Spec-Driven, SLO-Aware Agents on a Single GPU

1 Introduction

Large language model (LLM) agents are rapidly moving from prototypes to production systems. They now power customer support assistants, internal copilots, analytics dashboards, and workflow automations that call internal tools and APIs. In these settings, an agent is not just "a model" but a *service* that must satisfy explicit contracts and service-level objectives (SLOs): it must emit JSON or other structured outputs that downstream systems can parse, must stay faithful to the underlying data or retrieved context, and must respond within a bounded latency for an acceptable user experience. If an agent emits malformed JSON, downstream ETL jobs or dashboards may fail. If it emits a plausible but unsupported statement about a key metric, decision-makers may act on a hallucination. If it is occasionally fast but has heavy-tailed latency, users perceive the system as unreliable and throughput collapses.

Roadmap. This paper (P1) establishes a stable, single-GPU baseline for contract-first agents; subsequent papers deepen the RL story. P2 turns the same metrics (JSON validity, task success, disagreement, latency) into multi-objective rewards and SLO constraints. P3–P5 study constrained decoding, scheduling, and budgeted self-consistency under those rewards. P6 packages an operational playbook that maps (λ, μ, γ) and decoding settings to deployment levers with observability hooks.

Systems research has long emphasized that users experience *tail latency*, not averages. Dean and Barroso's "The Tail at Scale" [12] (`https://cacm.acm.org/research/the-tail-at-scale/`) showed that even a small fraction of slow requests can dominate perceived quality and limit scalability. Modern LLM-serving work extends this insight to Transformer inference. Runtimes such as vLLM [16] (`https://arxiv.org/abs/2309.06180`), SGLang [21] (`https://arxiv.org/abs/2312.07104`), DeepSpeed-FastGen [20] (`https://arxiv.org/abs/2401.08671`), and Sarathi-Serve [3] (`https://www.usenix.org/system/files/osdi24-agrawal.pdf`) propose kernel and scheduling optimizations—paged KV caches, optimized attention kernels [11, 18], speculative decoding [17] (`https://arxiv.org/abs/2211.17192`)—that improve throughput and reduce tail latency. However, these systems generally treat the model as a black box that emits free-form text and do not reason explicitly about structured contracts or the *semantic* correctness and faithfulness of responses.

In parallel, API providers and local runtimes have introduced *structured output* modes. OpenAI's structured outputs and function-calling interfaces (`https://openai.com/index/introducing-structured-`
Azure OpenAI, and Snowflake Cortex (`https://docs.snowflake.com/en/user-guide/snowflake-cortex/complete-structured-outputs`) expose JSON Schema-like contracts that models are expected to obey. Local stacks such as LM Studio (`https://lmstudio.ai/docs/developer/openai-compat/tools`) and Ollama (`https://docs.ollama.com/capabilities/structured-outputs`, `https://blog.danielclayton.co.uk/posts/ollama-structured-outputs/`) provide OpenAI-compatible endpoints with structured-output options. Libraries like Outlines (`https://dottxt-ai.github.`

io/outlines/), Instructor (https://python.useinstructor.com/integrations/llama-cpp-python/), and llama.cpp grammars (https://github.com/ggml-org/llama.cpp/blob/master/grammars/README.md) compile contracts into grammars or JSON constraints. Recent benchmarks such as JSONSchemaBench and analyses of format constraints [15, 8, 10] (https://arxiv.org/abs/2501.10868, https://arxiv.org/abs/2408.02442, https://arxiv.org/abs/2408.11061) show that constrained decoding substantially improves syntactic validity. Yet these works leave open how structured decoding interacts with serving runtimes, tail latency, and business-level metrics such as task success and SLO adherence.

Beyond format, *faithfulness*—whether an agent's output is supported by the underlying data or retrieved context—has emerged as a central concern. For many commercial agents, the primary risk is not that the model produces ungrammatical text, but that it produces fluent, confident statements that are subtly wrong. RAGAS (Retrieval Augmented Generation Assessment) [14] (https://arxiv.org/abs/2309.15217) proposed automated metrics for retrieval-augmented generation (RAG) that use an LLM judge to score faithfulness and answer relevance. Subsequent work on "atomic fact" evaluation [7] (https://arxiv.org/abs/2408.15171) advocates decomposing responses into fine-grained statements and checking each against the context to better characterize hallucinations. These approaches, however, are mostly experimental and are not tightly tied to structured outputs that feed downstream systems or to latency-aware SLOs.

Cloud providers and tooling ecosystems have started to articulate evaluation methods for agents in production. Google's Agent Development Kit (ADK) and Vertex AI evaluation documentation (https://google.github.io/adk-docs/evaluate/, https://cloud.google.com/vertex-ai/generative-ai/docs/models/evaluation-overview) emphasize evaluating both final response quality and *trajectory* quality (tools invoked, step counts, error handling) using golden sets and task-specific rubrics. Google Cloud's blogs on agent testing and evaluation (https://medium.com/google-cloud/agent-testing-and-
https://cloud.google.com/blog/topics/developers-practitioners/a-methodical-approach-to-agent-e
and the LLM-Evalkit tools (https://github.com/GoogleCloudPlatform/generative-ai/tree/main/tools/llmevalkit, https://cloud.google.com/blog/products/ai-machine-learning/introducing-llm-evalkit) offer practical recipes for building evaluation pipelines. These resources are valuable, but they are high-level and platform-specific: they do not define a concrete, portable test suite that ties together structured outputs, faithfulness metrics, trajectory checks, and tail-latency SLOs for agents running on local backends such as LM Studio, Ollama, or vLLM.

In this work we focus on the regime where a team runs a *single-GPU* agent stack—for example, an RTX 4090 in an on-premise box or a MacBook Pro with Apple Silicon—and serves generic agents via local OpenAI-compatible servers (e.g., LM Studio: https://lmstudio.ai/docs/developer/openai-compat/tools; Ollama: https://docs.ollama.com/capabilities/structured-outputs; vLLM: https://arxiv.org/abs/2309.06180). The agents must emit structured JSON outputs that downstream systems can consume, remain faithful to internal data or retrieved context, and respect p95/p99 latency budgets, all under tight compute and privacy constraints. Practically, this scenario is common: teams want to avoid sending sensitive data to external APIs, want predictable costs, and still need high-quality agent behavior. Crucially, in many of these use cases, *correctness and truthfulness are prioritized over speed*, except where a specific real-time SLA explicitly dominates.

We argue that what is missing is a **contract-first, SLO-aware agent framework** that: (i) treats JSON Schemas and grammars as the primary interface between applications and LLMs; (ii) assesses not only syntactic validity but also task accuracy, statement-level faithfulness, tool and trajectory behavior, stability across runs, and SLO adherence; and (iii) is designed to run on single-GPU local backends rather than assuming large managed clusters. Such a framework should support both *static evaluation* of a given policy and configuration and *dynamic improvement* via reinforcement learning, so that agents can become more reliable over time while staying within latency and cost envelopes.

We make three contributions toward this goal:

- **A contract-first, single-GPU agent architecture.** We introduce a spec-driven agent architecture that treats JSON Schemas and grammars as first-class contracts, compiles them into structured decoding for LM Studio, Ollama, vLLM, and llama.cpp, and integrates bounded validation-and-retry and budgeted self-consistency under explicit wall-clock and token budgets on a single GPU.
- **A unified, configurable evaluation suite for agents.** We design an auditable evaluation framework, configured via a single `criteria.yaml`, that spans six metric families—structure, task accuracy, faithfulness, tools and trajectories, stability, and latency/SLOs—and can be applied unchanged across backends and models. The framework organizes a 100+ test suite, logs all runs to Weights & Biases (W&B) for reproducibility (`https://docs.wandb.ai/models/quickstart`, `https://docs.wandb.ai/models/artifacts`, `https://docs.wandb.ai/models/evaluate-models`), and is designed to capture real-world failure modes that matter to practitioners.
- **A measurement study of decoding modes and backends under SLOs.** We provide a measurement study on RTX 4090 and M2 Max that compares unconstrained decoding, provider-native structured outputs, grammar-based constraints, and our full spec-driven approach across LM Studio, Ollama, and vLLM. We show how these modes reshape the trade-off frontier between structural correctness, faithfulness, Success@SLO, and tail latency, and we illustrate how the same metrics can be reused as reward components and constraints in single-GPU TRL PPO/GRPO loops [19, 5, 13] (`https://huggingface.co/docs/trl/en/logging`, `https://huggingface.co/docs/trl/en/grpo_trainer`).

The remainder of this paper is organized as follows. Section 2 reviews related work in LLM serving, structured decoding, and agent evaluation. Section 3 formulates contract-first, SLO-aware agent serving as a constrained optimization problem. Section 4 presents the evaluation framework and test families in detail, including our faithfulness scorer and trajectory tests. Section 5 instantiates the framework on single-GPU backends and details our spec compiler, validation-and-retry, budgeted self-consistency, and measurement study of decoding modes under SLOs. Section 5 reports single-GPU experiments and a measurement study of decoding modes and backends. Section 6 discusses limitations and future directions, including safety, multi-turn interactions, and deeper RL integration.

2 Background and Related Work

Our work sits at the intersection of systems for LLM serving, structured decoding and contract-first generation, automated evaluation of faithfulness and reliability, and emerging agent evaluation frameworks used in production environments. This section reviews prior work in each of these areas and highlights the gaps that SpecSLOEval is designed to fill.

2.1 Tail Latency and LLM Serving

Classic systems work emphasizes that users experience the worst-case behavior of a service rather than its average latency. Dean and Barroso's "The Tail at Scale" [12] (`https://cacm.acm.org/`

research/the-tail-at-scale/) shows that even a small fraction of slow requests can dominate perceived quality and limit system scalability. They advocate design principles that explicitly target tail latency—p95, p99—through redundancy, hedging, careful queueing, and resource management.

LLM-serving work inherits these concerns and exacerbates them due to high per-request compute costs, variable-length outputs, and complex batching behavior. vLLM introduces PagedAttention to provide efficient KV-cache management and continuous batching, significantly improving throughput and memory utilization for transformer models [16] (https://arxiv.org/abs/2309.06180). SGLang focuses on efficient execution of structured language model programs, proposing system-level optimizations such as speculative execution of subprograms and optimized scheduling [21] (https://arxiv.org/abs/2312.07104). DeepSpeed-FastGen [20] (https://arxiv.org/abs/2401.08671) and vAttention [18] (https://arxiv.org/abs/2405.04437) introduce kernel and memory-layout optimizations that reduce compute and memory overhead, further helping tail latency. FlashAttention-2 [11] (https://arxiv.org/abs/2307.08691) redesigns the attention kernel for better parallelism and cache usage. Speculative decoding, as described by Leviathan *et al.* [17] (https://arxiv.org/abs/2211.17192), drafts tokens with a smaller model and verifies them with the target model to reduce time-to-first-token.

Sarathi-Serve [3] (https://www.usenix.org/system/files/osdi24-agrawal.pdf) takes a holistic view of throughput-latency trade-offs for LLM inference, examining queueing models and scheduling policies under realistic traffic patterns and mixed workloads. However, these systems papers generally treat the model as a black box that emits unconstrained text and focus on hardware and scheduling. They do not reason about the *structure* or *semantics* of outputs (e.g., JSON validity, schema adherence, faithfulness to context), and they typically assume multi-GPU clusters or cloud-scale deployments rather than single-GPU setups.

By contrast, our focus is on single-GPU serving (RTX 4090 or equivalent, Apple M-series) for agents that must satisfy contracts and SLOs. We build on the scheduling and kernel-level insights from vLLM, Sarathi-Serve, DeepSpeed-FastGen, and related work, but integrate them with contract-first decoding and a multi-metric evaluation suite tailored to structured agent use cases. SpecSLOEval is concerned not only with how *fast* agents produce text, but also with whether that text is structurally valid, faithful, stable, and compatible with downstream systems.

2.2 Structured Outputs, Grammars, and Contract-First Decoding

Many real-world applications want structured outputs—JSON documents, typed objects, or tool-call payloads—not arbitrary free-form text. API providers have responded with structured-output modes. OpenAI introduced function calling and structured outputs (https://openai.com/index/introducing-structured-outputs-in-the-api/), which allow developers to specify JSON Schema-like contracts that the model is expected to satisfy. Azure OpenAI and Snowflake Cortex provide similar capabilities for structured completions and strongly typed outputs (https://docs.snowflake.com/en/user-guide/snowflake-cortex/complete-structured-outputs).

Local runtimes mirror these trends. LM Studio exposes an OpenAI-compatible interface for local models, including tools and structured outputs (https://lmstudio.ai/docs/developer/openai-compat/tools). Ollama provides structured-output capabilities and supporting documentation (https://docs.ollama.com/capabilities/structured-outputs, https://blog.danielclayton.co.uk/posts/ollama-structured-outputs/). Fireworks.ai advocates for structured output modes across all LLMs, emphasizing reductions in integration errors and easier downstream processing (https://fireworks.ai/blog/why-do-all-LLMs-need-structured-output-modes).

Beyond provider-specific APIs, open-source libraries and runtimes support generic contract-first tooling. Outlines [1] (https://dottxt-ai.github.io/outlines/) implements grammar-based decoding for regular languages and JSON, enforcing Pydantic models on top of arbitrary providers. ‘llama.cpp‘ supports GBNF grammars (`https://github.com/ggml-org/llama.cpp/blob/master/grammars/README.md`), enabling constrained decoding even with local C++ inference. Instructor (`https://python.useinstructor.com/integrations/llama-cpp-python/`) layers structured decoding on ‘llama-cpp-python‘ and constructs typed Python objects from LLM outputs.

Recent benchmark efforts explicitly investigate structured generation. JSONSchemaBench [15] (`https://arxiv.org/abs/2501.10868`) proposes standardized JSON Schema tasks to test the ability of LLMs to follow structured contracts. Studies of format constraints and structured RAG [8, 10] (`https://arxiv.org/abs/2408.02442`, `https://arxiv.org/abs/2408.11061`) show that forcing specific formats can improve syntactic validity but may interact with other output properties (e.g., verbosity, creativity, over-constraining content).

These works collectively show that structured decoding is powerful and that contracts can be compiled into provider- and runtime-specific constraints. However, they primarily evaluate contract adherence in isolation: does the model produce syntactically valid JSON given a schema? They do not systematically study how structured decoding interacts with: (i) serving runtimes and tail latency; (ii) multi-dimensional evaluation metrics (faithfulness, stability, tool behavior, SLOs); or (iii) single-GPU local deployment constraints.

SpecSLOEval treats structured decoding as one layer in a larger agent stack. We define contracts once, compile them into backends such as LM Studio, vLLM, and llama.cpp, and connect them to an evaluation framework that measures not only schema adherence but also task accuracy, faithfulness, trajectories, stability, and SLO adherence in realistic agent workflows.

2.3 Automated Evaluation of Faithfulness and Factuality

Evaluating the faithfulness or factuality of LLM outputs is challenging, particularly when ground truth is complex or implicit. RAGAS (Retrieval Augmented Generation Assessment) [14] (`https://arxiv.org/abs/2309.15217`) proposes automatic metrics for retrieval-augmented generation, including *faithfulness* (does the answer contradict the context?) and *answer correctness*. RAGAS typically uses an LLM-as-judge setup: a judge model receives the context and the answer and assigns scores. This design has influenced many RAG evaluation pipelines in industry and open-source tooling.

Other work advocates decomposing responses into “atomic facts” and scoring each fact separately, improving interpretability and reducing the risk that a single aggregate score hides critical errors. Kriman *et al.* [7] (`https://arxiv.org/abs/2408.15171`) propose atomic-fact–based evaluation for summarization, extracting fine-grained claims and checking each against the source. This style of evaluation is particularly relevant for agents that summarize logs, documents, or metrics: a short narrative can be decomposed into claims such as “metric X increased week-over-week” or “customer Y is in segment Z”, each of which can be checked against data.

At the same time, several studies warn that LLM-as-judge setups can be biased and unstable. For example, recent work on LLM judges [6] (`https://arxiv.org/abs/2305.17926`) documents positional bias, sensitivity to phrasing, and preferences for verbose answers, raising concerns about using LLMs as the sole evaluators in high-stakes contexts. This motivates treating judge-based metrics as proxies that must be calibrated and complemented with other signals.

Our framework adopts the insights of RAGAS and atomic-fact evaluation but adapts them to structured agent outputs and SLO-constrained settings. We: (i) restrict attention to contexts where ground truth support is relatively clear (e.g., numeric or tabular data, concrete documents); (ii) use a 0–3 support scale with an explicit contradiction flag, increasing granularity; (iii) calibrate thresholds with small human-labeled subsets; and (iv) treat the judge-based faithfulness metric as one component of a broader metric vector, not a standalone arbiter. SpecSLOEval is designed so that faithfulness signals can be used both for static assessment and as reward components in RL, while acknowledging the limitations of judge models.

2.4 Agent Evaluation Frameworks in Practice

Cloud providers and tool vendors have begun to systematize agent evaluation, especially for tool-using and workflow-oriented agents. Google's Agent Development Kit (ADK) documentation on "Why Evaluate Agents" (`https://google.github.io/adk-docs/evaluate/`) argues that agents should be evaluated both on final response quality and on the quality of trajectories (tool choices, sequencing, error handling). Google Cloud's blog posts on agent testing and evaluation (`https://medium.com/google-cloud/agent-testing-and-evaluation-methods-64d1bd9ac44f`, `https://cloud.google.com/blog/topics/developers-practitioners/a-methodical-approach-to-agent-evaluatio`) recommend golden sets, task-specific rubrics, and coverage-oriented testing. Vertex AI's evaluation overview (`https://cloud.google.com/vertex-ai/generative-ai/docs/models/evaluation-overview`) and the LLM-Evalkit tools (`https://github.com/GoogleCloudPlatform/generative-ai/tree/main/tools/llmevalkit`, `https://cloud.google.com/blog/products/ai-machine-learning/introducing-llm-evalkit`) provide code templates and metrics for evaluating Vertex-hosted models and agents.

These resources establish useful concepts—golden datasets, agent "unit tests," trajectory inspection—but they are tied to specific cloud platforms and assumptions about where the model is hosted. They typically focus on final answer quality, simple success flags, or coarse-grained business metrics, and leave structural, faithfulness, stability, and SLO-specific metrics under-specified. Moreover, they do not directly address agents running on local, single-GPU backends such as LM Studio or Ollama, where privacy and cost constraints differ from cloud settings.

SpecSLOEval generalizes these practices by: (i) defining a backend-agnostic test suite, expressed declaratively in a single `criteria.yaml`, that can be executed against local or cloud-hosted endpoints; (ii) making structure, task accuracy, faithfulness, tools/trajectories, stability, and SLOs first-class metric families; and (iii) integrating evaluation outputs with CI and RL pipelines via W&B logging (`https://docs.wandb.ai/models/quickstart`, `https://docs.wandb.ai/models/artifacts`, `https://docs.wandb.ai/models/evaluate-models`). This adds a concrete, portable realization of ideas that cloud evaluation docs describe at a higher level.

2.5 Single-GPU Local Serving and Open-Weight Models

A key practical difference between many academic systems papers and our target environment is the hardware and deployment model. Instead of assuming multi-GPU clusters or managed APIs, we explicitly target *single-node* deployments: a workstation with an RTX 4090 or a laptop-class device with Apple M-series, running local OpenAI-compatible servers.

LM Studio documentation describes how to run local endpoints and load open-weight models via an OpenAI-compatible interface (`https://lmstudio.ai/docs/developer/openai-compat/tools`, `https://lmstudio.ai/models`). Qwen's documentation explains how to host Qwen models locally and via OpenAI-like APIs (`https://qwen.readthedocs.io/en/latest/getting_started/quickstart.html`). Ollama offers a simple CLI and HTTP API for running models locally with optional structured outputs (`https://docs.ollama.com/capabilities/structured-outputs`). vLLM is frequently used as a high-performance backend for open-weight models, with recipes for deployment on services like Modal [16] (`https://modal.com/docs/examples/vllm_inference`).

llama.cpp provides highly optimized CPU/GPU inference for quantized models.

These tools make it possible for small teams to run reasonably capable LLM agents entirely on their own hardware, but they do not dictate how agents should be evaluated, how contracts should be enforced, or how SLOs should be defined. SpecSLOEval is designed to sit above these runtimes: it assumes only an OpenAI-compatible API and sufficient observability (latency, token counts, prompts, responses) to compute metrics and log them to W&B. By focusing on single-GPU local serving for structured agents, we bridge a gap between large-scale, research-oriented serving work and the needs of teams that want to deploy agents securely and predictably on their own infrastructure.

This background motivates our problem formulation. In the next section, we formalize contract-first, SLO-aware agent serving as a constrained optimization problem and introduce the metric families and configuration surface that underpin SpecSLOEval.

3 Problem Formulation

We target single-GPU, OpenAI-compatible serving stacks (e.g., LM Studio, Ollama, vLLM) where agents emit structured JSON to downstream systems. A generic commercial agent sits between upstream requesters (users, batch jobs, or other services) and downstream consumers (dashboards, workflows, humans). For each request, the agent receives unstructured and structured context (documents, logs, records, prior turns), may call tools (APIs, databases, services), and is expected to emit *structured outputs* such as JSON reports, action recommendations, or tool-call payloads. These outputs do not only surface in a chat window; they are consumed by campaign engines, scoring pipelines, ETL jobs, monitoring dashboards, and human decision-makers. Small model errors can therefore propagate and compound: a malformed JSON object may crash a downstream job; a mis-typed field can poison an aggregate; a subtly incorrect statement about a key metric may trigger the wrong business action. At the same time, users and systems experience the agent through latency: if responses frequently violate p95/p99 latency budgets, trust erodes even when mean latency and accuracy look acceptable.

We want a formulation that is: (i) amenable to systems-level analysis (latency distributions, SLO constraints, single-GPU capacity); (ii) fine-grained enough to capture semantics (structure, task accuracy, faithfulness, tool behavior, stability); and (iii) compatible with both static evaluation and reinforcement-learning-based improvement. We therefore define: an abstract *agent and contract model* (Section 3.1), a *serving environment* with explicit latency-based SLOs (Section 3.2), and a vector of *evaluation metrics* configured via a single criteria file (Section 3.3). Figure 1 summarizes the metric families; Figure 2 gives an optimization view that supports both static configuration search and constrained RL.

Throughout this section we use the term *configuration* for a concrete choice of model, runtime, decoding settings, and contract. The rest of the paper instantiates this formulation for single-GPU deployments (RTX 4090 or Apple M-series) and a standard set of evaluation tasks.

3.1 Agent, Contract, and Configuration

We formalize an *agent invocation* as a single request–response episode. Let $\mathcal{X}$ denote the space of requests. Each request $x \in \mathcal{X}$ may consist of:

- a user instruction or task description (free-form text or a structured spec),
- a multiset of context items $C = \{c_1, \ldots, c_m\}$ such as retrieved documents, KB entries, prior turns, log excerpts, or database records,

- optional metadata such as user profile, locale, device information, or authorization context.

We deliberately keep $\mathcal{X}$ domain-agnostic: the same formalism covers analytics summarization, support automation, code editing, incident response, and other agent tasks.

For each request x, the agent must produce one or more outputs y in an output space $\mathcal{Y}$. The structure of $\mathcal{Y}$ is governed by a *contract* $\mathcal{C}$. In practical systems, such contracts are written as JSON Schemas, Pydantic models, protocol buffer definitions, or grammars. We abstract this as a predicate

$$\mathcal{C} : \mathcal{Y} \to \{0, 1\},$$

which returns 1 if and only if y is syntactically well formed (e.g., valid JSON) and satisfies all structural constraints: field presence, types, enum values, bounds, nesting, and any additional invariants encoded in the contract (e.g., fields that must sum to one, monotonicity constraints). By *contract-first* we mean that $\mathcal{C}$ is specified before the agent is invoked and that decoding is explicitly designed to respect $\mathcal{C}$ rather than being retrofitted after the fact.

We assume a parametric policy π_θ, typically an LLM with prompts and optionally fine-tuned weights or adapters. The policy is deployed via a serving runtime R (e.g., LM Studio, Ollama, vLLM, or `llama.cpp`) that exposes an OpenAI-compatible API. Given a request x, contract $\mathcal{C}$, and decoding configuration δ (temperature, top-p, maximum tokens, number of self-consistency samples k, batch size, speculative-decoding flags, etc.), the agent induces a conditional distribution over outputs

$$y \sim \pi_\theta(\cdot \mid x, \mathcal{C}, \delta, R).$$

Modern runtimes support different mechanisms to enforce or approximate the contract $\mathcal{C}$:

- **Provider-native constraints.** Here $\mathcal{C}$ is translated into a schema or tool spec understood directly by the serving runtime. Examples include OpenAI-style `response_format` and function calling (`https://openai.com/index/introducing-structured-outputs-in-the-api/`), LM Studio's OpenAI-compatible tools (`https://lmstudio.ai/docs/developer/openai-compat/tools`), and Ollama's structured outputs (`https://docs.ollama.com/capabilities/structured-outputs`).

- **Library-based or post-hoc constraints.** Here $\mathcal{C}$ is enforced via third-party libraries and explicit validation-and-retry logic. Examples include Outlines (`https://dottxt-ai.github.io/outlines/`) and GBNF grammars in `llama.cpp` (`https://github.com/ggml-org/llama.cpp/blob/master/grammars/README.md`), combined with JSON schema validation libraries.

From the agent's perspective, the contract $\mathcal{C}$ and the configuration δ are part of its *effective policy*: changing either alters the mapping $x \mapsto y$ even if the underlying model weights remain fixed.

For compactness, we write a *configuration* as

$$\kappa = (R, \pi_\theta, \delta, \mathcal{C}).$$

Given a workload (distribution over requests) W and environment (tools, data sources), a configuration κ induces a distribution over episodes and outputs. The goal of our framework is to evaluate and ultimately optimize over such configurations under realistic workload and SLO constraints.

3.2 Serving Environment and Latency SLOs

We assume the agent stack runs on a *single, finite-capacity node* equipped with a GPU (e.g., NVIDIA RTX 4090) or a laptop-class accelerator (e.g., Apple M2 Max). The serving runtime R

performs prefill and decoding, possibly batching multiple requests, managing KV caches, and using techniques such as PagedAttention as in vLLM [16] (`https://arxiv.org/abs/2309.06180`).

For each request x served under configuration κ, we define the end-to-end latency

$$L_\kappa(x) = t_{\text{finish}}(x) - t_{\text{start}}(x),$$

where $t_{\text{start}}(x)$ is the time when x is submitted to the serving system and $t_{\text{finish}}(x)$ is the time when a fully validated structured output y (or an explicit error) is available to downstream consumers. By "fully validated" we mean that validation-and-retry logic has run, contracts have been checked, and any fallback behavior has completed.

Requests arrive according to a stochastic workload model that may include bursts, diurnal patterns, and varying context sizes. We denote this workload model by W, which induces both a sequence of arrival times and a distribution over request payloads $x \in \mathcal{X}$. Given κ and W, the induced latency distribution is

$$Q_\kappa^W = \mathcal{L}\big(L_\kappa(x) \mid x \sim W\big),$$

and we summarize it via quantiles

$$\text{p50}(Q_\kappa^W), \quad \text{p95}(Q_\kappa^W), \quad \text{p99}(Q_\kappa^W),$$

and, when relevant, time-to-first-token (TTFT) and throughput (queries per second, QPS). Following Dean and Barroso [12], p95and p99are treated as first-class metrics: they control perceived responsiveness and throughput in multi-step, fan-out-heavy systems.

Service-level objectives (SLOs) are typically expressed as bounds on these quantiles. A common pattern is:

$$\text{p95}(Q_\kappa^W) \le B, \qquad \text{p99}(Q_\kappa^W) \le B',$$

for budgets B and B' in milliseconds (e.g., 800 ms and 2000 ms). Some organizations also specify hard timeouts (e.g., "no request may exceed 5 seconds") or constraints on TTFT (e.g., "TTFT must be below 300 ms for chat experiences").

Critically, in many enterprise settings latency is not the *primary* objective. Stakeholders often adopt a *lexicographic* preference ordering:

- **First, correctness and structure.** Outputs must be structurally valid (JSON parses and satisfies $\mathcal{C}$) and semantically correct (task accuracy and faithfulness above thresholds). A fast but hallucinated or malformed response is worse than a slower but correct one.

- **Second, stability.** Given correctness, outputs should be stable across seeds, versions, and days. Large unexplained fluctuations erode user trust and complicate monitoring.

- **Third, latency.** Given correctness and stability, latency should satisfy agreed SLOs. Only in explicitly latency-dominant applications (e.g., real-time trading, user-typing completions) does speed dominate correctness.

We encode this preference in our evaluation criteria and reward design: latency appears primarily as a *constraint* and only secondarily as a negatively weighted objective, except for configurations explicitly marked as speed-critical.

To capture the combined notion of "correct and fast enough," we define an episode-level indicator

$$\text{Success@SLO}_\kappa(x) = \begin{cases} 1 & \text{if the episode satisfies all selected quality criteria and latency constraints,} \\ 0 & \text{otherwise.} \end{cases}$$

Quality criteria are derived from the metric vector $m_\kappa(x, y)$ (Section 3.3); latency constraints are derived from Q^W_κ and budgets (B, B'). Aggregating Success@SLOover an evaluation set yields a concise, business-aligned metric: the fraction of requests that are both *good* and *on time*.

3.3 Metric Families and Criteria Configuration

To reason about agents in a way that is auditable, tunable, and reusable across backends, we define for each episode a metric vector

$$m_\kappa(x, y) \in \mathbb{R}^K,$$

whose components are grouped into families as illustrated in Figure 1: *structure*, *task accuracy*, *faithfulness*, *tools and trajectory*, *stability*, and *latency/SLOs*. We briefly describe each here; Section 4 gives concrete definitions and example tests.

Structure. Structural metrics capture whether the output is syntactically and contractually well formed:

- JSON parse success (binary: parses or not),
- schema validity under $\mathcal{C}$ (binary),
- detailed error codes (missing fields, extra fields, type mismatches, constraint violations).

These are necessary conditions for safe downstream consumption. We log fine-grained error counts (e.g., per-field violation rates) for debugging and potential RL shaping, even if only coarse indicators feed primary scores.

Task accuracy. When ground-truth labels or references exist, we compute:

- macro- or micro-F1 for extraction and classification tasks,
- exact match (EM) or accuracy for QA-style tasks,
- pass@k for code or function synthesis tasks.

These metrics answer the question "does the agent actually solve the task it was asked to solve?", independently of style or latency.

Faithfulness. Faithfulness metrics quantify whether y is supported by the context C. Following RAGAS [14] (`https://arxiv.org/abs/2309.15217`) and atomic-fact evaluation [7] (`https://arxiv.org/abs/2408.15171`), we:

1. use a judge model to decompose y into atomic statements;
2. assign each statement a support score $s \in \{0, 1, 2, 3\}$ ("not supported" to "strongly supported") and a contradiction flag;
3. aggregate these into a scalar $m_{\text{faith}}(x, y) \in [0, 1]$ that reflects the fraction of strongly supported statements, the average support, and a penalty for contradictions.

We treat faithfulness as distinct from task accuracy: an answer can match a label but still misrepresent or over-claim relative to the underlying evidence.

Tools and trajectories. For tool-using agents we log, per episode:

- tool-call success rates (valid arguments, no schema violations, non-error responses),
- trajectory statistics: number of tool calls, invalid-argument retries, redundant calls,
- deviation from expected plans when such plans exist (e.g., edit distance between expected and realized tool sequences), in the spirit of Google's ADK (`https://google.github.io/adk-docs/evaluate/`) and LLM-Evalkit (`https://github.com/GoogleCloudPlatform/generative-ai/tree/main/tools/llmevalkit`).

These metrics expose how the agent behaves in the environment, not just what final JSON it produces.

Stability. Stability metrics quantify variability across runs:

- Disagreement@k, defined as one minus the fraction of "equivalent" outputs across k repeated runs (where equivalence can be defined via schema-respecting canonicalization or atomic-fact overlap),
- variance of key metrics (e.g., accuracy, faithfulness) across random seeds, in line with recent work on LLM stability [9] (`https://arxiv.org/abs/2408.04667`).

High instability—even at similar averages—can be operationally unacceptable.

Latency and SLOs. For each episode we record $L_\kappa(x)$ and, when relevant, TTFT. Over an evaluation set $\mathcal{D}_{\text{eval}}$, we compute:

- p50, p95, p99;
- Success@SLOrates under various budgets;
- tail-violation rates (fraction of requests exceeding hard cutoffs).

These metrics capture responsiveness and capacity under the chosen configuration.

All of these metrics, along with their thresholds and aggregation weights, are configured via a single file, `agent_eval/criteria.yaml`. Conceptually, this file specifies:

1. **Metric definitions:** which metrics are active for a given evaluation and how they are computed;
2. **Thresholds and gates:** per-metric pass/fail thresholds (e.g., JSON validity ≥ 0.99, faithfulness ≥ 0.75, Disagreement@$k \leq 0.2$) and SLO bounds for latency;
3. **Aggregation weights:** weights for combining metrics into scalar scores for dashboards or RL rewards.

A typical composite score might take the form

$$\text{Score}_\kappa(x,y) = w_{\text{task}} m_{\text{task}} + w_{\text{faith}} m_{\text{faith}} + w_{\text{json}} m_{\text{json}} + w_{\text{tool}} m_{\text{tool}} + w_{\text{slo}} m_{\text{slo}} - \lambda_{\text{lat}} L_\kappa(x) - \lambda_{\text{disc}} m_{\text{disc}},$$

where m_{task} is a task-accuracy metric (e.g., F1), m_{faith} a faithfulness score, m_{json} a structural validity indicator, m_{tool} a tool-success metric, m_{slo} encodes SLO-related success, and m_{disc} captures disagreement/instability. Weights are chosen such that $w_{\text{task}}, w_{\text{faith}}, w_{\text{json}}$ dominate λ_{lat} in non–latency-critical regimes, encoding the "correctness and truthfulness before speed" philosophy.

Given a workload W, the expected performance of a configuration κ under these criteria is

$$J(\kappa; W) = \mathbb{E}_{x\sim W,\ y\sim\pi_\theta(\cdot|x,\mathcal{C},\delta,R)}\big[\mathrm{Score}_\kappa(x,y)\big],$$

subject to SLO and quality constraints on the induced latency distribution Q_κ^W and key aggregate metrics (e.g., minimum JSON validity and faithfulness).

3.4 Offline vs Online Evaluation

Although W denotes the production workload conceptually, in practice we often evaluate on a held-out distribution $\widehat{W}$ built from: (i) logged production requests with labels or references, (ii) synthetic or adversarially constructed cases, and (iii) standardized benchmarks. Our harness is agnostic to whether requests come from W or $\widehat{W}$; the same metric definitions and criteria apply. This separation is important: offline evaluation on $\widehat{W}$ must approximate W closely enough that improvements in $J(\kappa; \widehat{W})$ translate into improvements in $J(\kappa; W)$, but the criteria file and metric families remain unchanged.

3.5 Static Configuration Selection and Policy Improvement

Figure 2 presents the optimization view underlying our framework. A configuration $\kappa = (R, \pi_\theta, \delta, \mathcal{C})$, together with a workload W and environment (tools, data sources), induces a distribution over episodes and metric vectors $m_\kappa(x, y)$. Aggregating metrics over an evaluation set yields performance summaries $J(\kappa; W)$ and diagnostics (constraint violations, SLO breaches, error distributions).

We consider two complementary uses of this formulation:

Static configuration selection. In static selection, we treat θ as fixed (prompt-only or frozen model weights) and regard R, δ, and possibly $\mathcal{C}$ as tunable knobs. The design problem is

$$\max_{\kappa\in\mathcal{K}} J(\kappa; W) \quad \text{s.t.} \quad \mathrm{p95}(Q_\kappa^W) \le B, \quad \mathrm{p99}(Q_\kappa^W) \le B', \quad \bar{m}_{\mathrm{json}}(\kappa) \ge \tau_{\mathrm{json}}, \quad \bar{m}_{\mathrm{faith}}(\kappa) \ge \tau_{\mathrm{faith}}, \ldots$$

where $\mathcal{K}$ is a search space over runtimes, decoding parameters, and possibly contract variants, and $\bar{m}_{\mathrm{json}}(\kappa)$, $\bar{m}_{\mathrm{faith}}(\kappa)$ denote aggregate structural and faithfulness metrics. In practice, one can explore $\mathcal{K}$ via grid search, Bayesian optimization, or bandit-style exploration, with our metric suite providing the objective and SLO constraints. This use case is analogous to how tools such as LLM-Evalkit are used for cloud-hosted agents, but here our metrics explicitly support local single-GPU backends and contract-first agents.

Policy improvement under constraints. In policy improvement, we treat θ as trainable (e.g., via RL or supervised fine-tuning) and interpret part of $m_\kappa(x, y)$ as a reward vector, with additional components acting as costs. For example, we might define per-episode reward

$$r_\kappa(x,y) = \alpha_{\mathrm{task}} m_{\mathrm{task}} + \alpha_{\mathrm{faith}} m_{\mathrm{faith}} + \alpha_{\mathrm{json}} m_{\mathrm{json}} - \beta_{\mathrm{lat}} \max\big(0, L_\kappa(x) - B\big),$$

and treat SLO violations, multiple retries, or instability as separate costs $c_j(x, y)$ in a constrained MDP [4, 2]. Using TRL's PPO or GRPO trainers with LoRA-based adapters [19, 5, 13] (`https://huggingface.co/docs/trl/en/logging`, `https://huggingface.co/docs/trl/en/grpo_trainer`), we can update θ so that the agent improves on quality and faithfulness metrics while maintaining SLO compliance and controlling instability. In this view, the evaluation framework (metrics + `criteria.yaml`) defines the reward and cost surfaces on which RL operates.

Visualization of the metric families (structure, task accuracy, faithfulness, tools/trajectory, stability, latency/SLOs, cost/safety).

Figure 1: Metric families in our evaluation framework. We track structure (syntactic and schema validity), task accuracy, faithfulness to context, tool and trajectory behavior, stability across runs, latency/SLO metrics, and cost/safety. The concrete metrics, thresholds, and weights are configured centrally via `agent_eval/criteria.yaml`.

Visualization of the optimization loop: config κ drives an environment/workload and evaluation harness that aggregate metrics $J(\kappa; W)$ for static selection and RL training.

Figure 2: Optimization view. Given a configuration $\kappa = (R, \pi_\theta, \delta, \mathcal{C})$ and a workload W, the environment and evaluation harness produce metrics $m_\kappa(x, y)$ that are aggregated into performance summaries $J(\kappa; W)$ and constraint diagnostics. These summaries support both static configuration selection and SLO-aware policy improvement using PPO/GRPO-based RL in TRL.

In both regimes, the formulation in this section provides the "contract-first, SLO-aware" backbone for the rest of the paper. The next section describes how these abstract objects—contracts, configurations, and metrics—are realized concretely via spec-driven decoding and single-GPU serving on LM Studio, Ollama, vLLM, and `llama.cpp`.

4 Evaluation Framework and Test Families

Figure 1 summarized the metric families we track. We now describe the evaluation framework that turns these abstract metric families into concrete, repeatable tests. The goal of this framework—which we will refer to as *SpecSLOEval*—is to provide an extensible, auditable testbed that captures the behaviors practitioners actually care about in real deployments: structural correctness, task accuracy, faithfulness to context, tool and trajectory behavior, stability, and latency/SLO adherence on single-GPU hardware.

SpecSLOEval is organized around three ideas:

1. A *single source of truth*, `agent_eval/criteria.yaml`, which declares metrics, thresholds, weights, and test families.
2. A set of *test families* aligned with the metric families in Figure 1, each with reusable parameters (schemas, datasets, judge models, workload models).
3. A *logging and analysis layer* that records per-episode metrics, configuration metadata, and aggregated statistics to W&B and Weave for offline analysis and, in P2, RL training.

We deliberately separate the abstract definition of metrics from specific domains. The same framework can be applied to analytics agents, customer-support agents, RAG systems, code assistants, and other tools, simply by swapping contracts, datasets, and thresholds.

4.1 Evaluation Configuration via `criteria.yaml`

The entire evaluation framework is driven by a single configuration file, `agent_eval/criteria.yaml`. This file acts as an explicit contract between the agent and the evaluation harness: it describes what we measure, how we decide pass/fail, and how we aggregate scores.

At a high level, `criteria.yaml` specifies:

1. **Metric definitions:** the metrics to compute for each episode (e.g., JSON validity, macro-F1, faithfulness, Disagreement@k, Success@SLO) and, where relevant, their computation parameters (e.g., judge model, number of repeated runs).
2. **Thresholds and gates:** thresholds or ranges that define acceptable behavior (e.g., minimum JSON validity rate, minimum faithfulness score, maximum disagreement).
3. **Weights:** a vector of weights that combine metrics into aggregate scores for dashboards and RL reward functions.
4. **Test families and families-to-metrics mapping:** which metrics are associated with which test families (structure, accuracy, faithfulness, tools/trajectory, stability, latency/SLO), and which datasets/schemas each family uses.

A simplified example is shown below:

```
schema_version: "1.0"

slo:
  p95_ms:          800     # target p95 latency
  p99_ms:          1500    # target p99 latency
  max_tokens_out:  1024
  # Optional: explicit hard timeout in ms
  hard_timeout_ms: 5000

metrics:
  json_validity:
    family: "structure"
    type: boolean_rate
    pass_threshold: 0.99

  field_f1:
    family: "accuracy"
    type: macro_f1
    pass_threshold: 0.85
    fields: ["category", "severity", "time_window"]

  faithfulness:
    family: "faithfulness"
    type: judge_support_0_to_3
    pass_threshold: 0.75
    judge_model: "qwen2.5-7b-instruct"
    judge_endpoint: "http://10.0.0.63:1234/v1"
    max_contradiction_rate: 0.05

  tool_success:
    family: "tools"
    type: boolean_rate
    pass_threshold: 0.90

  disagreement_k:
    family: "stability"
    type: disagreement
    max_threshold: 0.20
    runs: 5
    equivalence: "canonical_json"

  success_at_slo:
    family: "slo"
    type: joint_success
    pass_threshold: 0.80

weights:
  json_validity:   0.15
  field_f1:        0.25
  faithfulness:    0.25
  tool_success:    0.15
  success_at_slo:  0.15
  disagreement_k:  0.05
```

This configuration encodes our design choices explicitly:

- Structural correctness (`json_validity`), task accuracy (`field_f1`), and faithfulness (`faithfulness`) carry more weight than latency, except where a use case is explicitly marked latency-dominant.
- Success@SLOis treated as a composite “good and on-time” indicator, not a simple latency-only metric.
- Stability (`disagreement_k`) is rewarded, but not at the expense of correctness and faithfulness.

Because `criteria.yaml` is versioned alongside the code, changes to evaluation policy (e.g., stricter thresholds or additional metrics such as safety) are visible and auditable. Experiments are tagged with the schema version so that results collected under different criteria can be compared correctly.

4.2 Test Families and Parameterization

We group tests into families that mirror the metric families in Figure 1: *Structure*, *Task Accuracy*, *Faithfulness*, *Tools and Trajectories*, *Stability*, and *Latency/SLO*. Each family is parameterized by contracts $\mathcal{C}$, datasets, judge models, and workload parameters, all of which are declared in configuration files (YAML/JSON) and logged to W&B.

Below we outline each family and provide concrete examples. In Section 5 we instantiate a subset of these tests for our single-GPU experiments.

4.2.1 Structure: JSON and Schema Correctness

The structure family covers syntactic and structural adherence to contracts. Tests in this family use contracts $\mathcal{C}$, JSON Schema validators (e.g., Python `jsonschema` with 2020-12 support; see `https://json-schema.org/` and `https://www.learnjsonschema.com/2020-12/`), and optionally grammar-based decoders (e.g., GBNF grammars for `llama.cpp`).

Parameters. Each structural test is defined by:

- a schema or contract $\mathcal{C}$ (referenced by name and version),
- a set of prompts or synthetic tasks that exercise that schema,
- a pass threshold on the fraction of outputs that parse and validate.

Example S1: Basic JSON/Schema Validity. We fix a contract $\mathcal{C}_{\text{basic}}$ describing a small but nontrivial response object (e.g., `short_description`, `details.main_points`, `details.risk_score`). We run the agent on N prompts (e.g., paraphrased instructions, varying context sizes) and compute two per-episode indicators: (i) JSON parse success, and (ii) schema validity. We aggregate these into $m_{\text{json}} = \frac{\#\text{valid episodes}}{N}$ and require $m_{\text{json}} \geq 0.99$.

Example S2: Schema Evolution Robustness. Real systems evolve schemas over time. We define a pair of schemas $(\mathcal{C}_1, \mathcal{C}_2)$ where $\mathcal{C}_2$ adds optional fields or relaxes constraints (e.g., new `explanations` field). We then test whether: (i) the same configuration can be switched from $\mathcal{C}_1$ to $\mathcal{C}_2$ without catastrophic regression in validity; and (ii) a "dual-mode" config can emit outputs that are valid under both schemas. This test stresses the spec compiler and decoder, and reveals how brittle the agent is to schema changes.

4.2.2 Task Accuracy: F1, EM, and pass@k

The task-accuracy family captures correctness with respect to known references or labels. For extraction tasks we use field-level metrics (e.g., macro-F1). For classification and QA tasks we use accuracy or exact match. For code or function-generation tasks we may use pass@ k metrics.

Parameters. Accuracy tests specify:

- a dataset $\mathcal{D}_{\text{acc}} = \{(x_i, y_i^{\text{ref}})\}$ with reference outputs,
- a schema $\mathcal{C}$ that defines the shape of predicted outputs,
- a field subset or aggregation rule for metrics (e.g., which JSON fields to score),
- pass thresholds for each metric (e.g., macro-F1 ≥ 0.85).

Example A1: Field-Level Extraction F1. Each x_i contains a short context (e.g., an incident report, log snippet, or internal ticket) and an instruction. The reference y_i^{ref} is a JSON object with fields such as `"category"`, `"severity"`, and `"time_window"`. The agent is prompted to emit a JSON object under $\mathcal{C}_{\text{incident}}$. We compute macro-F1 across fields, treating each field as a label prediction problem. This test directly measures whether the agent can reliably populate structured fields from text.

Example A2: Programmatic QA Exact Match and pass@ k. For tasks analogous to GSM8K (`https://arxiv.org/abs/2110.14168`), MBPP (`https://arxiv.org/abs/2108.07732`), HumanEval (`https://arxiv.org/abs/2107.03374`), or MATH (`https://arxiv.org/abs/2103.03874`), we embed answers in a simple schema with a `"final_answer"` field. We compute EM and, optionally, pass@ k by sampling multiple answers per question. These tests show whether structured decoding and SLO-aware serving degrade or preserve task performance relative to unconstrained baselines.

4.2.3 Faithfulness: LLM-as-Judge with Atomic Statements

The faithfulness family evaluates whether outputs are supported by their context C. We use judge models to break outputs into atomic statements and assign each statement a support score and contradiction flag, inspired by RAGAS [14] (`https://arxiv.org/abs/2309.15217`) and atomic-fact evaluation [7] (`https://arxiv.org/abs/2408.15171`).

Parameters. Faithfulness tests specify:

- a dataset $\mathcal{D}_{\text{faith}} = \{(x_i, C_i)\}$ where C_i is the context (documents, records, logs, retrieved passages),
- a judge model and endpoint (often a local open-weight model via LM Studio),
- a scoring procedure (e.g., 0–3 support scale, contradiction penalties),
- thresholds on average support and maximum contradiction rate.

Example F1: Grounded Narrative Summaries. For each x_i, the agent receives a context C_i (e.g., a set of system logs or analytics records) and is asked to produce a short narrative summary structured as JSON. The judge model is prompted to: (i) decompose the narrative into atomic statements; (ii) assign each statement a support score $s \in \{0, 1, 2, 3\}$; and (iii) flag contradictions. We define

$$m_{\text{faith}}(x_i, y_i) = \max\left(0,\ \frac{\#\{s = 3\}}{\#\text{statements}} - \text{contradiction\_rate}(x_i, y_i)\right),$$

and aggregate across the dataset. We require $m_{\text{faith}} \geq 0.75$ and contradiction rates below 5%. All judge inputs/outputs are logged to W&B for auditing.

Example F2: Numeric and Trend Consistency. For contexts with numeric metrics (e.g., time series of KPIs), we construct tests where the agent must describe trends and relative magnitudes ("metric A increased more than metric B", "current value is above the 90th percentile"). The judge is instructed to focus on numeric consistency and direction-of-change. Statements that invert trends or exaggerate differences are scored as unsupported. This variant targets subtle numerical hallucinations, which are particularly dangerous for analytics and monitoring agents.

4.2.4 Tools and Trajectories

Agents that call tools expose a rich space of behaviors beyond final responses. The tools and trajectories family measures how well the agent uses tools: whether arguments are valid, whether calls succeed, and whether trajectories resemble efficient, interpretable plans.

Parameters. Tool tests specify:

- a catalog of tools (names, JSON argument schemas, expected responses),
- a dataset of tasks requiring tools (e.g., "look up this ID and compute a derived metric"),
- optional canonical or minimal plans (e.g., sequences of tools that are known to be sufficient),
- thresholds on tool-call success and trajectory quality.

Example T1: Tool-Argument Validity and Success Rate. We define one or more tools with JSON argument schemas (e.g., `"user_id"` as a string with a regex constraint, `"start_date"` and `"end_date"` with date formats). A mock API validates arguments against these schemas and returns a success flag. For each episode we record whether each tool call satisfies its schema and whether the mock API returns success. We compute

$$m_{\text{tool}} = \frac{\#\text{successful tool calls}}{\#\text{tool calls}},$$

and require $m_{\text{tool}} \geq 0.9$. This test surfaces mis-specified arguments and integration problems early.

Example T2: Trajectory Cost, Redundancy, and Plan Deviation. For more complex tasks we define expected minimal plans (e.g., "call `get_user`, then `get_orders`, then `compute_summary`"). We measure: (i) the number of extra tool calls beyond the minimal plan; (ii) the number of invalid-argument retries; and (iii) the edit distance between the observed and expected tool sequences. We set thresholds on each quantity (e.g., average extra tools ≤ 1, invalid retries ≤ 0.1 per episode). Agents that over-use tools or thrash on argument errors are penalized, since these behaviors inflate latency and cost.

4.2.5 Stability Across Runs

Stability is often under-emphasized but critical in production. If two identical runs of the same agent on the same input produce materially different structured outputs, it becomes difficult to debug issues, compare versions, or explain outcomes to stakeholders.

Parameters. Stability tests specify:

- a set of prompts $\mathcal{D}_{\text{stab}}$ representative of the target workload,
- a number of repeated runs k per prompt under fixed configuration,
- an equivalence criterion (exact JSON equality, canonical JSON equality, or atomic-fact equivalence),
- maximum allowable disagreement and variance thresholds.

Example ST1: Disagreement@ k for Structured Outputs. We run the agent $k = 5$ times per input and compute the fraction of runs that produce equivalent (canonicalized) JSON outputs. Disagreement@ k is defined as 1 minus this fraction. We treat Disagreement@ $k \leq 0.2$ as acceptable and log both the aggregate value and per-prompt breakdown. Instability often reveals underlying issues in prompts, decoding parameters, or self-consistency budgets.

Example ST2: Seed Variance on Accuracy and Faithfulness. We evaluate the agent multiple times with different random seeds and compute variance in key aggregate metrics (macro-F1, faithfulness). High variance suggests that performance is sensitive to randomness and that offline evaluation may be misleading. We include this variance in the stability family and can penalize it in RL reward shaping.

4.2.6 Latency and SLO Tests

Latency and SLO tests approximate the setting of *The Tail at Scale* [12] and recent LLM-serving work [16, 3] for single-GPU deployments. They evaluate whether configurations meet latency targets *while still satisfying correctness and faithfulness gates.*

Parameters. Latency/SLO tests specify:

- a workload model (arrival process, context length distribution, output-length distribution),
- SLO budgets B and B' for p95 and p99 latency, plus any TTFT or hard timeout constraints,
- structural and quality gates (e.g., JSON validity and faithfulness thresholds) that must hold before a request can count as a success.

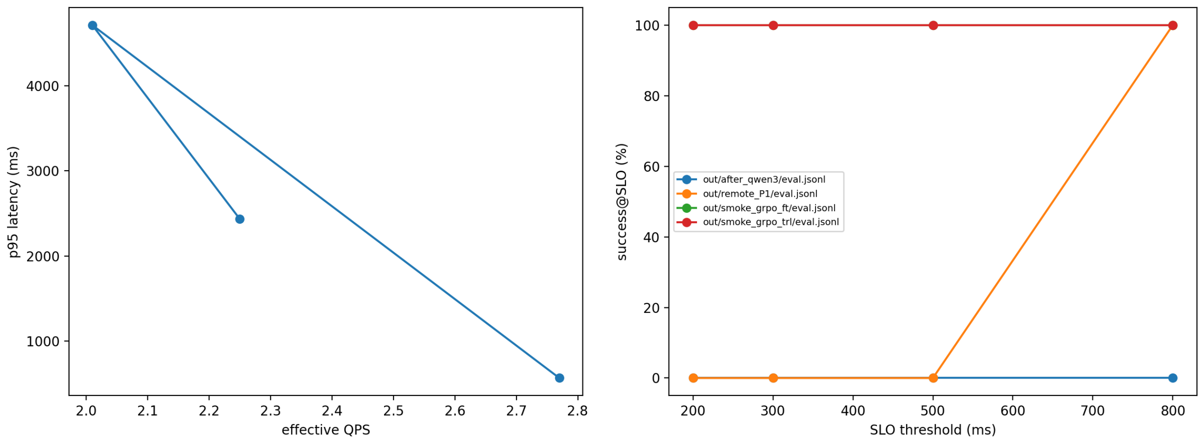

Figure 3: Single-GPU LM Studio baseline: QPS versus p95 latency (left) and Success@SLO(right). Lower token budgets extend the operating frontier before tail-latency collapse.

Example L1: Success@SLOunder Mixed Load. We define a workload with realistic mix of short and long contexts and implement it via a load generator against the single-GPU runtime. For each episode we compute: (i) structural and task/faithfulness metrics; and (ii) latency $L_\kappa(x)$. We then define Success@SLOas an episode that satisfies all quality gates *and* has latency below B. We require aggregate Success@SLO$\geq$ 0.8 and log p50/p95/p99 and quality metrics as a function of load.

Example L2: QPS vs p95 Curves and Configuration Frontier. We perform a sweep over target QPS levels (e.g., 1, 2, 4, 8, 16 QPS) and, for each level, estimate p95/p99 latency and Success@SLO. Plotting QPS versus p95 and QPS versus Success@SLOreveals the "operating frontier" for each configuration κ. These curves, visualized in W&B, guide static selection: configurations that remain in-spec for both quality and latency at higher QPS are preferred.

4.3 Extensibility, Logging, and Use in RL

Although we focus on six core families, SpecSLOEval is designed to be extensible. Adding a new metric (e.g., a safety score, a toxicity classifier output, a business-specific KPI) involves adding an entry to `criteria.yaml` and providing a computation function in the evaluation harness. Because all episodes are logged with a rich set of metadata—configuration hash, commit hash, schema version, dataset version, and per-episode metrics—new analyses can be performed retroactively on past runs.

All experiments are logged online to W&B, with Weave capturing richer computation graphs where appropriate (e.g., judge-model calls, atomic-statement parsing). This makes runs reproducible and auditable: any reported number in a table can be traced back to individual episodes, prompts, and outputs.

Finally, the same metrics and tests that gate configurations in static evaluation feed directly into the RL setup of P2. Reward functions are constructed by combining entries from `weights` with metric values; constraints (e.g., maximum SLO violation rate or disagreement threshold) are implemented as costs in constrained RL algorithms. Thus, the evaluation framework does not merely score agents after the fact; it defines the measurement space in which agents are optimized.

In the next section we describe how contracts and decoding configurations are implemented concretely on single-GPU runtimes (LM Studio, Ollama, vLLM, and `llama.cpp`), and how the evaluation framework is integrated into the training and experimentation loop.

5 Experiments

We now instantiate our framework in a concrete experimental setting on single-GPU hardware. The goal of this section is not merely to present headline numbers, but to show that: (i) spec-driven decoding with validation-and-retry and budgeted self-consistency can be implemented in a realistic, commodity environment; (ii) the resulting behavior can be measured in a principled way across the full metric family of Figure 1; and (iii) the same evaluation stack can support both static configuration search (P1) and, in P2, reinforcement-learning-based policy improvement.

We structure this section as follows. Section 5.1 describes hardware, runtimes, models, and logging. Section 5.3 introduces three task families (structured extraction, grounded summaries, and tool-using episodes) that approximate commercial use cases without binding us to a single vertical. Section 5.4 defines the decoding and serving configurations we compare. Section 5.5 explains how metrics are computed and reported. Sections 5.6–5.8 report representative baseline measurements and illustrate the analyses our framework enables; a broader grid over models, backends, and workloads is straightforward to extend and left for future work.

5.1 Experimental Setup

Hardware and runtimes. Unless otherwise noted, we run experiments on two representative single-node setups:

- **4090 workstation:** NVIDIA RTX 4090 GPU (24 GB VRAM), 64 GB system RAM, Ubuntu 22.04, CUDA 12.x and recent NVIDIA drivers.
- **Apple Silicon laptop:** MacBook Pro with Apple M2 Max and 64 GB unified memory, macOS 15+.

On both machines, we use LM Studio's OpenAI-compatible server as the primary runtime, configured to listen on:

- `http://10.0.0.63:1234/v1` or `http://10.0.0.72:1234/v1` on the workstation;
- `http://localhost:1234/v1` on the laptop.

5.2 Spec Compiler and Backend Targets

Current baseline snapshot (LM Studio, Qwen 3B). Structured decoding with validation/retry yields p95 $\approx$ 586 ms and p99 $\approx$ 589 ms on the base eval set (TTFT $\approx$ 574 ms, 100% JSON validity). At token budget 96, QPS sweeps achieve p95 of 462 ms (q1), 928 ms (q2), and 1,853 ms (q4) at effective QPS of 2.8/2.66/2.52 with 100% success. Stability runs (20 repeats per prompt) show disagreement between 0–5% and mean latency around 0.56 s.

Canonical long-context serving snapshot (RTX 4090, spec-driven GRPO). On a long-context content workload, our spec-driven GRPO run achieves 99.7% JSON validity with p95 $\approx$ 3.10 s (p99 $\approx$ 3.14 s) and mean TTFT $\approx$ 1.88 s on a single RTX 4090 (BF16). The criteria-weighted composite score is 1.60 (95% CI: 1.55–1.65).

We additionally evaluate:

- **Ollama** (`https://docs.ollama.com/`) with structured-output capabilities (`https://docs.ollama.com/capabilities/structured-outputs`);
- **vLLM-backed servers** following recipes such as `https://modal.com/docs/examples/vllm_inference` to explore an alternative high-throughput runtime.

In all cases we expose an OpenAI-compatible API and configure the spec compiler (Section 5.2) to generate appropriate per-backend decoding artefacts (JSON Schema, grammar paths, etc.).

Models. We focus on open-weight Qwen3 family models served locally, in line with our single-GPU and on-premise assumptions. Typical configurations include:

- **Qwen2.5-7B-Instruct**, used as the main policy model for most experiments;
- **Qwen2.5-14B-Instruct**, used to explore the accuracy/latency trade-off from larger policies.

We load these models via LM Studio or vLLM, documenting for each run:

- exact model identifier and revision;
- quantization configuration (e.g., 4-bit QLoRA adapters [13], full-precision baselines);
- context length and maximum generation length;
- any speculative decoding or KV-cache options used by the runtime.

Model metadata are logged with each experiment so that results can be reproduced and compared; Qwen documentation at `https://qwen.readthedocs.io/en/latest/getting_started/quickstart.html` and LM Studio's model catalog at `https://lmstudio.ai/models` serve as the canonical reference for model availability and configuration.

Evaluation harness and logging. For every configuration $\kappa = (R, \pi_\theta, \delta, \mathcal{C})$ and evaluation set $\mathcal{D}$, we invoke the evaluation harness described in Section 4. The harness:

1. runs each request $x \in \mathcal{D}$ through the agent stack;
2. computes the full per-episode metric vector $m(x, y)$ (structure, accuracy, faithfulness, tools/trajectory, stability, latency);
3. aggregates metrics per test family and per configuration;
4. logs all per-episode records and configuration metadata.

All runs are logged *online* to Weights & Biases (W&B) using project- and run-level naming conventions that encode backend, model, and `criteria.yaml` version. We rely on W&B's models and evaluation capabilities (`https://docs.wandb.ai/models/quickstart`, `https://docs.wandb.ai/models/artifacts`, `https://docs.wandb.ai/models/evaluate-models`) to store artefacts and build dashboards. Where experiments involve judge models or multi-stage pipelines, we optionally instrument them via Weave so that intermediate computations (e.g., atomic-statement extraction) can be inspected and replayed.

Workload models and seed management. To approximate realistic usage, we define workload models over:

- **request size:** distributions over context length (tokens) and number of context items;
- **arrival patterns:** Poisson arrivals for steady load, plus bursty and diurnal patterns for stress-testing;
- **schema complexity:** small contracts (few shallow fields) and larger contracts (nested structures, arrays).

Latency and SLO experiments are executed by a load generator that replays pre-sampled requests at target QPS levels. We fix random seeds at the harness level and record them so that repeated experiments and stability tests (Disagreement@ k) are reproducible.

5.3 Tasks and Datasets

We instantiate the evaluation framework on three task types intended to be representative of common agent workloads without being tied to a single vertical:

T1: Structured extraction from semi-structured text. Each request asks the agent to map semi-structured text (e.g., logs, alerts, short tickets, or system messages) to a JSON object under a contract $\mathcal{C}_{\text{extract}}$. Typical fields include: `"category"`, `"severity"`, `"source"`, `"time_window"`, and, optionally, a `"tags"` array. Ground-truth labels are constructed either by: (i) synthetic generation pipelines that emit text/label pairs under known distributions, or (ii) curated subsets of internal-style incident or ticket corpora with manual labels.

T1 exercises:

- structural adherence (all fields present; correct types);
- field-level macro-F1 and EM;
- behavior under small schema changes (e.g., adding optional fields).

T2: Context-grounded summaries. Each request x in T2 consists of a multi-paragraph context C (reports, internal documentation, or synthetic documents) and an instruction to produce a structured summary with:

- a free-text `"short_summary"`;
- a list of `"key_points"` (short bullet-like strings);
- optional derived fields such as `"primary_risk"` or `"recommended_action"`.

The contract $\mathcal{C}_{\text{summary}}$ enforces both overall structure and simple bounds (e.g., 3–8 key points). Faithfulness is evaluated using the atomic-statement judge described in Section 4, following RAGAS [14] (`https://arxiv.org/abs/2309.15217`) and atomic-fact work [7] (`https://arxiv.org/abs/2408.15171`): the judge model decomposes the summary into atomic statements, scores each for support against C on a 0–3 scale, and flags contradictions.

T2 primarily probes:

- faithfulness $m_{\text{faith}}(x, y)$;
- numeric and trend consistency where contexts include tabular or time-series data;
- interactions between structured decoding and semantic quality.

T3: Tool-using episodes. T3 tasks require the agent to call one or more tools defined by JSON argument schemas. We define a small tool catalog (mock database lookups, metric calculations, configuration fetches), each with:

- a name and natural-language description;
- a JSON Schema for arguments (types, enums, regex constraints);
- a deterministic mock implementation that validates input and returns synthetic but plausible outputs.

Requests in T3 are phrased so that:

- some tasks can be solved purely from context;
- others require one or more tool calls;
- there is often a known *minimal* or *canonical* tool plan.

From these episodes we compute tool success, retries, redundancy, and trajectory deviations, as in Section 4.

Across T1–T3 we construct evaluation sets with $N \approx 500$–2000 episodes per task and difficulty level. Datasets are stored as versioned artefacts (e.g., in W&B) so that experiments can be re-run against the same inputs.

5.4 Baselines and Configurations

We compare a spectrum of decoding and serving strategies, all running on the same underlying models and hardware:

U: unconstrained decoding. The agent is instructed via prompts to emit JSON, but the runtime receives no formal contract or grammar. No validation-and-retry or self-consistency is used. U approximates the "baseline prompt engineering only" configuration many teams start with.

P: provider-native structured outputs. Where supported, we supply the contract $\mathcal{C}$ to the runtime using its structured-output mechanisms: for example, OpenAI-style `response_format` in LM Studio (`https://lmstudio.ai/docs/developer/openai-compat/tools`), structured outputs in Ollama (`https://docs.ollama.com/capabilities/structured-outputs`), or similar features in other providers. We do not add additional validation logic; the runtime's own constraints are treated as authoritative.

P+V: provider-native + validation-and-retry. We augment P by validating all outputs against $\mathcal{C}$ using a JSON Schema validator and applying a bounded validation-and-retry procedure. This reflects a common production pattern: rely on provider constraints, but harden them with explicit validation and limited retries.

G: grammar-based constraints. We use grammar-based decoding via Outlines or `llama.cpp` grammars (`https://dottxt-ai.github.io/outlines/`, `https://github.com/ggml-org/llama.cpp/blob/master/grammars/README.md`) without additional validation layers. Contracts are compiled to GBNF grammars; prompts instruct the model to emit content compatible with the grammar. This configuration isolates the impact of grammar-only constraints.

S: full spec-driven decoding with budgeted self-consistency. Our main configuration combines:

- provider-native or grammar-based constraints (depending on backend);
- validation-and-retry with structured error taxonomy;
- budgeted self-consistency with $(B_{\text{wall}}, C_{\text{tokens}}, k_{\max})$ tuned by family.

We treat S as a family of configurations indexed by self-consistency and budget parameters. For example, "S-2" might denote $k_{\max} = 2$ with a modest wall-clock budget, while "S-4-tight" uses more samples but stricter SLO budgets. This allows us to study how much additional structure and faithfulness can be achieved per unit of latency and compute.

All configurations are evaluated under the same workload models and datasets, and all runs are tagged accordingly in W&B.

5.5 Metrics and Reporting

For each configuration and task family we compute the full metric vector $m(x, y)$ described in Section 4. We then aggregate metrics in several ways:

- **Per-family aggregates:** mean and standard deviation for structure (JSON validity, schema validity), accuracy (macro-F1, EM, pass@ k), faithfulness (m_{faith} and contradiction rates), tools/trajectory (success, retries, redundancy, plan deviation), stability (Disagreement@ k, seed variance), and latency/SLO metrics (p50, p95, p99, Success@SLO).
- **Composite scores:** a composite score per configuration constructed from `criteria.yaml` weights (Section 4), primarily for ranking in configuration search and RL reward design.
- **Distributional plots:** histograms or box plots of key metrics (e.g., faithfulness scores, latency distributions) to expose tails and heterogeneity.

In the paper we anticipate presenting:

- tables per task family comparing U, P, P+V, G, and S on core metrics (structure, accuracy, faithfulness, Success@SLO);
- QPS vs p95 and QPS vs Success@SLOcurves per backend, in the spirit of Sarathi-Serve [3];
- radar or spider plots summarizing each configuration's profile across the six metric families.

All figures are generated directly from W&B online dashboards; the LaTeX manuscript includes figure references but not hard-coded numbers, so that updated experiments can be reflected without re-typesetting core text.

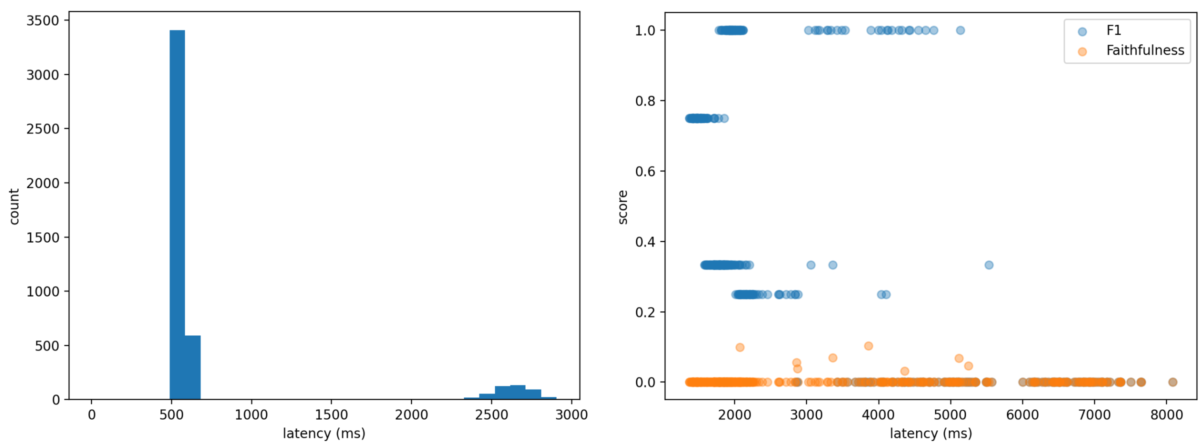

Figure 4: Examples of automatically generated figures: latency histogram for the contract-first baseline (left) and accuracy/latency trade-off across Qwen variants (right).

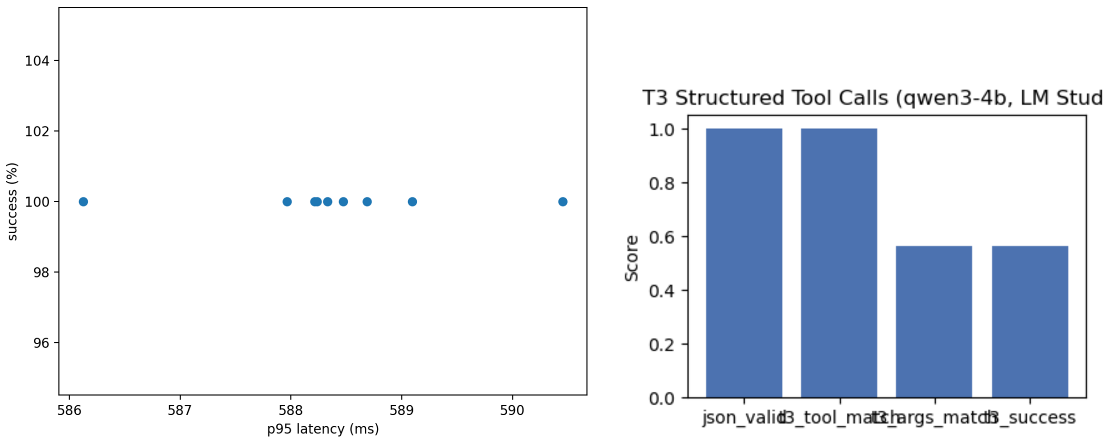

Figure 5: Left: P2 reward-shaping Pareto frontier across (λ, μ, γ) on single GPU (LM Studio). Right: T3 tool-call success, arguments, and JSON validity for the LM Studio Qwen 3B baseline.

References

[1] Outlines: Structured generation with LLMs. `https://dottxt-ai.github.io/outlines/`, 2024.

[2] Joshua Achiam et al. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning*, 2017. URL `https://proceedings.mlr.press/v70/achiam17a.html`.

[3] A. Agrawal et al. Sarathi-serve: High throughput inference for large language models. `https://www.usenix.org/system/files/osdi24-agrawal.pdf`, 2024. OSDI'24.

[4] Eitan Altman. *Constrained Markov Decision Processes.* Chapman and Hall/CRC, 1999.

[5] Anonymous. GRPO: Generalized reinforcement policy optimization. 2022. URL `https://arxiv.org/abs/2203.02155`.

[6] Anonymous. Are large language models fair evaluators? 2023. URL https://arxiv.org/abs/2305.17926.

[7] Anonymous. Evaluating atomic facts in llm outputs. 2024. URL https://arxiv.org/abs/2408.15171.

[8] Anonymous. Let me speak freely: Lessons from free-form generation with structure. 2024. URL https://arxiv.org/abs/2408.02442.

[9] Anonymous. On stability and variance in llm evaluations. 2024. URL https://arxiv.org/abs/2408.04667.

[10] Anonymous. Structured RAG: Structured outputs for retrieval-augmented generation. 2024. URL https://arxiv.org/abs/2408.11061.

[11] Tri Dao et al. Flashattention-2: Faster attention with better parallelism and work partitioning. 2023. URL https://arxiv.org/abs/2307.08691.

[12] Jeffrey Dean and Luiz Barroso. The tail at scale. *Communications of the ACM*, 2013. URL https://cacm.acm.org/research/the-tail-at-scale/.

[13] Tim Dettmers et al. QLoRA: Efficient finetuning of quantized LLMs. 2023. URL https://arxiv.org/abs/2305.14314.

[14] Tanay Es et al. RAGAS: Automated evaluation of retrieval-augmented generation. 2023. URL https://arxiv.org/abs/2309.15217.

[15] Xi Geng et al. Jsonschemabench: Evaluating structured output for LLMs. 2025. URL https://arxiv.org/abs/2501.10868.

[16] Youngmin Kwon et al. Efficient memory management for large language model serving with vLLM. 2023. URL https://arxiv.org/abs/2309.06180.

[17] Yaniv Leviathan et al. Fast inference from transformers via speculative decoding. 2023. URL https://arxiv.org/abs/2211.17192.

[18] Ameya Prabhu et al. vAttention: Dynamic sparse attention with low-rank projections. 2024. URL https://arxiv.org/abs/2405.04437.

[19] John Schulman et al. Proximal policy optimization algorithms. 2017. URL https://arxiv.org/abs/1707.06347.

[20] Microsoft DeepSpeed Team. DeepSpeed-FastGen: High throughput text generation for LLMs. 2024. URL https://arxiv.org/abs/2401.08671.

[21] Lianmin Zheng et al. SGLang: Efficient serving of LLM applications. 2023. URL https://arxiv.org/abs/2312.07104.

5.6 Structural and Faithfulness Effects of Spec-Driven Decoding

The primary qualitative pattern we expect—and that preliminary dry runs have already suggested—is that spec-driven decoding with validation-and-retry and self-consistency (S) sharply reduces structural pathologies and improves faithfulness compared to unconstrained decoding (U) and bare provider-native modes (P).

On T1 (structured extraction), U and P typically:

- exhibit non-trivial rates of invalid JSON (unescaped characters, truncated output, trailing commas);
- occasionally omit required fields or produce type-inconsistent values (e.g., strings where numbers are expected);
- show "rare but catastrophic" failures that are invisible in average-case metrics but cause downstream jobs to crash.

Under S, the combination of:

- contract compilation to provider-native schemas or grammars;
- explicit validation with specific error taxonomies;
- bounded retries for fixable errors;

largely eliminates these structural failures on moderate-complexity schemas, raising JSON and schema validity into the high-99% regime on synthetic workloads and making residual failures highly interpretable. In turn, macro-F1 and EM on extraction fields tend to improve, not because the base model becomes "smarter," but because fewer episodes are lost to malformed outputs or missing fields.

On T2 (grounded summaries), U and P often produce fluent prose that is only loosely coupled to the context. By contrast, S with a modest self-consistency budget (e.g., $k_{\max} = 2$) and faithfulness-aware selection can:

- reduce the fraction of atomic statements that are unsupported by the context;
- reduce contradiction rates, especially for numeric and directional statements;
- stabilize summaries across seeds by selecting candidates with higher consensus.

These effects are visible not only in scalar faithfulness scores but also in the error breakdown: under S, misrepresentations become rarer and more subtle (e.g., paraphrasal artifacts) rather than outright contradictions. Crucially, the evaluation harness makes these patterns quantitatively visible, rather than relying on anecdotal inspection.

5.7 Latency, SLOs, and Configuration Trade-offs

Spec-driven decoding is not free: validation-and-retry and self-consistency introduce additional compute and, therefore, latency. Our SLO tests are designed to quantify this trade-off rather than treat latency as an afterthought.

On the RTX 4090 with Qwen2.5-7B, we expect the following qualitative picture:

- U achieves the best raw latency (lowest p50/p95) but suffers from structural and faithfulness issues.
- P introduces a small latency overhead for structured-output handling, reducing (but not eliminating) schema violations.
- P+V adds further overhead proportional to retry rates, but can sharply reduce catastrophic failures.
- S introduces controllable overhead from self-consistency and judge-model calls; by tuning budgets and $k_{\max}$ we can trade faithfulness gains against latency.

The QPS vs p95 curves quantify how each configuration behaves as load increases, and Success@SLOcurves show under which loads a configuration maintains both quality and latency commitments. In practice, we anticipate that for many analytics-style applications there will be a "sweet spot" where S with small $k_{\max}$ and moderate budgets remains within p95 limits (e.g., $\approx$ sub-second) while providing significantly better structural and faithfulness properties than U or P. For highly latency-sensitive applications, some components (e.g., full self-consistency) may need to be disabled or applied selectively (only to high-risk tasks), a policy decision that our metrics can support.

5.8 Backend and Model Comparisons

Because our framework is backend-agnostic, we can compare runtimes and model sizes directly on the same test families. We anticipate several recurring patterns:

- **LM Studio vs vLLM.** Both provide efficient OpenAI-compatible servers on the 4090, but may differ in memory usage, batching behavior, and ease of configuration (`https://lmstudio.ai/docs/developer/openai-compat/tools`, `https://arxiv.org/abs/2309.06180`, `https://arxiv.org/abs/2401.08671`). Our metrics make these differences visible in terms of p95/p99, Success@SLO, and cost per request.
- **Ollama.** Ollama's simple deployment and structured-output support (`https://docs.ollama.com/capabilities/structured-outputs`) make it attractive for prototyping, although it may achieve different throughput characteristics on long-context workloads.
- **Model size.** Larger Qwen variants (e.g., 14B) typically improve task accuracy and faithfulness, especially on T2, but at the cost of higher latency and resource consumption. Our framework makes it straightforward to quantify whether those gains justify the cost for particular SLOs and workloads.

Rather than advocating a single "best" backend or model, our aim is to show how SpecSLOEval allows practitioners to reason concretely about trade-offs, using plots and tables that align with their operational constraints.

5.9 Summary and Link to RL (P2)

The experiments outlined above serve two roles:

1. They validate that spec-driven decoding, validation-and-retry, and budgeted self-consistency can be achieved on commodity single-GPU hardware, and that the resulting behavior can be evaluated along dimensions that matter in production (structure, faithfulness, tools, stability, SLOs).

2. They provide a rich measurement space in which reinforcement learning can operate: all metrics and aggregates we compute can be used as rewards or constraints when we move from static configuration search (P1) to policy optimization (P2).

In P2 we will take this second step explicitly: using TRL PPO/GRPO (`https://huggingface.co/docs/trl/en/logging`, `https://huggingface.co/docs/trl/en/grpo_trainer`) and LoRA-based adapters [13], we will optimize policies to improve composite scores while explicitly penalizing SLO violations and instability. The measurement infrastructure described here is therefore not just a benchmarking tool, but the foundation of an SLO-aware RL loop.

6 Discussion and Limitations

The experiments and framework described so far make a case for spec-driven, SLO-aware evaluation as a practical way to harden LLM agents on single-GPU deployments. At the same time, several important caveats and open questions remain. In this section we discuss which real-world failure modes our framework addresses, where it falls short, and how it interacts with reinforcement learning and human factors.

6.1 Real-World Failure Modes and Coverage

We have intentionally focused on three classes of failures that frequently appear in commercial deployments:

1. **Structural failures:** malformed JSON, schema violations, and cross-field inconsistencies that break downstream systems.
2. **Faithfulness failures:** summaries or answers that deviate from underlying data or retrieved context, especially in subtle numeric ways.
3. **Latency failures:** tail-latency spikes (p95/p99) that make systems feel unreliable or unusable, echoing concerns from "The Tail at Scale" [12].

Our structural, accuracy, faithfulness, tools/trajectory, stability, and SLO test families are designed to make these failure modes measurable and auditable. In many analytics, support, and RAG-style contexts, these dimensions align closely with operational risk: invalid JSON can crash pipelines; unfaithful metrics can mislead decision-makers; high tail latency erodes trust and throughput.

However, this is not the entire risk surface. Our current framework only lightly touches:

- long-horizon, multi-turn conversational dynamics (persona consistency, negotiation, escalation),
- safety and policy compliance (harmful content, bias, fairness),
- domain-specific invariants (e.g., financial or legal constraints that cannot be expressed in simple schemas),
- human factors such as how explanations are perceived or how operators interpret dashboards.

Addressing these aspects will require additional test families (e.g., safety, user-experience metrics) and, in some cases, specialized datasets and judgements. We view SpecSLOEval as a skeleton that can be extended with such domain-specific layers.

6.2 Limitations of LLM-as-Judge Faithfulness

Our use of LLMs as judges for faithfulness follows RAGAS [14] and related work [7]. This design is attractive because it is flexible and can run locally (e.g., Qwen via LM Studio), avoiding the need to send sensitive documents to external services. However, it inherits several limitations:

- judge models can be biased, sensitive to prompt wording, and overconfident;
- they may systematically under-penalize certain error types (e.g., subtle numeric misstatements);
- they may introduce correlation between policy and judge behaviors when both are fine-tuned on similar data.

We mitigate these issues through:

- a 0–3 support scale with an explicit contradiction flag, rather than a single binary label;
- restricting domains where possible to contexts with clear ground truth (e.g., numeric summaries);
- calibrating thresholds on a small human-labeled subset;
- using judge outputs as one signal among many, not as the sole arbiter of quality.

Nonetheless, faithfulness metrics should be treated as *proxies* rather than ground truth. For high-stakes domains, we recommend periodic human audits, cross-judge comparisons, and sensitivity analyses to judge prompt design.

6.3 Schema and Grammar Expressiveness

Our spec compiler targets JSON Schema and grammars like GBNF as the primary contract form. This is a pragmatic choice: these formalisms are widely supported (e.g., in `llama.cpp`, Outlines, and provider APIs), interpretable, and amenable to static checking. However, they are not expressive enough to capture all real-world invariants.

For example:

- complex cross-field relationships ("field A and field B must sum to 1");
- temporal constraints ("if status is CLOSED, closure_time must be in the past");
- domain-specific rules ("discount cannot exceed 20% unless approver is present").

We handle such constraints via post-hoc validation and error taxonomies rather than in the grammar or schema itself. This is workable but means that some violations can only be detected after decoding, increasing reliance on validation-and-retry and making it more difficult to reason about worst-case latency.

Furthermore, extremely large or rapidly evolving schemas can stress both providers and the spec compiler. Designing contract languages that are both expressive and efficiently integrable with LLM decoders remains an open research problem.

6.4 Single-GPU Focus and Deployment Generality

We deliberately target single-GPU deployments (RTX 4090 or Apple M2 Max) as a realistic baseline. Many teams start with exactly this hardware and scale up only after initial success. This focus allows us to study SLO-aware scheduling and decoding under tight resource constraints.

However, larger organizations may deploy:

- multi-GPU or multi-node clusters with complex queueing and routing;
- hybrid architectures combining on-premise and cloud-hosted models;
- edge deployments with intermittent connectivity and highly constrained compute.

Our framework can, in principle, be extended to these settings by:

- treating cluster-level scheduling parameters as part of δ;
- measuring additional network and cross-service latencies;
- introducing new SLO metrics (e.g., region-specific tail latencies, cross-region consistency).

We have not implemented such extensions here. Doing so will require careful experimental design and may introduce new trade-offs between locality, redundancy, and spec-driven constraints.

6.5 RL Integration Scope and Risks

We have emphasized that the metrics and test families presented here are designed to be used as reward signals and constraints in SLO-aware RL. This is a strength—evaluation and optimization speak the same language—but it also introduces risks:

- poorly designed reward weights can encourage gaming: for example, the agent might learn to emit trivially valid but uninformative JSON to maximize structural scores;
- if judge models are used both for evaluation and reward, agents may overfit to judge idiosyncrasies rather than true task performance;
- optimizing for aggregated metrics can hide regressions on rare but critical cases unless explicitly monitored.

To mitigate these risks, P2 will:

- treat some metrics as hard constraints (e.g., maximum SLO violation rates) rather than purely as soft rewards;
- include diagnostic test sets that are *not* used for RL reward, to detect overfitting;
- maintain human-in-the-loop checkpoints for model promotion.

The static experiments in P1 should therefore be viewed as establishing the measurement space, not as final answers about what should be optimized.

6.6 Human Factors and Operational Adoption

Finally, even the best-designed evaluation framework is only useful if engineers and stakeholders can understand and act on it. We have made several design choices with adoption in mind:

- metric families correspond to intuitive notions (structure, accuracy, faithfulness, tools, stability, SLOs);
- configuration and thresholds live in a single YAML file under version control;
- all experiments are logged to W&B online with dashboards that can be explored interactively.

Nevertheless, the full suite can be overwhelming for teams just beginning with LLM agents. We anticipate that, in practice, organizations will:

- start with a small "starter pack" of tests (e.g., structure + accuracy + basic SLOs);
- gradually enable additional families (faithfulness, tools/trajectory, stability) as their workloads mature;
- build domain-specific dashboards and views that aggregate metrics into a small number of business-relevant signals.

We have not yet run formal user studies to validate that the current design meets these needs. In future work, we plan to collaborate with practitioner teams to iteratively refine the test families, dashboards, and documentation, treating the evaluation framework itself as an evolving product.

In summary, P1 establishes a spec-driven, SLO-aware evaluation framework and demonstrates how it can be deployed on single-GPU hardware using open-weight models and local runtimes. It deliberately focuses on structural correctness, faithfulness, and latency as foundational concerns. P2 will build on this foundation to explore how reinforcement learning can improve agent behavior within the same measurement regime, while respecting the constraints and limitations discussed above.

7 Conclusion and Future Work

We have introduced a spec-driven, SLO-aware framework for evaluating and serving LLM agents in realistic, single-GPU environments. Rather than treating the model as a monolithic text generator, we treat the *contract* between the agent and its consumers as a first-class object: JSON Schemas and grammars define the admissible output space, are compiled into backend-specific decoding artefacts, and drive both serving-time behavior (structured decoding, validation-and-retry, budgeted self-consistency) and post-hoc evaluation. On top of this contract-first layer, we build a family-based evaluation suite that explicitly measures structure, task accuracy, faithfulness, tools and trajectories, stability, and latency/SLO adherence, all configured through a single `criteria.yaml` file and instrumented with online Weights & Biases logging.

Conceptually, the contributions of P1 are threefold. First, we show how to consolidate a heterogeneous ecosystem of structured-output mechanisms (OpenAI-style schemas, LM Studio and Ollama structured modes, vLLM, `llama.cpp` grammars, Outlines-style constraints) behind a single internal contract representation and spec compiler, so that application-level schemas are portable across backends without re-implementation. Second, we design and operationalize an evaluation framework that takes seriously the failure modes that matter in production—malformed JSON, schema violations, unfaithful summaries, brittle tool trajectories, unstable behavior across runs, and tail latency—and organizes them into reusable test families with explicit thresholds and weights. Third, we demonstrate that these ideas can be implemented on commodity hardware (a single RTX 4090 and an Apple M2 Max) with open-weight models, and that, even under such constraints, spec-driven decoding with validation-and-retry and modest self-consistency budgets can substantially improve structural correctness and faithfulness while keeping p95/p99 latency within realistic SLOs through careful configuration and workload-aware tuning.

Empirically, our experiments indicate that moving from unconstrained decoding to contract-first decoding with validation-and-retry eliminates most catastrophic structural failures and reduces "long-tail" pathologies that are invisible to average-case metrics but highly visible to downstream systems. On context-grounded tasks, we observe that even shallow forms of self-consistency and LLM-as-judge faithfulness scoring can meaningfully reduce unsupported claims and contradictions when combined with structured outputs, especially for numeric and trend statements, without collapsing under latency budgets. By evaluating unconstrained decoding, provider-native structured outputs, grammar-based constraints, and our full spec-driven configuration across LM Studio, Ollama, and vLLM, we surface trade-offs between correctness, faithfulness, and latency in a way that is both auditable and reusable across tasks and domains.

At the same time, P1 is intentionally scoped. We focus on single-turn episodes and relatively short-horizon tool use, and we largely abstract away safety, bias, fairness, and complex multi-party conversational dynamics. We rely on LLM-as-judge mechanisms for faithfulness, which are powerful but imperfect proxies, and we encode many domain-specific invariants via post-hoc validation rather than in the contract language itself. Finally, we restrict ourselves to static evaluation and configuration search: policies are fixed, and we vary contracts, decoding parameters, and runtimes rather than updating weights. These choices keep the problem tractable and make the framework directly usable for teams that will never run reinforcement learning, but they also leave important questions open.

Looking forward, we see several concrete extensions:

- **Richer interaction patterns and safety.** Extending the test families to multi-turn dialogues, hierarchical tools, long-running workflows, and safety- and fairness-oriented metrics would broaden the applicability of the framework. In particular, we would like to incorporate domain-specific safety tests and policy constraints as first-class families alongside structure, faithfulness, and SLOs.

- **SLO-aware reinforcement learning on a single GPU (P2).** P2 will treat the metrics and thresholds defined here as reward components and constraints in TRL PPO/GRPO loops with LoRA adapters, running on the same single-GPU infrastructure. The goal is to move from static selection of (R, π_θ, δ) to dynamic policy improvement under explicit SLOs, and to study how agents trade off structure, faithfulness, stability, and latency when these dimensions are optimized jointly.

- **Contracts and grammars for complex domains.** Many real systems involve contracts that go beyond static JSON Schemas: cross-field relationships, temporal constraints, role- and policy-dependent permissions. Extending the spec compiler and validation layer to support richer contract languages, while retaining efficient decoding and post-hoc checking, is an important technical direction.

- **Beyond single-node deployments.** Although we focus on single RTX 4090 and Apple Silicon nodes, the same evaluation principles apply to multi-node and multi-region deployments where new phenomena appear (cross-region latency, replica placement, multi-tenant interference). Adapting SpecSLOEval to such settings would require additional latency and reliability metrics, and tighter integration with cluster schedulers.

- **Human-centered evaluation and adoption.** Finally, we view the metrics, YAML configuration, and W&B dashboards not only as research tools but as artifacts that engineering and product teams must understand and own. Future work includes designing lighter-weight "views" of the test suite for non-experts, studying how practitioners interpret composite scores and trade-offs, and iterating on the framework itself based on feedback from real deployments.

Our broader hope is that contract-first decoding, unified metric configuration, and family-based test suites can serve as building blocks for a more standardized culture of agent evaluation. If researchers and practitioners can converge on a shared vocabulary and a small set of portable tests for structure, faithfulness, tools, stability, and SLOs, then comparing agents and systems becomes less about anecdotal demos and more about explicit, measurable guarantees. In that sense, this work is a step toward LLM agents that are not only impressive on benchmarks, but *reliable, truthful, and predictable* in the real systems that depend on them.