawn from the same distribution as training (stratified split), so cold-start is not the primary concern; omitting warm-up isolates the IRT head's contribution.
Learning rate	0.002 (Adam, batch 512)	0.005 (Adam, batch 16)	Tuned on our validation set; smaller batch size matches our hardware setup (single GPU).
Latent dims (K)	25	25	Same as original.
Epochs	Not specified (early stopping implied)	30 (best epoch on val)	Matches our multi-label router training duration.
NIRT variant	Also evaluated (uses predefined ability categories + neural network)	Not implemented	MIRT and NIRT perform similarly in the original paper; MIRT is simpler and sufficient to test the IRT hypothesis.

1.30 pp accuracy gap we observe reflects a genuine limitation of the frozen-encoder IRT formulation relative to end-to-end fine-tuning, rather than an artifact of mismatched evaluation conditions.

R Additional limitations and design considerations

Per-turn correctness vs. end-to-end task success. Our evaluation scores each turn independently via AST matching, but does not measure end-to-end agent task completion under environment dynamics. Extending to trajectory-level success on execution environments such as τ-bench [35] or SWE-bench [13] is future work.

Cost-model assumptions. We use list-price API rates for all models. Self-hosted deployments would substitute amortized GPU-hour costs for open-weight models, which can shift absolute numbers; the relative cost ordering is robust to such substitutions.

Per-turn routing and prompt caching. Switchcraft selects a model independently at each turn, so consecutive turns may be served by different models. In stateless API deployments, the full conversation is re-transmitted regardless, so switching models does not increase billed tokens. However, switching forfeits prompt-caching discounts (typically 50% off repeated prefixes) and increases time-to-first-token. A production deployment could constrain routing to per-conversation granularity or incorporate caching discounts into cost-aware selection.

Reasoning effort and cascading. Many recent LLMs expose a configurable reasoning effort (e.g., low/medium/high). A natural extension is for the router to select not only the model but also the reasoning level, and its associated cost. Similarly, multi-model cascading (trying a cheap model first and falling back to an expensive one on failure) could further reduce average cost.