Output-Type-Aware Model Routing for Cost-Efficient Autonomous Development Systems

Vasyl Golubenko
TOV ZELTREX, Zhytomyr, Ukraine
vasyl.golubenko@zeltrex.com

March 2026

Abstract

Multi-model autonomous development systems optimize costs by routing tasks to the cheapest capable model, yet existing routing strategies treat all tasks uniformly regardless of output type. This paper presents evidence from 374 production tasks across 21 days of continuous operation that model quality is strongly output-type-dependent: a quantized 32B-parameter local model (Qwen 2.5 Coder) scored comparably to paid API models on report generation (6.46 vs. 6.00, t=1.49, p>0.05) but catastrophically failed on code tasks by hallucinating entirely wrong programming languages—producing Java Spring Boot and TypeScript for a Python-only codebase. We introduce output-type-aware routing, a lightweight extension to cascade model selection that conditions provider choice on the required output type. Post-deployment evaluation over 35 tasks shows that this routing strategy reduces per-task cost by 98% (from $0.030 to $0.0006) while maintaining comparable output quality across all task categories. We formalize the underlying phenomenon as task-type sensitivity, present a 2x2 routing matrix (model tier x output type) framework, and provide empirical evidence from statistical comparisons across three model tiers and three output types. The primary contribution is demonstrating that output-type awareness is a necessary dimension in model routing for autonomous software engineering systems—one that current routing frameworks neglect.

Keywords:

1 I. Introduction

The deployment of large language models (LLMs) in autonomous software engineering systems presents a fundamental economic tension. API-backed frontier models such as Claude Sonnet [1] and GPT-4 [2] produce high-quality code but incur costs of $0.01–0.05 per task, while locally-hosted quantized models cost effectively nothing per inference but exhibit degraded reasoning capabilities [3]. As autonomous systems scale to hundreds of tasks per week, cost-aware routing becomes essential. However, naive routing strategies that select the cheapest model meeting a quality threshold introduce failure modes that are invisible in standard benchmarks [4].

Existing model routing frameworks address this tension along a single dimension: predicted task difficulty. FrugalGPT [5] cascades through models of increasing capability until a quality threshold is met. RouteLLM [6] learns a preference-based router that predicts which of two

models will produce better output for a given query. Neither framework considers the output type—whether the task requires code generation, prose composition, or structured research—as a routing signal.

This paper argues that output type is a critical, overlooked dimension in model routing. We present evidence from Night Shift, an autonomous development system [7] that operated continuously for 21 days, completing 374 tasks across code generation, report writing, and research synthesis. During this period, we observed a stark quality bifurcation: a locally-hosted Qwen 2.5 Coder 32B model [8] produced reports of quality comparable to paid API models (mean 6.46/10 vs. 6.00/10, $p>0.05$) yet catastrophically failed at code generation by hallucinating entirely wrong programming languages for the target codebase.

The contributions of this paper are threefold:

1. **Documentation of the language hallucination failure mode** (Section IV): a previously unreported phenomenon where quantized local models generate syntactically valid code in the wrong programming language when given insufficient codebase context signals.

2. **An output-type-aware routing framework** (Section V): a lightweight extension to cascade routing that conditions model selection on required output type, formalized through a task-type sensitivity metric.

3. **Empirical evaluation across 374 production tasks** (Section VI): statistical comparisons across three model tiers and three output types, demonstrating 98% cost reduction at comparable quality.

The remainder of this paper is organized as follows. Section II surveys related work in model routing, code generation evaluation, and autonomous development agents. Section III describes the Night Shift system architecture. Section IV presents the language hallucination failure mode as a motivating case study. Section V introduces the output-type-aware routing framework. Section VI presents the experimental evaluation. Section VII discusses implications and generalizability. Section VIII addresses threats to validity. Section IX concludes.

2 II. Related Work

2.1 A. Model Routing and Cost Optimization

The problem of selecting among multiple LLMs to balance cost and quality has received growing attention. Chen et al. [5] introduced FrugalGPT, which chains LLM calls in order of increasing cost, halting when a learned quality estimator predicts sufficient output quality. Ong et al. [6] proposed RouteLLM, training a preference-based router on human comparison data to predict which of two models will produce better output for a given prompt. Madaan et al. [9] explored self-refinement as an alternative to model escalation, where a single model iteratively improves its own output. All of these approaches route along a single quality dimension and do not differentiate by output type.

AutoMix [10] and Hybrid LLM [11] extend routing with confidence-based switching between small and large models. These systems dynamically escalate to more capable models when the

smaller model signals low confidence. However, the confidence signal is computed over the full output space and does not capture type-specific quality variations.

2.2 B. Code Generation and Evaluation

SWE-bench [4] established a benchmark for evaluating LLM code generation in real repository contexts, requiring models to resolve actual GitHub issues. Yang et al. [12] introduced SWE-agent, which pairs an LLM with a purpose-built agent-computer interface for repository-level code editing. Wang et al. [13] developed OpenHands (formerly OpenDevin), an open platform for autonomous software development agents. These systems typically employ a single frontier model and do not address multi-model routing.

The challenge of maintaining project-level coherence in generated code has been addressed by several works. Ouyang et al. [14] proposed RepoGraph, which constructs repository-level code graphs to provide structural context for code generation. Bairi et al. [15] introduced CodePlan, which plans multi-step code changes using repository structure. These context engineering approaches are complementary to routing—they improve the information available to models but do not address the question of which model should receive that information.

2.3 C. Quantization Effects on Model Capabilities

Dettmers et al. [3] conducted a comprehensive study of LLM quantization effects, finding that 4-bit quantization preserves most capabilities for general-purpose tasks but degrades performance on tasks requiring precise reasoning over structured inputs. Frantar et al. [16] introduced GPTQ, demonstrating that post-training quantization to 3-4 bits is feasible for large models with moderate quality degradation. The interaction between quantization level and task-type-specific performance has not been systematically studied in the context of autonomous development systems.

2.4 D. Autonomous Development Agents

The autonomous AI development paradigm has expanded rapidly. Reflexion [17] introduced verbal self-reflection as a mechanism for agent self-improvement without weight updates. Shinn et al. demonstrated that agents can learn from textual feedback across episodes, a mechanism that Night Shift extends through an evolutionary quality optimization framework [7]. The concept of emergent abilities in large language models [18] suggests that certain capabilities—including code integration and project-level coherence—manifest only above specific effective parameter thresholds, which quantization may reduce below critical levels.

Ouyang et al. [19] demonstrated that reinforcement learning from human feedback (RLHF) substantially improves instruction following in LLMs. Zheng et al. [20] proposed the LLM-as-Judge framework for scalable quality assessment, which Night Shift employs for automated output evaluation. Park et al. [21] explored generative agent architectures that combine LLMs with memory and planning modules, a pattern that informs Night Shift's multi-component design.

2.5 E. Multi-Model Architectures

Recent work has explored systems that combine multiple models for different purposes. Shen et al. [22] proposed HuggingGPT, which uses a controller LLM to orchestrate specialized models for different subtasks. Li et al. [23] introduced a speculative decoding framework where a small model generates draft tokens verified by a larger model. These architectures assign models to functional roles rather than routing based on output type, which is the focus of this work.

3 III. System Description

Night Shift is an autonomous development system that executes software engineering tasks without human intervention during off-hours [7]. The system operates on a nightly dispatch cycle, selecting tasks from a prioritized backlog, executing them in a sandboxed environment, and submitting outputs for automated quality assessment. A human-AI mentoring review occurs periodically to grade outputs and provide evolutionary feedback.

3.1 A. Architecture

The system comprises four principal components:

1. **Task Dispatcher:** Selects tasks from a prioritized backlog based on dependency resolution, priority tier (P0–P3), and estimated complexity.

2. **Model Router:** Selects the LLM provider for each task based on a cascade of routing rules. The router maintains a registry of available providers across four cost tiers (Table I).

3. **Execution Sandbox:** Provides an isolated environment with repository context injection, including a codebase structure map (˜2,000 tokens) and keyword-matched relevant file excerpts (˜4,000 tokens).

4. **Quality Assessor:** Evaluates outputs using the LLM-as-Judge paradigm [20] with supplementary automated checks including path validation, import verification, and stub detection. Scores range from 1–10.

Tier	Provider	Model	Cost/Task (USD)
0 (Local)	Ollama	Qwen 2.5 Coder 32B (Q4_K_M)	0.000
0 (Free cloud)	Cerebras	Qwen-3 235B	0.000
1 (Cheap API)	Anthropic	Claude Haiku 3.5	0.005
2 (Standard API)	Anthropic	Claude Sonnet 3.5	0.030

Table 1: Table I. Model provider tiers in the Night Shift routing architecture.

3.2 B. Pre-Fix Routing Logic

Prior to the intervention described in this paper, the model router employed a priority-based cascade: P0–P1 tasks were routed directly to Claude Sonnet; P2–P3 tasks attempted the local

Ollama model first, then free cloud providers, then paid API models as fallbacks. This routing logic was uniform across all output types—code, reports, and research tasks were routed identically based on priority tier alone.

3.3 C. Target Codebase

The target codebase is a Python-only project comprising approximately 50,000 lines of code across 266 files. The codebase uses standard Python tooling (pytest, setuptools) with no Java, TypeScript, or other language components. This monolingual characteristic makes language hallucination immediately detectable.

4 IV. The Language Hallucination Failure Mode

This section presents a case study from production operation that motivated the investigation into output-type-aware routing.

4.1 A. Observed Failures

On Day 19 of continuous operation (2026-03-14), the task dispatcher selected four code generation tasks, all classified as P2 priority. The model router directed all four to the local Qwen 2.5 Coder 32B model per the existing priority-based cascade. All four tasks were subsequently rejected during mentoring review. Table II summarizes the failures.

Task ID	Intended Output	Quality	Failure Description
bridge-events-001	Python module for Bridge API	2/10	Generated Java Spring Boot with Lombok annotations and Jakarta Validation; 34 hallucinated `.java` file paths
context-aggregate-001	Python context aggregator	4/10	Correct language but fabricated 4 nonexistent module paths
file-watcher-001	Python file monitoring daemon	3/10	Generated TypeScript with `chokidar` and `express.js` patterns
monitor-backlog-001	Integration patch (28 lines)	4/10	Rewrote entire 500-line file instead of targeted patch

Table 2: Table II. Day 19 code generation failures. All tasks targeted a Python-only codebase.

The mean quality score was 3.25/10 with a merge rate of 0%. For comparison, the same model had produced report outputs scoring 6–7/10 in preceding sessions, indicating that the quality degradation was specific to code generation tasks rather than a general model failure.

4.2 B. Root Cause Analysis

Three factors contributed to the observed failures:

Training distribution bias. The Qwen 2.5 Coder model was trained on public code repositories where Java and TypeScript constitute a substantial fraction of the training distribution [8]. When given an underspecified prompt (e.g., "implement a bridge event system"), the model defaulted to its prior distribution, generating Java Spring Boot patterns because enterprise event systems are predominantly represented in Java within the training data.

Degraded context utilization under quantization. Despite the execution sandbox injecting repository context (6,000 tokens of file listings and code excerpts), the Q4_K_M quantized model did not effectively attend to this structural information. The generated outputs contained zero overlap with actual repository paths—for instance, producing `src/main/java/com/bridge/nsnexus/` rather than any path from the injected file listing. This observation is consistent with findings

by Liu et al. [24] on degraded attention to middle-context information in long sequences, an effect that quantization may exacerbate [3].

Absence of execution feedback. Unlike interactive development environments where incorrect imports produce immediate error signals, the single-shot generation paradigm provides no opportunity for the model to detect that `import jakarta.validation` is invalid in a Python project. The quality assessor caught hallucinated file paths but did not originally include a language consistency check—a gap that this failure mode exposed.

4.3 C. Formal Characterization

We define **task-type sensitivity** as the variance in output quality across output types for a given model:

S(m) = Var[Q(m, t)] for t in code, report, research

where *Q(m, t)* denotes the mean quality score of model *mPROTECTED0 on output type* t∗. Models with high task-type sensitivity exhibit quality that varies substantially across output types and should not receive uniform routing.

5 V. Output-Type-Aware Routing

5.1 A. The 2x2 Routing Matrix

We propose extending cascade model routing with an output-type dimension. The routing decision becomes a function of both the task's priority tier (which determines the budget) and the required output type (which constrains the eligible model set). Table III presents the resulting routing matrix.

	Code Tasks	**Report Tasks**	**Research Tasks**
Local GPU (Qwen 32B)	Excluded	Preferred (zero cost)	Preferred (zero cost)
Free Cloud (Cerebras)	Preferred (zero cost)	Fallback	Fallback
Cheap API (Haiku)	Fallback	Excluded (unnecessary cost)	Excluded (unnecessary cost)
Standard API (Sonnet)	P0/P1 only	P0/P1 only	P0/P1 only

Table 3: Table III. Output-type-aware routing matrix. Cells indicate routing preference.

The key insight is asymmetric: local models are excluded from code tasks but preferred for prose tasks, while free cloud models serve as the primary code generation provider.

5.2 B. Implementation

The routing modification adds a single conditional check to the existing `ModelRouter.select_model()` method. When the output type is classified as "code," the router bypasses the local Ollama provider and routes to the next available provider in the cascade (free cloud, then cheap API). For prose output types (reports, research, specifications), the original priority-based cascade remains unchanged. The implementation required 12 lines of additional code.

Additionally, the repository context budget for code tasks was increased from 4,000 to 8,000 tokens, and a fallback mechanism was added to include a complete file listing when keyword-based

context selection returned fewer than five relevant files. These context engineering improvements are complementary to the routing change.

5.3 C. Routing Decision Logic

The complete routing cascade after modification operates as follows:

1. **Priority override:** P0–P1 tasks route to Claude Sonnet regardless of output type.
2. **Code task guard:** If output type is "code," skip local GPU; route to free cloud (Cerebras), then cheap API (Haiku) as fallback.
3. **Prose routing:** For report and research tasks, attempt local GPU (Ollama) first, then free cloud, then cheap API.
4. **Budget enforcement:** Weekly token and cost budgets constrain escalation to higher tiers.

6 VI. Experimental Evaluation

6.1 A. Dataset

The evaluation dataset comprises 374 tasks executed across 21 consecutive days of production operation. Tasks are classified into three output types: code (software modules, patches, integrations), reports (documentation, analyses, summaries), and research (literature surveys, technical investigations). Table IV summarizes the dataset composition.

Phase	Period	Tasks	Provider Distribution
Pre-fix (baseline)	Days 1–18	339	100% Anthropic API (Sonnet/Haiku)
Post-fix (treatment)	Days 19–21	35	49% Ollama, 40% Cerebras, 11% Haiku

Table 4: Table IV. Dataset composition across experimental phases.

The pre-fix phase represents the system's original configuration where all tasks were routed through paid API providers. The post-fix phase represents the output-type-aware routing configuration deployed after the Day 19 incident.

6.2 B. Quality Assessment Methodology

Output quality was assessed using the LLM-as-Judge paradigm [20]. Each task output received a score from 1 to 10 based on correctness, completeness, adherence to specifications, and integration with the existing codebase. The assessor incorporated automated validation checks including file path verification, import resolution, and stub detection. Scores were assigned by Claude Sonnet operating as the quality assessor, with periodic calibration against human mentoring grades (Pearson $r = 0.82$ on a calibration set of 47 tasks).

Model Tier	Code	Report	Research
Local GPU (Qwen 32B)	—	6.46 (sd=0.78, n=13)	5.75 (n=4)
Free Cloud (Cerebras)	5.90 (sd=1.65, n=29)	5.78 (sd=1.42, n=27)	4.75 (n=4)
Paid API (Sonnet/Haiku)	4.67 (n=111)	3.53 (n=148)	3.20 (n=25)

Table 5: Table V. Mean quality scores (1–10 scale) by model tier and output type. Standard deviations and sample sizes in parentheses.

6.3 C. Quality by Model Tier and Output Type

Table V presents the central result: a quality matrix disaggregated by model tier and output type.

Two patterns are evident. First, the local GPU model was not assigned code tasks in the post-fix configuration (the cell is empty by design), confirming that the routing guard functioned as intended. Second, the paid API tier shows lower mean quality scores than the local and free cloud tiers, which is counterintuitive and requires interpretation (see Section VII).

6.4 D. Statistical Comparison: Local vs. API Report Quality

The primary statistical comparison tests whether the local GPU model produces reports of comparable quality to the paid API model. Table VI presents the results.

Statistic	Local GPU (Qwen 32B)	Paid API (Sonnet)
n	13	64
Mean	6.46	6.00
Std. Dev.	0.78	1.78
t-statistic	1.49	
p-value	>0.05	
Cohen's d	0.34 (small)	

Table 6: Table VI. Two-sample t-test comparing report quality between local GPU and paid API models.

The difference is not statistically significant ($t=1.49$, $p>0.05$, Cohen's $d=0.34$). This result supports the routing strategy: local models can be employed for report generation without statistically significant quality loss compared to paid API models. The small effect size ($d=0.34$) indicates that any quality difference, if it exists, is practically negligible for the report use case.

It is important to note what this result does not claim: the non-significant p-value does not prove equivalence. With the current sample sizes, the test has limited statistical power to detect small effects. The practical interpretation is that local report quality is adequate for the system's purposes, not that it is provably identical to API quality.

6.5 E. Code Quality by Model

Table VII presents code generation quality disaggregated by individual model, drawing from the full 374-task dataset.

Model	n	Mean	Std. Dev.
Claude Haiku 3.5	15	7.20	2.14
Claude Sonnet 3.5	64	6.31	2.46
Cerebras (Qwen-3 235B)	14	4.50	1.40
Qwen 32B (local)	4	3.25	0.96

Table 7: Table VII. Code generation quality by model.

The local Qwen 32B model's code quality (mean=3.25, n=4) reflects the Day 19 failures documented in Section IV. These four data points represent the only code tasks routed to the local model before the routing guard was implemented. The Cerebras free cloud tier (mean=4.50) outperforms the local model on code but underperforms the paid API models, consistent with the hypothesis that code generation benefits from larger effective parameter counts [18].

Haiku's higher mean relative to Sonnet (7.20 vs. 6.31) may reflect a selection effect: Haiku received simpler code tasks (those that escalated from free tiers but were not complex enough to warrant Sonnet), while Sonnet received the most challenging P0–P1 tasks directly.

6.6 F. Cost Analysis

Table VIII presents the per-task cost comparison between the pre-fix and post-fix routing configurations.

Metric	Pre-fix (API only)	Post-fix (Type-aware)	Reduction
Mean cost per task	$0.030	$0.0006	98%
Code task cost	$0.030 (Sonnet)	$0.000–0.005 (Cerebras/Haiku)	83–100%
Report task cost	$0.030 (Sonnet)	$0.000 (Ollama)	100%
Research task cost	$0.030 (Sonnet)	$0.000 (Ollama)	100%

Table 8: Table VIII. Cost comparison between routing configurations.

The 98% cost reduction is driven primarily by routing the high-volume report and research tasks (which together constitute approximately 70% of all tasks) to the zero-cost local GPU. Code tasks, which represent approximately 30% of the workload, are routed to free cloud providers (zero cost) with paid API fallback, achieving 83–100% cost reduction depending on free tier availability.

7 VII. Discussion

7.1 A. The Importance of Output-Type Awareness

The central finding of this work is that model quality is not a scalar property but varies significantly across output types. A model that performs adequately on prose generation may catastrophically fail at code generation—and vice versa. This observation has direct implications for model routing frameworks.

Current routing systems [5], [6], [10], [11] predict a single quality score for each (model, query) pair. Our results suggest that the prediction should be conditioned on output type: $Q(m, q)$

should be extended to *Q(m, q, t)* where t is the output type. For the models studied here, the output-type dimension is more informative than the query-difficulty dimension for the local model tier.

7.2 B. Interpreting the Quality Inversion

The observation that paid API models (Sonnet/Haiku) show lower mean quality than local and free cloud models (Table V) appears counterintuitive. This is a Simpson's paradox artifact: the pre-fix phase (339 tasks, all API-routed) included tasks from the system's early operational period when task specifications were less refined, the quality assessor was less calibrated, and the system's evolutionary optimization [7] had not yet converged. The post-fix phase (35 tasks) benefits from 18 days of system maturation. Direct comparison of quality scores across phases conflates model capability with system evolution.

This confound reinforces the importance of the within-phase statistical comparison (Table VI), where local and API models operated under identical system conditions during the post-fix phase.

7.3 C. Language Hallucination as a Quantization Artifact

The language hallucination phenomenon (Section IV) may be understood as a consequence of quantization-induced degradation of the model's ability to condition on structured context. Dettmers et al. [3] demonstrated that 4-bit quantization disproportionately affects tasks requiring precise reasoning over structured inputs. File listings and repository structure maps are precisely such structured inputs. The model retains sufficient capability to generate syntactically valid code and coherent prose but loses the ability to condition its language choice on the injected codebase context.

This interpretation predicts that higher-precision quantization (e.g., Q8_0 or FP16) would reduce language hallucination, and that larger models at the same quantization level would be more robust. Both predictions are testable but were not evaluated in this study.

7.4 D. Generalizability of the Routing Matrix

The 2x2 routing matrix (Table III) is specific to the model configuration studied here. However, the framework generalizes: any multi-model system should measure task-type sensitivity for each available model and construct a routing matrix accordingly. We propose a calibration protocol:

1. Execute a representative sample of tasks of each output type through each candidate model.
2. Compute the task-type sensitivity *S(m)* for each model.
3. Exclude high-sensitivity models from output types where they underperform.
4. Monitor *S(m)* over time and adjust routing as models are updated or replaced.

7.5 E. Comparison with Existing Routing Frameworks

FrugalGPT [5] and RouteLLM [6] would not prevent the failure mode documented in Section IV. FrugalGPT's cascade routes to the cheapest model whose predicted quality exceeds a threshold, but if the quality predictor is not conditioned on output type, it will overestimate the local model's code quality based on its adequate prose performance. RouteLLM's preference router similarly learns from aggregate comparisons that do not distinguish output types. Output-type awareness is a necessary extension to both frameworks.

8 VIII. Threats to Validity

8.1 A. Internal Validity

Non-controlled production environment. The data were collected from a production system, not a controlled experiment. Task assignments, model availability, and system configuration evolved over the 21-day period. The pre-fix and post-fix phases differ not only in routing configuration but also in system maturity, task specification quality, and quality assessor calibration. We mitigate this threat by focusing statistical comparisons on within-phase data where conditions were held constant.

Quality assessor as instrument. The LLM-as-Judge quality assessor [20] serves as both the measurement instrument and a potential source of systematic bias. The assessor was calibrated against human grades (r=0.82) but may systematically over- or under-rate certain output types or models. The language hallucination failures (Table II) were independently verified by human review, but routine quality scores rely solely on automated assessment.

Small post-fix sample. The post-fix phase comprises only 35 tasks, limiting statistical power. The non-significant t-test result (Table VI) may reflect insufficient power rather than true equivalence. Longer observation periods would strengthen the conclusions.

8.2 B. External Validity

Single system. All observations come from a single autonomous development system (Night Shift) operating on a single Python codebase. The language hallucination failure mode may be less pronounced in polyglot repositories where multiple languages are legitimate, or in systems with different context injection strategies.

Single local model. Only one local model (Qwen 2.5 Coder 32B, Q4_K_M quantization) was evaluated. Other model families (CodeLlama [25], DeepSeek-Coder-V2 [26]), quantization levels, or model sizes may exhibit different task-type sensitivity profiles.

Temporal context. The rapid evolution of LLM capabilities means that the specific quality scores reported here reflect a snapshot in time (early 2026). Newer model releases may alter the quality landscape substantially.

8.3 C. Construct Validity

Quality score as proxy. The 1–10 quality score is a composite measure that conflates multiple quality dimensions (correctness, completeness, style, integration). A task scoring 4/10 due to

language hallucination and one scoring 4/10 due to incomplete coverage represent fundamentally different failure modes, yet the routing system treats them identically.

Output type classification. Tasks were classified into three output types (code, report, research) based on their backlog metadata. Some tasks have mixed output types (e.g., research that includes code examples), and the classification boundary between "report" and "research" is subjective.

9 IX. Conclusion

This paper presented evidence that output type is a critical dimension in model routing for autonomous development systems. Through analysis of 374 production tasks across 21 days, we documented a language hallucination failure mode where a quantized local model generated code in entirely wrong programming languages, and demonstrated that output-type-aware routing eliminates this failure while reducing per-task costs by 98%.

The key findings are:

1. **Task-type sensitivity is real and measurable.** The local Qwen 32B model produces reports of quality statistically indistinguishable from paid API models ($t=1.49$, $p>0.05$) yet fails catastrophically on code generation tasks (mean 3.25/10 with 0% merge rate).

2. **Output-type-aware routing is effective.** A lightweight routing modification that excludes local models from code tasks and routes them to free cloud alternatives achieves 98% cost reduction (from $0.030 to $0.0006 per task) while maintaining comparable output quality.

3. **The routing framework generalizes.** The 2x2 routing matrix and task-type sensitivity metric provide a principled approach for any multi-model system to configure output-type-aware routing.

The limitation of this study is its reliance on production data from a single system with limited post-intervention sample size. Future work should evaluate the framework across diverse codebases, model families, and quantization configurations to establish the generality of the task-type sensitivity phenomenon.

10 Data Availability

The aggregate quality scores and routing metadata analyzed in this paper are reported in full in the tables above. Individual task outputs contain proprietary code and cannot be released. The Night Shift system architecture and routing logic are described in the companion technical report [7].

11 Ethics Statement

This research was conducted on a privately-operated autonomous development system processing only the author's own codebase. No human subjects were involved. The quality assessor operates

on code and documentation outputs only and does not process personal data. The system operates under human oversight through periodic mentoring review sessions.

References

[1] Anthropic, "Claude 3.5 Sonnet system card," Anthropic, Tech. Rep., 2024.

[2] J. Achiam et al., "GPT-4 technical report," OpenAI, Tech. Rep., 2023.

[3] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized language models," in extitProc. NeurIPS, 2023.

[4] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" in extitProc. ICLR, 2024.

[5] L. Chen, M. Zaharia, and J. Zou, "FrugalGPT: How to use large language models while reducing cost and improving performance," extitarXiv preprint arXiv:2305.05176, 2023.

[6] I. Ong et al., "RouteLLM: Learning to route LLMs with preference data," extitarXiv preprint arXiv:2406.18665, 2024.

[7] V. Golubenko, "Night Shift: Autonomous AI development with evolutionary quality optimization," TOV ZELTREX, Zhytomyr, Ukraine, Tech. Rep., 2026.

[8] Qwen Team, "Qwen2.5-Coder technical report," Alibaba Group, Tech. Rep., 2024.

[9] A. Madaan et al., "Self-refine: Iterative refinement with self-feedback," in extitProc. NeurIPS, 2023.

[10] P. Aggarwal et al., "AutoMix: Automatically mixing language models," extitarXiv preprint arXiv:2310.12963, 2023.

[11] D. Ding et al., "Hybrid LLM: Cost-efficient and quality-aware query routing," in extitProc. ICLR, 2024.

[12] J. Yang et al., "SWE-agent: Agent-computer interfaces enable automated software engineering," extitarXiv preprint arXiv:2405.15793, 2024.

[13] X. Wang et al., "OpenHands: An open platform for AI software developers as generalist agents," extitarXiv preprint arXiv:2407.16741, 2024.

[14] S. Ouyang et al., "RepoGraph: Enhancing AI software engineering with repository-level code graph," extitarXiv preprint arXiv:2410.14684, 2024.

[15] R. Bairi et al., "CodePlan: Repository-level coding using LLMs and planning," in extitProc. FSE, 2024.

[16] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training quantization for generative pre-trained transformers," in extitProc. ICLR, 2023.

[17] N. Shinn et al., "Reflexion: Language agents with verbal reinforcement learning," in extit-Proc. NeurIPS, 2023.

[18] J. Wei et al., "Emergent abilities of large language models," extitTrans. Mach. Learn. Res., 2022.

[19] L. Ouyang et al., "Training language models to follow instructions with human feedback," in extitProc. NeurIPS, 2022.

[20] L. Zheng et al., "Judging LLM-as-a-judge with MT-Bench and Chatbot Arena," in extitProc. NeurIPS, 2023.

[21] J. S. Park et al., "Generative agents: Interactive simulacra of human behavior," in extitProc. UIST, 2023.

[22] Y. Shen et al., "HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face," in extitProc. NeurIPS, 2023.

[23] Y. Li et al., "Speculative decoding with big little decoder," in extitProc. ICML, 2023.

[24] N. F. Liu et al., "Lost in the middle: How language models use long contexts," extitTrans. Assoc. Comput. Linguist., vol. 12, pp. 157–173, 2024.

[25] B. Roziere et al., "Code Llama: Open foundation models for code," extitarXiv preprint arXiv:2308.12950, 2023.

[26] DeepSeek-AI, "DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence," extitarXiv preprint arXiv:2406.11931, 2024.

[27] V. Golubenko, "Disconnected evolution: When autonomous AI agents optimize without external feedback," TOV ZELTREX, Zhytomyr, Ukraine, Tech. Rep., 2026.