The Control-Plane Bottleneck in Agentic Workloads: A Trace-Based Capacity Model

George Weale
Columbia University
New York, NY, USA
george.weale@columbia.edu

***Abstract*—Agentic applications combine model inference with orchestration: tool-schema validation, credential acquisition, request serialization, retrieval, retry control, state persistence, and external I/O. Capacity discussions often compress this heterogeneous execution into tokens per second and accelerator utilization. This paper develops a trace-based model in which every agent run is a typed directed acyclic graph and each node consumes one or more resource classes. The model makes control-plane CPU demand observable while preserving the possibility that inference, memory, network, or an external tool service is the binding resource. It integrates a workload contract, a conservative stability screen, a critical-path latency decomposition, and an evaluation methodology for comparing heterogeneous bottlenecks. The resulting decision procedure ties capacity claims to reproducible traces and controlled counterfactuals, reducing the risk of provisioning inference capacity while leaving the workflow that makes inference useful underprovisioned.**

***Index Terms*—agentic systems, capacity planning, control plane, observability, queueing, CPU, GPU, workflow traces**

I. Introduction

An agent run is not one model request. A practical run may plan, invoke a tool, validate a structured response, read or write state, recover from a timeout, ask for authorization, and invoke another model. ReAct-style interleaving of reasoning and actions makes this loop explicit [1]; tool-use training and tool-augmented benchmarks make it increasingly common [2], [3]. Model serving remains important, and techniques such as PagedAttention change its memory and throughput profile [4]. Yet a useful capacity model must account for the non-inference work between model calls.

This paper examines when, for a declared agent workload and service-level objective (SLO), added accelerator capacity has less effect on end-to-end latency or throughput than added CPU-side orchestration capacity, lower per-step overhead, or a healthier external dependency. "Control plane" denotes the software that coordinates work rather than any particular cloud product.

The paper contributes a typed run model, resource-demand equations, a reproducible measurement contract, and a controlled evaluation methodology. Together, these elements support operational capacity analysis grounded in workload behavior rather than a model name, token count, or anecdotal tool use.

II. Motivation and Related Work

The motivating failure mode is an incomplete resource boundary. In a simple completion service, request arrival, model execution, and response delivery may be close enough that accelerator utilization is an informative primary signal. In an agent system, a single user request may create several queues with different policies: an inference queue, an executor queue, a credential or authorization queue, a connection pool, a remote API quota, and a durable state writer. A slow node can amplify work elsewhere. For example, a timeout may create a retry, the retry may repeat validation and serialization, and a backlog may cause a caller to issue a second request before the first side effect is observed. The model treats these as causal events rather than as unexplained latency around a model call.

Recent work establishes the relevance of model-side serving optimization. Orca separates scheduling of generative-model requests to improve serving efficiency [8]; vLLM exposes memory-management techniques that improve throughput for autoregressive inference [4]; and phase-splitting designs separate prompt processing from token generation [9]. These contributions are complementary to this paper. They address efficient inference serving, whereas the trace model places inference within the larger tool-mediated workflow. Neither a high GPU utilization value nor a low token latency alone establishes that the agent's critical path is accelerator-bound.

The systems literature also cautions against optimizing the mean while ignoring the path that determines tail behavior. Dean and Barroso describe how large fan-out services magnify tail latency [10]. The SRE literature similarly treats retries, load shedding, and dependency failure as first-class design concerns [11]. Agent traces make these concerns concrete because they record an action graph with repeated tool invocation and policy checks. The present design adopts a narrower stance than generic observability: it defines resource demand and causal dependency at the typed-event level so that a reported bottleneck can be reproduced under a stated workload contract.

Tool-use research supplies the application setting. ReAct makes the alternating reasoning-action loop explicit [1]; Toolformer studies learned API invocation [2]; and API-Bank provides a benchmark setting for tool-augmented language models [3]. The contribution here is a capacity method that can identify model inference, database service, network I/O, CPU-side coordination, or a third-party rate limit as the binding

constraint under a declared workload.

III. Workload and Observation Contract

For a run r, let $G_r = (V_r, E_r)$ be a directed acyclic execution graph. Each vertex $v \in V_r$ has type

$$k(v) \in \{\mathsf{infer}, \mathsf{dispatch}, \mathsf{validate}, \mathsf{serialize}, \mathsf{auth}, \mathsf{state}, \mathsf{io}, \mathsf{retry}\}.$$

An edge represents a causal dependency, not merely temporal adjacency. A trace event is the typed record

$$e = (r, v, k, t_s, t_f, c, g, m, n, q, x),$$

where t_s, t_f are monotonic timestamps; c, g, m, n are CPU time, accelerator time, memory-byte seconds, and network bytes; q is a queue or dependency outcome; and x is a versioned attribute map. The collector MUST record clock source, sampling policy, redaction policy, and trace-loss rate. OpenTelemetry supplies a useful interoperability vocabulary, but a trace contract must still define application-specific event meanings [5].

Let $R = \{\mathsf{cpu}, \mathsf{gpu}, \mathsf{mem}, \mathsf{net}, \mathsf{tool}\}$. The demand of resource j for run r is

$$D_{r,j} = \sum_{v \in V_r} d_j(v), \qquad d_j(v) \geq 0. \tag{1}$$

The control-plane CPU share is a descriptive statistic, not a verdict:

$$\chi_r = \frac{\sum_{k(v) \neq \mathsf{infer}} d_{\mathsf{cpu}}(v)}{D_{r,\mathsf{cpu}}}. \tag{2}$$

It is only meaningful with a workload contract $\mathcal{C}$ fixing model revision, prompt and tool schemas, concurrency policy, cache state, authentication mode, retry policy, hardware, runtime versions, workload mix, and observation boundaries. Without $\mathcal{C}$, comparing traces risks measuring configuration drift rather than architecture.

IV. Capacity Model

Let arrivals be a stationary-enough interval with rate λ, and let n_j be the effective parallel servers available to resource j. A first stability screen is

$$\rho_j = \frac{\lambda \mathbb{E}[D_{r,j}]}{n_j} < \rho_j^{\max} < 1. \tag{3}$$

Equation (3) is intentionally a screen, not an assertion that each component is an $M/M/k$ queue. Real services have batching, bursty arrivals, correlated retries, and rate limits. It is useful precisely because it flags resource demand that token-only accounting hides. Little's law, $L = \lambda W$, provides a cross-check between observed queue depth and delay [6].

For a realized graph, end-to-end latency is separated into execution, waiting, and external delay:

$$T_r = \max_{p \in \mathcal{P}(G_r)} \sum_{v \in p} [s(v) + w(v) + b(v)], \tag{4}$$

where s, w, and b denote service, local queueing, and blocking/dependency delay. The critical path avoids a misleading aggregate: a CPU-intensive branch that is not on the path may affect throughput but not a particular run's latency.

Given an SLO vector $\mathbf{z}$ (for example, p95 latency, completed runs per second, and error budget), a candidate capacity vector $\mathbf{n}$ is feasible only if a predeclared estimator reports each constraint within its confidence interval. The planning problem is

$$\min_{\mathbf{n} \in \mathbb{Z}_+^{|R|}} \ \mathrm{Cost}(\mathbf{n}) \quad \text{s.t.} \quad \widehat{\mathbf{z}}(\mathbf{n}, \mathcal{C}) \preceq \mathbf{z}_{\max}. \tag{5}$$

A trace supports the description "CPU bottleneck" only when a controlled counterfactual shows that changing CPU-side capacity or overhead improves the relevant outcome more than the compared alternatives under the same contract.

V. Trace Protocol and Testable Hypotheses

The protocol assigns a run identifier at ingress and propagates it through inference, workers, queues, and tool adapters. It retains a sampled causal graph, not prompt content by default. Each tool event records declared idempotency, side-effect class, timeout budget, retry number, payload-size bucket, and outcome. Each inference event records model route, token counts where permitted, batching state, and service versus queue time. Sensitive values are replaced by stable hashes or coarse buckets before export.

The smallest useful unit is a run manifest. It should name the workload generator; model endpoint and revision; executor implementation; tool schemas; authorization mode; cache warming procedure; resource limits; and every dependency stub or live service. It should also record exclusions: trace sampling, dropped spans, client-side versus provider-side time, clock synchronization method, and events deliberately unavailable for privacy reasons. These exclusions matter because a trace can give false confidence when the unseen portion contains the dominant service time.

The trace schema must separate a requested action from an observed side effect. A dispatch span indicates an attempted call. A tool-result span indicates a returned response. A state-observation span indicates whether a declared postcondition was later observed. This distinction prevents a capacity study from counting a fast failure as useful throughput. It also permits retry analysis: retry work may be attributed to a transient network outcome, a semantic validation failure, an authorization denial, or an intentionally rejected unsafe operation.

The primary hypotheses are: (H1) the non-inference CPU share χ rises with tool-call fan-out and retry depth; (H2) at fixed workload and SLO, the marginal benefit of CPU-side changes can exceed the marginal benefit of accelerator additions; and (H3) p95 latency degradation can be localized more accurately by critical-path attribution than by aggregate host utilization. These hypotheses can fail. A workload dominated by long generations, a slow database, or a provider rate limit should not be relabeled as a CPU problem.

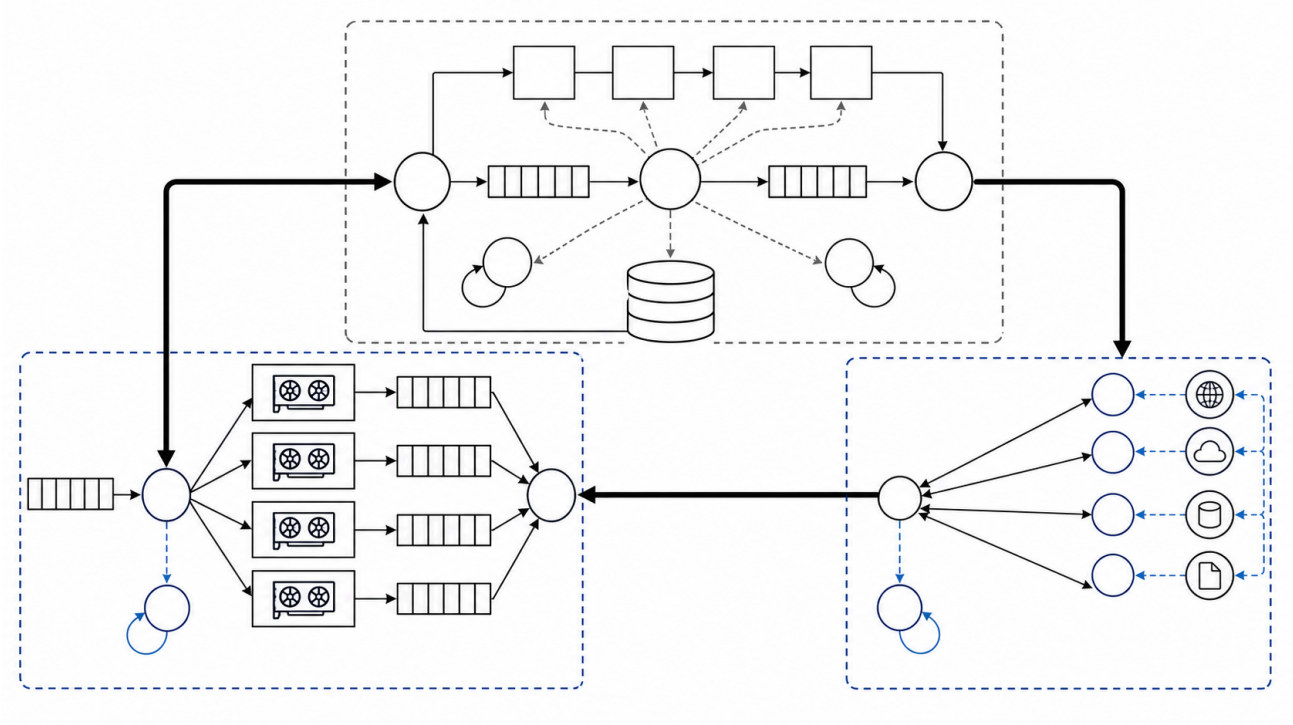

Fig. 1. Heterogeneous agent workflow with distinct inference, orchestration, and external-service paths.

TABLE I
TRACE AND MANIFEST ARTIFACTS FOR REPRODUCIBLE CAPACITY ANALYSIS.

Artifact	Minimum recorded content	Why it is needed
Workload manifest	Task family, arrival generator, seed or trace digest, completion predicate, concurrency policy	Separates a workload change from a capacity change.
Inference event	Route, queue time, service time when observable, token buckets, batching state, outcome	Places model execution on the causal path without exposing prompt content.
Control-plane event	Dispatch, validation, serialization, authorization, cache, or persistence type; CPU time; queue time	Makes non-inference work attributable rather than residual.
Tool event	Adapter revision, deadline, idempotency declaration, retry number, response class, side-effect observation	Distinguishes attempted calls from completed and observed effects.
Environment manifest	Runtime, hardware, worker limits, dependency mode, cache warming, redaction and sampling policy	Enables reproduction and reveals hidden configuration drift.
Failure dossier	Error class, dropped-span count, relevant critical path, policy-compliant completion status	Prevents an apparent throughput gain from masking unsuccessful work.

VI. EVALUATION METHODOLOGY

The evaluation uses three reproducible workload families: a single-turn retrieval baseline; a bounded tool workflow with schema validation and state writes; and a branching workflow with induced retryable failures. Each family includes fixture data, tool stubs, a seed, a trace schema, a machine description, and a manifest of all versions.

The experiment varies CPU workers, accelerator replicas, request concurrency, tool latency, and retry probability one factor at a time, then with a fractional factorial design. Primary outcomes are completed runs per second, p50/p95/p99 end-to-end latency, error-budget consumption, χ, critical-path resource attribution, and cost per successful run. Report medians and percentile intervals across independently randomized repetitions; bootstrap intervals are appropriate for skewed latency data [7]. A result is actionable only when the workload contract, trace completeness, rejected runs, and confidence intervals accompany it.

For each factor, the evaluator first establishes a steady-state observation window and separately retains warm-up and drain periods. A capacity comparison uses matched arrival traces or a seeded generator, then reports the degree of matching. The primary contrast is not "CPU versus GPU" in isolation; it is the change in successful, policy-compliant completions when one resource or one implementation overhead is altered while the remaining contract is held fixed. When a remote tool is the limiting service, the analysis identifies a tool-service bottleneck even if local CPU utilization is high.

An interpretable report includes three views: resource utilization over time, the distribution of typed critical-path con-

tributions, and an error taxonomy. The first view is operationally familiar but insufficient; the second explains why delay occurred; the third guards against an apparent throughput gain caused by more failures or less validation. Together they connect each capacity conclusion to the causal path and completion quality that produced it.

An ablation removes validation, persistence, and retry logic separately. This identifies work that is merely incidental in a prototype versus necessary in a safe production workflow. Failure injection should include tool timeouts, malformed JSON, permission denials, cache misses, and partial state-write acknowledgements. The study must distinguish useful completion from syntactically successful HTTP responses.

VII. Decision Procedure and Reporting

The capacity procedure begins by rejecting a vague workload description. A study owner first declares a task mix in terms of typed run families, not only prompts or average token counts. For each family, the manifest records tool-call fan-out, maximum execution depth, retry policy, side-effect policy, cache policy, and success predicate. A workload generator then emits either a reproducible trace or a distribution with a recorded seed. This makes a later comparison interpretable when one experiment has more retryable failures, a different cache temperature, or a different fraction of stateful actions.

Second, the investigator checks trace integrity before modeling capacity. The report should quantify completed spans, orphaned spans, clock-skew corrections, dropped events, and opaque provider intervals. If model service time is unavailable because it occurs behind a managed endpoint, the report must state that boundary rather than assigning the hidden time to local orchestration. If a tool returns a response before its externally visible effect is committed, the trace should record the distinction. These details often decide whether a run is useful, retryable, or merely fast.

Third, the investigator evaluates a small set of controlled changes. Examples include increasing executor workers; reducing schema-validation allocations; changing connection-pool limits; adding model replicas; or replacing a serial state-write path with a safe batched path. Each change is evaluated against a matched workload, one stated SLO, and an unchanged correctness policy. An observed improvement counts only if the completion predicate remains satisfied and if the improvement exceeds both measurement noise and the degradation imposed elsewhere. For example, fewer retries may improve CPU utilization while hiding a loss of error detection; the error taxonomy would reveal that trade-off.

Finally, the result should be written as a bounded decision record. It names the workload range, hardware and dependency configuration, SLO, confidence interval, rejected runs, and the exact counterfactual supported. A defensible sentence is: "Under contract $\mathcal{C}$, changing component x improved declared outcome y within the observed range." An indefensible sentence is: "Agents need CPUs more than GPUs." The latter erases workload type, implementation, model, dependency behavior, and scale. The point of the method is not rhetorical advocacy for one processor class; it is a repeatable way to locate the resource or boundary whose change most improves a safe agent outcome.

A reproducible implementation combines a trace release and controlled sandbox rather than relying on a benchmark score alone. It includes a reference implementation of the event schema, workload manifests, fault-injection adapters, capacity-estimation scripts, and a redaction test suite. Independent reproduction matters because instrumentation and aggregation choices can change an apparent bottleneck.

VIII. Reproducibility and Audit Record

An implementation packages the study as an executable audit record rather than a slide deck. The record begins with a signed experiment manifest that identifies the code revision, runtime image digest, resource allocation, model route, tool schemas, fixture set, and workload generator. It then names the exact command or workflow used to start each run, the collection interval, and the storage location of raw traces. An independent reviewer should be able to recreate the declared environment, rerun a bounded workload, and compare the resulting event schema even if absolute latencies differ across hardware.

Trace redaction needs its own test plan. A redactor should be evaluated against representative prompts, tool payloads, identifiers, and error objects to establish that prohibited fields are not exported through attributes, exception strings, or baggage. The redaction policy should distinguish direct removal, stable hashing, coarse bucketing, and retention inside a protected enclave. It should also retain enough structural information to detect a changed causal path. For example, removing a customer identifier may be appropriate, while removing the distinction between an authorization denial and a timeout may make the capacity model unreviewable.

The audit record should retain transformations in order. Raw spans are immutable inputs. A parser produces typed events; an attribution pass assigns resources and graph edges; an analysis pass estimates critical paths and SLO statistics; and a reporting pass produces tables or plots. Each pass should record version, input digest, output digest, and warnings. This discipline lets a reviewer ask whether a conclusion changed because the workload changed, because attribution rules changed, or because a renderer rounded an interval differently. It also makes it possible to rerun an analysis after discovering a trace-collection defect without silently replacing the original evidence.

Finally, the protocol should specify what must be published or disclosed for a negative result. If a capacity change has no measurable effect, the study should still publish workload coverage, rejected runs, trace completeness, and the smallest diagnostic artifact that explains the null result. If a third-party tool or managed model obscures a required interval, the conclusion should be narrowed accordingly. These conventions make failure to validate the CPU hypothesis informative rather than embarrassing, and they guard against selective reporting

of only the workloads that reinforce a preferred hardware narrative.

IX. Limitations and Responsible Interpretation

Resource use is workload-, implementation-, and provider-dependent. CPU time observed by a client may omit managed-service work; sampling can miss short tasks; tracing changes performance; and a capacity-optimal point can be unacceptable for resilience or privacy. An agent's model quality, tool correctness, and authorization policy are also outside equations (1)–(5). The model therefore supports bounded engineering decisions rather than a general prediction that CPUs replace GPUs. Its value lies in identifying the typed work on the path to a successful, safe outcome.

X. Conclusion

This paper treats agent infrastructure as a heterogeneous workflow rather than an accelerator-only serving problem. Typed traces, a declared contract, and controlled resource counterfactuals turn an intuitive claim about "control-plane bottlenecks" into a reproducible capacity analysis. The framework supports disciplined capacity decisions by connecting resource changes to critical paths, workload conditions, completion quality, and service-level objectives.

References

[1] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in *ICLR*, 2023.

[2] T. Schick et al., "Toolformer: Language models can teach themselves to use tools," in *NeurIPS*, 2023.

[3] J. Li et al., "API-Bank: A comprehensive benchmark for tool-augmented LLMs," in *EMNLP*, 2023.

[4] W. Kwon et al., "Efficient memory management for large language model serving with PagedAttention," in *SOSP*, 2023.

[5] OpenTelemetry Authors, "OpenTelemetry specification," 2026. [Online]. Available: https://opentelemetry.io/docs/specs/otel/

[6] J. D. C. Little, "A proof for the queueing formula: $L = \lambda W$," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.

[7] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. New York, NY, USA: Chapman and Hall, 1993.

[8] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. K. Lee, and B.-G. Chun, "Orca: A distributed serving system for transformer-based generative models," in *OSDI*, 2022, pp. 521–538.

[9] P. Patel et al., "Splitwise: Efficient generative LLM inference using phase splitting," arXiv:2311.18677, 2023.

[10] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[11] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., *Site Reliability Engineering: How Google Runs Production Systems*. Sebastopol, CA, USA: O'Reilly Media, 2016.

[12] L. Kleinrock, *Queueing Systems, Volume 1: Theory*. New York, NY, USA: Wiley, 1975.