The Cloud Should Adapt to Its Applications, Not the Other Way Around!

Sachin Ashok[†*] Karuna Grewal[§*] Talha Waheed[†] Gangmuk Lim[†] Gerard Matthew[†¶]
Aditya S Prerepa[†] P. Brighten Godfrey[†] Justin Hsu[§] Radhika Mittal[†]
[†]University of Illinois Urbana-Champaign, USA [§]Cornell University, USA [¶]Amazon, USA

Abstract

Modern cloud infrastructure uses various controllers (deployed within service meshes and orchestrators like Kubernetes) that assist applications with scheduling, auto-scaling, load-balancing, etc. Knowledge of certain application properties or *insights* can enable these controllers to better optimize performance. However, today, controllers either treat the application as an opaque box, or adopt a piecemeal approach to extract specific insights from specific applications. We argue that the emergence of LLM agents changes this dynamic. It is now feasible for the infrastructure to understand the application, in a general and flexible sense. We explore the architectural implications of this shift. We envision an *application insight plane* (`AIP`) that leverages LLM agents to extract useful insights from any given cloud application through reasoning across code, configuration, telemetry, analysis tools, and controlled experiments. `AIP` makes these insights available for reuse across cloud controllers and AI agents.

We evaluate an example of this approach by testing one LLM-based concurrency-model extractor on eight disparate microservice applications, and by showing how this insight enables a controller to reduce a service's tail latency by up to 30%. Realizing this vision raises open questions: which insights to standardize, how to keep them valid as applications change, and how to extract and share them without compromising privacy or security.

CCS Concepts

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → *Abstraction, modeling and modularity*; • **Computing methodologies** → Intelligent agents.

[*]Equal contribution.



This work is licensed under a Creative Commons Attribution 4.0 International License.
HotNets '26, Salt Lake City, UT, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859957

Keywords

Application insight plane, Cloud infrastructure, LLM agents, Application-aware resource management, Microservices

ACM Reference Format:
Sachin Ashok, Karuna Grewal, Talha Waheed, Gangmuk Lim, Gerard Matthew, Aditya S Prerepa, P. Brighten Godfrey, Justin Hsu, and Radhika Mittal. 2026. The Cloud Should Adapt to Its Applications, Not the Other Way Around!. In *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26), November 16–17, 2026, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3847290.3859957

1 Introduction

Cloud infrastructure has traditionally offered generic resources and mechanisms, leaving applications and their developers to adapt around them. Developers specify resource needs and replica counts in deployment configurations, tune thread and connection pools, and implement queues, retries, admission control, or backpressure to handle overload and resource variability. In recent years, some of this adaptation logic has moved out of applications and into the infrastructure via *cloud controllers* such as schedulers, autoscalers, load balancers, and overload controllers [12]. This shift has reduced developer burden and made common mechanisms reusable across applications. But while the logic moved, application-specific knowledge did not. Properties such as call structure, execution behavior, and resource demands remain buried in code, configuration, and operator experience.

Certain controllers do try to infer application properties, largely via external observation of metrics, logs, traces, and RPC events. One commonly used property is the application's call graph, which informs overload control, routing, tracing, and critical-path analysis [56, 64, 77, 79]. Another is its resource profile, which informs autoscaling, resource allocation, request routing, and cluster scheduling [15, 44, 56, 65].

But broadly, cloud controllers' knowledge of applications remains limited, in two ways. First, *such knowledge is bespoke rather than general purpose.* Each controller typically extracts the particular property needed for its own decision, using specific methods and representations that other controllers cannot easily reuse. Some also rely on application

instrumentation, developer annotations, or assumptions tied to a particular language or runtime. The result is fragmented application knowledge rather than a reusable understanding that can support a broader range of applications and control tasks. Second, *many useful application properties are difficult for cloud controllers to extract.* For example, Envoy's adaptive concurrency filter [25] searches online for an upper limit on the number of in-flight requests. Knowledge of the service's execution model, such as its worker pool, queue limits, or blocking behavior, could give the controller a better starting point and constrain its search, but this information is hidden inside application code. As another example, a load balancer could perform much better if it knew which request types are latency-sensitive, but that information might only appear in design and requirements documentation.

This paper is centered around a simple thesis: cloud infrastructure should understand the applications it manages. That is, infrastructure should have explicit, queryable knowledge of application structure and behavior, like call graphs, concurrency limits, resource demands, and performance relationships. The thesis rests on two key arguments.

First, we argue that application properties, rather than being discovered piecemeal across today's fragmented landscape of controller-specific extractors, should be treated as a coherent class of reusable knowledge that any cloud controller can query to make informed decisions. We refer to these properties as *application insights*. They capture how an application is structured, how it executes requests, and how its workload and resource demands shape performance. In §2, we present a taxonomy that organizes them into two classes: structural insights, such as call graphs and concurrency models, and performance insights, such as resource profiles and workload-to-performance relationships.

Second, we argue that we are at an inflection point where generic extraction of application insights is practical. Traditional methods were limited mostly to external observation, and even then required a custom analyzer for every application and property. Today, LLM-based agents change what is practical. As we discuss in §3.1, they can reason across multiple types of input: static artifacts such as source code, configuration, and documentation; runtime evidence such as logs, metrics, traces, and events; and active observations produced by analyzers, load generators, or controlled experiments. Critically, LLM agents can also select, combine, and invoke existing tools (e.g., profilers), adapt them to an application's characteristics, form hypotheses about its behavior, and run targeted experiments in a sandbox. They can therefore combine application-specific reasoning with specialized analyzers, supporting a wider range of insights and applications without a new end-to-end pipeline for every case. This creates an opportunity to make application understanding a shared, general-purpose capability of cloud infrastructure.

We tap into this opportunity and propose an *Application Insight Plane* (`AIP`) – a framework for extracting relevant insights from any given application and making them easily available to different cloud controllers. `AIP` organizes specialized *insight providers* behind a common discovery and query interface. Each provider maintains one family of insights, combining LLM-based reasoning with specialized tools as the application requires, and records each result together with where it applies, how it was derived, and when it should be updated. The resulting insights can be reused by conventional controllers, AI agents, and human operators – the plane's *consumers* – while each consumer remains free to decide how to act on them. What distinguishes `AIP` from today's bespoke extractors is this treatment of application insights as a shared class of knowledge that the infrastructure keeps scoped, valid, and reusable as the application changes, rather than as the private output of one controller.

Building such a plane, however, raises four challenges that assembling existing analyzers behind an API would not solve. **Generality.** A provider must recover an insight, such as a concurrency model, from a Java thread pool, a Python worker pool, or a Node.js event loop alike, or abstain when the evidence is insufficient. **Validity.** An inferred property can be wrong, incomplete, or valid only under particular conditions. For example, a service's concurrency limit depends on whether its thread pool or its database connection pool saturates first, which varies with request mix. A limit set too low throttles the service, so a wrong insight can be worse than none and a consumer must know when one applies. **Reuse.** Providers and consumers are developed independently, so an insight must not presuppose how a consumer will use it: a concurrency model extracted once should serve both an overload controller limiting in-flight requests and an autoscaler scaling replica counts. **Trust.** Tenants may not want to expose code, configuration, and telemetry to the cloud operator, derived insights can reveal implementation details, and a provider that reads repositories and logs can be manipulated by instructions planted in them. Each challenge concerns a different point in an insight's life: how it is extracted, whether it still holds, where it applies, and who may access it. The difficulty is thus not only in producing an insight once but in maintaining it throughout its lifecycle. `AIP` offers initial mechanisms for this (§3), and §5 lays out the research agenda.

As a first step, we evaluate one provider end to end in §4 to gather preliminary evidence that insights can be general and useful. We test whether an LLM-based concurrency insight provider can recover worker pools, queue limits, and blocking behavior across eight microservice applications. We then augment Envoy's concurrency controller to consume the extracted concurrency model and guide its control logic. This augmented controller, `AIP-Envoy`, reduces tail latency

by up to 30% compared to default Envoy. Together, these results are an initial step toward the plane we envision, in which insights are extracted once, kept valid as applications change, and reused by any controller, and toward the broader thesis that infrastructure should understand the applications it manages rather than treat them as opaque boxes.

2 Application Insights

We list representative *application insights* that cloud controllers can use, in two classes: *structural insights*, describing how an application is built, and *performance insights*, describing how it behaves. The list is not exhaustive. Its purpose is to establish what a shared insight plane must be able to represent and, for each insight, to show how controllers could use it and to what extent it can be extracted today.

2.1 Structural Insights

Aspects of the application's *functional design*.

(I1) Service Call Graph. A request sent to a microservice typically begins at a front end, but can result in several backend calls to other microservices which in turn call other microservices. The service call graph captures this invocation structure – each vertex in the call graph is a microservice (or a request at that microservice), and an edge $A \rightarrow B$ means that A results in another request to some microservice B.

Use case. A call graph lets controllers anticipate a request's downstream effect. For example, an autoscaler observing a burst of requests at service A can predict load at a downstream service B and pre-warm its replicas. Knowledge of call graph has also been used to inform overload control, routing, tracing, and critical-path analysis [56, 64, 77, 79].

Extraction. Call graph information is primarily encoded in application source code or binaries, but can also be inferred from externally visible communication patterns. Static dependency analysis [24, 59] and binary analysis [5, 60] can recover possible call edges, while instrumentation and distributed trace analysis-based techniques [1, 13, 17, 18, 28, 66] can extract paths taken at runtime. However, the calls a request makes may depend on its parameters or on exceptional conditions. The techniques also differ in reach: source-level analyses are often language-specific, while binary and trace-based ones apply more broadly but expose less detail.

(I2) Information flow. It offers deeper insight than the call graph by capturing data dependencies, not just the service invocations of (I1). For instance, a service A could forward a parameter of the request it received to a service B, take the output of B, modify it, and pass it to a service C, or send a batch of requests conditioned on an input parameter.

Use case. Such insights could be useful for security analysis [3, 10, 36, 39], or in understanding the provenance of a certain behavior during troubleshooting efforts.

Extraction. Information flow can be extracted by observing value propagations in runtime traces [3, 38, 40–43] or by source-level data-flow analysis.

(I3) Concurrency model. It describes how and to what extent a service instance parallelizes its work, which depends heavily on the application's frameworks (e.g., gRPC, Apache, Redis) and configuration: a framework may spawn a thread per request or use a fixed-size pool, and calls may run synchronously [22], asynchronously [8], or fire-and-forget, with results handled by one thread [67] or in fork-join style [16].

Use case. It can improve non-invasive trace reconstruction tools such as TraceWeaver [13] and WAP5 [66] by combining call graph structure (I1), information-flow dependencies (I2), and concurrency constraints (I3) to rule out impossible span mappings. In §4, we use this insight to constrain the search space of Envoy's concurrency controller.

Extraction. Source code analysis can identify concurrency primitives, like asynchronous calls, thread pools, event loops, and fork/join synchronization. Deployment and runtime configurations can supply scheduling policies, pool sizes, queue bounds, and concurrency limits. Tracing and profiling can reveal observed overlap.

2.2 Performance Insights

Quantitative aspects of the application's *runtime behavior*.

(I4) Resource profiles. They define an application's demands across multiple resources – CPU, memory, network.

Use case. Resource profiles let controllers distinguish request traffic classes by their resource demands (e.g., CPU-heavy, memory-bound, I/O-intensive), enabling load balancers to reduce bottleneck contention more effectively [6, 27, 47, 48, 76].

Extraction. Monitoring systems and profilers such as Kubernetes *metrics* [54] and Go's *pprof* [34] estimate aggregate CPU and memory demand. As with (I1), aggregate consumption is visible externally, so a coarse characterization can be measured over time. Predicting future demand is harder, because resources are multidimensional and nonlinear in load, and so is fine-grained characterization: the CPU used by one request is hard to attribute when many run concurrently. Existing techniques help only in special cases, such as eBPF-based tracing [70], which can recover thread-level usage where a request is handled by a single thread.

(I5) Performance models. These describe how application resources and workload translate to performance metrics such as request latency, throughput, or failure rate.

Use case. Such models enable informed inter-cluster request routing. For instance, SLATE [56] combines call-graph, resource, and performance insights to route selected traffic, like moving CPU-intensive requests away from an overloaded cluster while considering latency and egress cost.

Extraction. Infrastructure can monitor workload (requests per second), resource utilization, and performance in a given environment. Controlled staging experiments can vary multidimensional conditions (workload level, type, resource use, interaction with downstream calls [21, 23, 57, 72, 80], etc.) to learn more complete empirical performance models [11].
(I6) Objectives. This insight captures an application's service level objectives (SLO), e.g., a target RPS or tail latency.

Use case. Objectives can inform resource-allocation controllers. A proxy like Envoy or a load balancer can prioritize latency-sensitive traffic according to the SLOs instead of a standard FIFO order. Combined with resource profile (I4), it can favor low-resource-demand requests to implement a Shortest Job First strategy. With call graph (I1), it can delay requests that may add load to an already overloaded service.

Extraction. Objectives are defined by operators in design documents, not inferred from code or runtime behavior.
Summary. These insights vary in how reliably they can be inferred. An infrastructure provider typically sees an application as an opaque box: it can observe communication patterns, resource usage, and latency, but not internal logic or intent. Runtime observation alone may therefore fail to recover precise information-flow dependencies, concurrency semantics, or SLOs, which require interpreting source code, frameworks, or operator specifications. Extraction is also fragmented across specialized tools: source-level analyses are often language-specific, performance tools expose particular metrics at specific granularities, and controllers depend on custom-built workflows. This fragmentation becomes especially problematic when a controller must combine insights from different tools, whose outputs use incompatible formats and abstractions. Ultimately, any single method provides only an incomplete view.

If infrastructure is to understand its applications broadly, it needs a general, interoperable mechanism that makes heterogeneous insights extractable, reusable, and composable.

3 Application Insight Plane

3.1 Generalizing Extraction with LLMs

LLM agents make broader and more general insight extraction practical. Unlike a fixed analyzer, an agent can adapt its workflow to the application and available inputs: it can inspect code, configs, and documentation, interpret telemetry, invoke specialized tools, and request extra observations when needed [26, 52, 68, 74, 78]. This flexibility helps it recognize the same underlying property across heterogeneous implementations. For example, Tomcat thread pools, Gunicorn process pools, and Node.js event loops are different realizations of the same concurrency model abstraction [9, 45, 63].

LLMs can also synthesize higher-level insights that existing tools may not. Conventional analyzers may identify

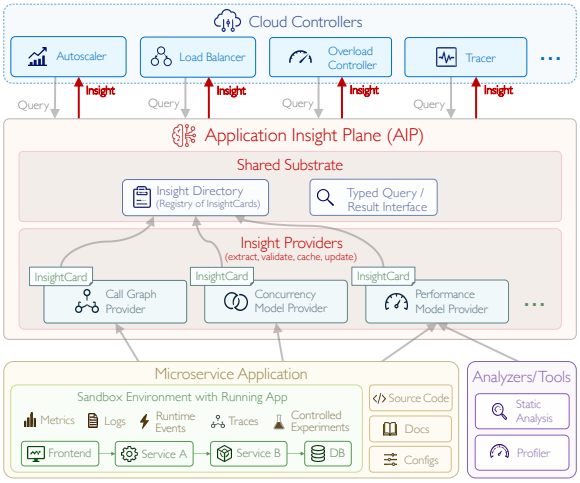


Figure 1: AIP architecture. Providers maintain insights that controllers query via a shared substrate.

RPCs, queues, worker pools, or blocking operations, but rarely relate them into a cross-framework, service-level model. An LLM-based provider can, for example, connect a server framework identified in build files, a worker limit found in configuration, and blocking behavior observed in traces into one concurrency insight. LLMs do not replace specialized analyses or guarantee correctness. Rather, they provide the semantic reasoning and orchestration needed to select and combine their outputs, while static checks, runtime validation, and experiments provide supporting evidence.

3.2 AIP Architecture Overview

AIP is organized around modular insight providers connected through a shared substrate (Figure 1). An *insight provider* is the component that extracts, validates, and maintains one family of insights, such as call graphs, concurrency models, or performance models, using artifacts, runtime observations, sandboxed executions, and analysis tools. The shared substrate supports provider discovery and common query and result interfaces, through which controllers request the typed insights their control logic needs. Providers are part of AIP but may be developed, deployed, and upgraded independently, allowing new insight types and extraction techniques to be added without modifying existing controllers.

3.3 Controller–AIP Interface

A controller requests an insight through AIP's interface. Providers register their capabilities with an *insight directory*, which AIP uses to locate and invoke a provider compatible with the controller's query. The provider may return a cached result, update an existing one, or perform a new extraction.

Provider discovery with Insight Cards. Each provider advertises an *Insight Card* describing the capability it offers:

$$\begin{aligned}\text{InsightCard} = \langle &\text{kind} : K,\ \text{accepts} : Q_K,\ \text{returns} : R_K,\\ &\text{scopes} : \Sigma_K,\ \text{requires} : E,\ \text{endpoint} : U\rangle.\end{aligned}$$

Here, kind identifies the insight family from the taxonomy in §2; accepts and returns identify its query and result schemas; scopes lists supported granularities; requires describes required evidence and permissions; and endpoint specifies how the provider is invoked. For example, a provider may advertise that it produces concurrency models for services when granted read-only access to code and configuration files.

Insight Cards are inspired by Agent Cards used for capability discovery in agent protocols [71], but specialize the abstraction for providers of application knowledge.

Query: Scope and granularity. A query specifies the desired insight and its requested scope:

$$\begin{gathered}Q_K = \langle \text{scope} : S,\ \text{parameters} : P_K\rangle,\ \text{where}\\ S ::= \text{App}(a) \mid \text{Svc}(a,s) \mid \text{Op}(a,s,o) \mid \text{Component}(a,s,c) \mid \text{TrafficClass}(a,p).\end{gathered}$$

Traffic-class predicates p may distinguish request headers, tiers, or regions.

Result: Applicability and freshness. A provider returns a value I_K together with metadata:

$$\begin{aligned}R_K = \langle &\text{kind} : K,\ \text{scope} : S',\ \text{value} : I_K,\ \text{basis} : B,\\ &\text{provenance} : V,\ \text{confidence} : C,\ \text{freshness} : F\rangle.\end{aligned}$$

The returned scope S' may be narrower than requested if only partial extraction is possible. The basis identifies the code, configuration, and workload from which the result was derived. Provenance describes how it was obtained, confidence communicates uncertainty, and freshness records when it was produced and when to update it. Because these fields are common to all kinds, consumers can handle any insight uniformly, while value keeps the schema of its kind.

These metadata fields make an insight's limitations explicit, but do not by themselves guarantee safety. Before acting on an insight, a controller should verify that its scope and basis match the deployed version, configuration, and workload, and fall back to its default policy when the evidence is insufficient, stale, or of low confidence. The concurrency limit in §4, for example, rests on a connection-hold time measured under one request mix, so a new mix makes it stale even with no code change. How to define such applicability checks, reconcile conflicting insights from multiple providers, and act on an insight's confidence value remain open questions (§5).

3.4 Insight Provider Operation

Inputs, tools, and execution boundaries. A provider may inspect code and documentation; consume metrics, traces, and runtime events; invoke specialized analyzers; or execute with controlled inputs in a sandbox. It supplies these tools with the required artifacts, reconciles their outputs, and maps them into its insight schema. Providers should operate under explicit capability boundaries analogous to those used by contemporary coding agents, including read-only or workspace-write sandboxes, approval gates, tool allow/deny lists, and restricted network access [7, 31, 62]. Depending on the workflow, a provider may receive read-only access, selected telemetry, or permission to run approved tools and experiments in a sandbox. The requires field declares these needs so AIP only invokes policy-compliant providers. Importantly, authorization to access application artifacts does not imply authorization to disclose the insights. AIP must enforce access policies on both extraction inputs and consumer queries. Exposing artifacts to AIP also widens the attack surface: a compromised provider or insight directory holds evidence about every application it serves, so the least-privilege boundaries that limit what a provider may read must also limit what the plane retains. The same centralization is what makes enforcement tractable, however, since there is one boundary to police rather than one per controller. Finally, because repositories and telemetry may contain adversarial instructions [37], provider execution must remain constrained by its granted capabilities even when an LLM is induced to misuse a tool.

Validation and maintenance. A provider may validate an inferred property against code and configuration, runtime observations, or a targeted experiment. A result can be reused while its basis remains valid, updated after changes, and withdrawn when contradicted. AIP leaves these mechanisms provider-specific but standardizes how scope, basis, provenance, confidence, and freshness are communicated.

4 Preliminary Evaluation

We test two premises: (1) that an LLM-based extractor can recover concurrency model properties across disparate applications, and (2) that such insights can improve a controller.

Concurrency extraction. We evaluate on 8 applications from 7 microservice suites: DeathStarBench, OnlineBoutique, SockShop, RobotShop, TrainTicket, TeaStore, and Spring Petclinic Microservices [29, 30, 35, 50, 58, 69, 73]. For each application, the extractor analyzes its repository, including source code, manifests, and configuration, but excludes external documentation, issue trackers, and web search; providers in practice may use any of these. The extractor uses a two-agent workflow. Both agents run GPT-5.5 via OpenAI Codex [61] at the extra-high reasoning setting. A first LLM agent extracts the component records, and a second, as a proxy for validation, independently audits them against explicit constructs in the code and configuration. The audit checks whether the source-code and configuration lines cited by the extractor for each result support the reported component type, concurrency mechanism, and limits. However, it does not

Table 1: InsightCard and Result in AIP format.

InsightCard advertised by the provider	
kind	concurrency_model
accepts	App(a), Svc(a, s), Component(a, s, c)
returns	ConcurrencyModel.v1 component records
scopes	app, service, component
requires	source, manifests, build files, configuration
Result returned for Petclinic visits service	
kind	concurrency_model
scope	Svc(petclinic, visits-backend)
value	schema: ConcurrencyModel.v1 execution_model: blocking_thread_pool limits: tomcat.max_threads=8; hikari.max_pool=32 blocking_points: JDBC call via HikariCP multiplexing: none while thread blocked
basis	source, Maven dependencies, deployment env vars
provenance	extractor pass plus independent audit
confidence	0.88

execute the services. The audit found several omissions and overly broad labels in the first pass, which a second pass corrected to produce the final output. The 7 applications with available source contain 100 services in total. The provider extracted models for all; the audit confirmed artifact support, not runtime correctness. SockShop (16 services) provides a useful negative case: its repository contained manifests and container-images, but no service source. These artifacts alone did not expose enough evidence to recover concurrency semantics, so the provider abstained for all 16 services rather than hallucinating.

Using the concurrency model. Envoy's adaptive concurrency filter [25] controls the number of in-flight requests by adjusting a limit based on observed latency. Without knowledge of the backend's finite concurrency resources, it may search over limits well above the backend's capacity, building internal queues. A concurrency model exposes these bottlenecks; a small adapter maps its fields to an upper bound without otherwise changing Envoy's algorithm. AIP-Envoy queries AIP for the model, which routes the request to a compatible provider and returns its result. Table 1 shows the provider's advertised InsightCard and the returned concurrency model result for Spring Petclinic's visits service [69]. Next, we test the optimization on this request path:

loadgen → gateway → Envoy → visits-backend → MySQL

The extractor identifies visits-backend as a blocking Tomcat [9] service that accesses MySQL through HikariCP [75], a JDBC database-connection pool. HikariCP's maximumPoolSize bounds the number of database connections; when all connections are in use, new requests wait. This creates an application-specific concurrency limit that Envoy's generic controller does not know. Default Envoy therefore searches under a configured maximum of 1000, far above the backend's actual capacity. Our AIP-Envoy uses the same controller but bounds its maximum through a small adapter:

$$C_{\max} = \min(T_{\text{tomcat}}, \lfloor P_{\text{hikari}}/\rho \rfloor) + 1, \qquad \rho = t_{\text{DB}}/t_{\text{req}}.$$

Here, T_{tomcat} is the configured Tomcat worker-pool size, P_{hikari} is HikariCP's configured max pool size, and ρ is the fraction of request time a database connection is held. The provider obtains T_{tomcat} and P_{hikari} from code and estimates ρ through profiling, yielding $\rho \approx 0.91$. Because ρ is profiled, $C_{\max}$ inherits the workload under which it was measured: a mix with more database-bound requests raises ρ and lowers the bound, so $C_{\max}$ must be updated when the mix shifts.

Experimental setup. We use two configurations to shift the bottleneck: config A uses 8 Tomcat threads and 32 Hikari connections, making the Tomcat worker pool limiting and giving $C_{\max} = 8 + 1 = 9$; config B uses 32 threads and 4 connections, making the database connection pool limiting and giving $C_{\max} = \lfloor 4/0.91 \rfloor + 1 = 5$. We compare AIP-Envoy, which uses $C_{\max}$, with default Envoy over a 90-second run per load point, excluding the first 20 seconds as warmup.

Results. Figure 2 shows that AIP-Envoy preserves performance at low loads and lowers tail latency under overload. In configuration A, it reduces p95 from 110.1 to 77.1 ms at 140 RPS and from 111.7 to 90.9 ms at 200 RPS, with at most 0.4% lower goodput. In configuration B, it prevents queuing behind the Hikari pool, reducing p95 by 24.6% at 80 RPS and 20.8% at 100 RPS with no material goodput loss.

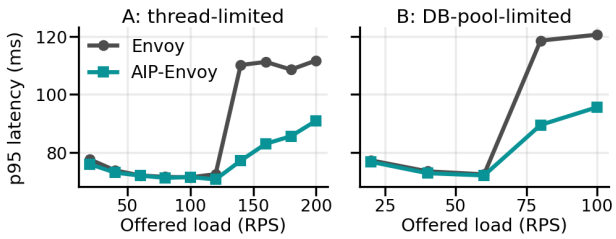


Figure 2: p95 latency under configurations A and B.

5 Directions for Future Work

Our prototype shows the promise of an LLM-driven application insight plane but surfaces some key open questions.

(1) Generalizing insight extraction. Extraction must generalize along two axes: across insight families and across the evidence available, since code, configuration, telemetry, or documentation may each be missing. AIP providers should in principle gain generality by combining LLM reasoning with specialized tools, but our evaluation currently tests only one family (concurrency models), and only with full source available. SockShop shows what happens otherwise: with manifests and container images but no source, the provider is forced to abstain rather than guess. What evidence short of source would suffice, and whether the approach extends to other insight families, are open questions.

(2) Establishing validity. Insights may be incorrect, incomplete, or valid only for a particular configuration or workload. Recording scope, confidence, and freshness makes these limitations explicit. Our prototype uses a second agent to judge whether artifacts support the first agent's claims, but this does not validate runtime behavior or future applicability. Validation is also needed on the consuming side: from scope, basis, and confidence, a controller must decide whether an insight still applies and how to fall back when it does not (§3.3). Future providers should build on emerging techniques for correctness checking and runtime validation.

(3) Standardizing abstractions. Each insight schema should capture reusable application knowledge without embedding controller policy. A concurrency model can serve multiple controllers: Envoy may use its capacity limits, while a tracer uses its asynchronous and blocking semantics. The challenge is to serve diverse controllers while supporting partial extraction, evolution, and backward compatibility.

(4) Preserving privacy and trust. AIP raises privacy concerns when extracting and sharing insights. Extraction may require sensitive source code, configuration, or telemetry that application owners may not want to expose. Moreover, the derived insights themselves may reveal sensitive details. An overload controller needs only a service's concurrency limit, not the code or database configuration behind it, yet our prototype's adapter receives the pool sizes and computes the limit itself. Computing it in the provider instead would expose only the limit. This motivates tenant-local extraction, fine-grained access controls, and sharing insights at the granularity consumers need. LLM-based extraction also introduces security risks (§3.4). Instructions planted in code or logs could induce an agent to misuse its tools, and active experiments could disrupt the application. Sandboxing, restricted tool permissions, and approval gates can mitigate these risks. Enforcing such protections while preserving the flexibility of insight extraction remains an open challenge.

6 Related Work

Application-aware cloud control. Prior systems exploit some application properties to improve overload control, routing, tracing, autoscaling, and resource allocation [15, 44, 56, 64, 65, 77, 79]. They establish the value of application knowledge but typically couple a task-specific extractor to one controller. AIP instead treats recurring properties as reusable insights that can be maintained independently.

Program analysis, observability, and LLM-based monitoring. Static and binary analysis, tracing, profiling, and controlled experiments recover complementary views of application behavior [3, 13, 14, 32–34, 49, 53, 66]. Their applicability depends on the language, runtime, and instrumentation, while their outputs differ in scope and form. LLM-based monitoring builds on the same evidence: LLMs now recommend root causes and mitigations for cloud incidents [2, 20]. These systems extract insights, as AIP does, but each produces output for one consumer (an operator or a diagnosis pipeline) in that consumer's terms, with no scope, basis, or validity fields that would let another controller reuse it. AIP does not replace these techniques; providers may combine them and publish their results through a common interface.

Agentic system design and operation. The closest related vision is Self-Defining Systems (SDS) [4], in which agents design, deploy, and evolve infrastructure tailored to an application; VibeServe realizes this idea for LLM serving stacks [51]. SDO and STRATUS use agent-built application context for deployment, diagnosis, and repair [19, 55], while Glia uses agentic experimentation to design system mechanisms [46]. These systems and AIP share a technique: an agent that reads code and telemetry, invokes tools, and checks its own claims. They differ in what the understanding is for and who holds it. These systems acquire application context for a particular design or operational task, and that context lives inside the task loop or, in SDO, in the agent's memory for its later tasks. AIP instead treats application understanding as a reusable infrastructure capability: providers expose typed, scoped insights with provenance and validity, and independent controllers, agentic or conventional, decide how to act on them. An agentic SRE could consume AIP insights and publish the context it builds during an incident through an AIP provider for others to reuse. The contribution is thus not LLM-based analysis or application-aware optimization alone but the abstraction, maintenance, and reuse of application knowledge across independent consumers.

7 Conclusion

Cloud infrastructure today observes applications but rarely understands them. We argue that controllers should have access to explicit, reusable knowledge of application structure and behavior. AIP offers a path toward this vision by making such insights discoverable and extensible, with LLM-enabled providers broadening what can be extracted. The next era of cloud control should be built around application understanding: defining what the cloud can and must know, and building new forms of control on that knowledge.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Ratul Mahajan, for their feedback. This work was supported by National Science Foundation awards 2312714 and 2152831, and a LinkedIn Ph.D. Fellowship.

References

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 74–89. https://doi.org/10.1145/945445.945454

[2] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending Root-Cause and Mitigation Steps for Cloud Incidents Using Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1737–1749. https://doi.org/10.1109/ICSE48619.2023.00149

[3] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure serverless computing using dynamic information flow control. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 118:1–118:26. https://doi.org/10.1145/3276488

[4] Thomas Anderson, Ratul Mahajan, Simon Peter, and Luke Zettlemoyer. 2025. Self-Defining Systems. White paper. https://sds.cs.washington.edu/ Accessed July 17, 2026.

[5] Angr. 2025. Angr. https://angr.io/. Accessed: 2025-07-05.

[6] Jonatha Anselmi. 2020. Combining Size-Based Load Balancing with Round-Robin for Scalable Low Latency. *IEEE Trans. Parallel Distrib. Syst.* 31, 4 (April 2020), 886–896. https://doi.org/10.1109/TPDS.2019.2950621

[7] Anthropic. [n. d.]. Claude Code: CLI Reference. https://docs.anthropic.com/en/docs/claude-code/cli-usage. Accessed July 17, 2026.

[8] Apache. [n. d.]. Apache http server project. https://httpd.apache.org/.

[9] Apache Software Foundation. [n. d.]. Apache Tomcat Configuration Reference: The Executor (Thread Pool). https://tomcat.apache.org/tomcat-10.1-doc/config/executor.html. Accessed July 17, 2026.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[11] Sachin Ashok, Sai Surya Duvvuri, Nagarajan Natarajan, Venkata N. Padmanabhan, Sundararajan Sellamanickam, and Johannes Gehrke. 2020. iBox: Internet in a Box. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (Virtual Event, USA) *(HotNets '20)*. Association for Computing Machinery, New York, NY, USA, 23–29. https://doi.org/10.1145/3422604.3425935

[12] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. 2021. Leveraging Service Meshes as a New Network Layer. In *Proceedings of the 20th ACM Workshop on Hot Topics in Networks* (Virtual Event, United Kingdom) *(HotNets '21)*. Association for Computing Machinery, New York, NY, USA, 229–236. https://doi.org/10.1145/3484266.3487379

[13] Sachin Ashok, Vipul Harsh, Brighten Godfrey, Radhika Mittal, Srinivasan Parthasarathy, and Larisa Shwartz. 2024. TraceWeaver: Distributed Request Tracing for Microservices Without Application Modification. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 828–842. https://doi.org/10.1145/3651890.3672254

[14] Sachin Ashok, Shubham Tiwari, Nagarajan Natarajan, Venkata N. Padmanabhan, and Sundararajan Sellamanickam. 2022. Data-Driven Network Path Simulation with iBox. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 6 (Feb. 2022), 26 pages. https://doi.org/10.1145/3508026

[15] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2023. Cilantro: Performance-Aware Resource Allocation for General Objectives via Online Feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 623–643. https://www.usenix.org/conference/osdi23/presentation/bhardwaj

[16] Celery. [n. d.]. Celery: Distributed Task Queue. https://docs.celeryq.dev/en/stable/.

[17] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. 2004. Path-based failure and evolution management. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1* (San Francisco, California) *(NSDI'04)*. USENIX Association, USA, 23.

[18] Xu Chen, Ming Zhang, Z. Morley Mao, and Paramvir Bahl. 2008. Automating network application dependency discovery: experiences, limitations, and new solutions. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 117–130.

[19] Yinfang Chen, Jiaqi Pan, Jackson Clark, Yiming Su, Noah Zheutlin, Bhavya Bhavya, Rohan Arora, Yu Deng, Saurabh Jha, and Tianyin Xu. 2025. STRATUS: A Multi-Agent System for Autonomous Reliability Engineering of Modern Clouds. In *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2506.02009

[20] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 674–688. https://doi.org/10.1145/3627703.3629553

[21] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 217–231.

[22] Google Cloud. [n. d.]. Building Scalable and Resilient Web Applications on Google Cloud Platform. https://cloud.google.com/architecture/scalable-and-resilient-apps?hl=en.

[23] Charlie Curtsinger and Emery D Berger. 2015. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 184–197.

[24] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

[25] Envoy Project Authors. 2026. Adaptive Concurrency. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/adaptive_concurrency_filter. Envoy documentation, accessed July 16, 2026.

[26] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2024. Large Language Models for Code Analysis: Do LLMs Really Do Their Job?. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 829–846. https://www.usenix.org/conference/usenixsecurity24/presentation/fang

[27] Hanhua Feng, Vishal Misra, and Dan Rubenstein. 2005. Optimal state-free, size-aware dispatching for heterogeneous M/G/-type systems. *Perform. Eval.* 62, 1–4 (Oct. 2005), 475–492. https://doi.org/10.1016/j.peva.2005.07.031

[28] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: a pervasive network tracing framework. In

Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (Cambridge, MA) *(NSDI'07)*. USENIX Association, USA, 20.

[29] FudanSELab. 2026. Train Ticket: A Benchmark Microservice System. https://github.com/FudanSELab/train-ticket. GitHub repository, accessed July 16, 2026.

[30] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[31] GitHub. [n. d.]. Customizing or Disabling the Firewall for GitHub Copilot Cloud Agent. https://docs.github.com/en/copilot/how-tos/copilot-on-github/customize-copilot/customize-cloud-agent/customize-the-agent-firewall. Accessed July 17, 2026.

[32] GitHub. 2025. Go-Callvis. https://github.com/ondrajz/go-callvis

[33] GitHub. 2025. Gocyto. https://github.com/protolambda/gocyto

[34] GitHub. 2025. PProf. https://github.com/google/pprof

[35] Google Cloud. 2026. Online Boutique: A Cloud-First Microservices Demo Application. https://github.com/GoogleCloudPlatform/microservices-demo. GitHub repository, accessed July 16, 2026.

[36] Jürgen Graf, Martin Hecker, and Martin Mohr. 2013. Using JOANA for Information Flow Control in Java Programs — A Practical Guide. *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)* (01 2013).

[37] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security* (Copenhagen, Denmark) *(AISec '23)*. Association for Computing Machinery, New York, NY, USA, 79–90. https://doi.org/10.1145/3605764.3623985

[38] Karuna Grewal. 2026. Safe<T>: Automaton-Theoretic Runtime Monitoring of Blackbox Distributed Systems. In *Companion Proceedings of the 2026 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Oakland, CA, USA) *(SPLASH Companion '26)*. Association for Computing Machinery, New York, NY, USA, 18–20. https://doi.org/10.1145/3837729.3842148

[39] Karuna Grewal, Loris D'Antoni, and Justin Hsu. 2022. P4BID: information flow control in p4. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 46–60. https://doi.org/10.1145/3519939.3523717

[40] Karuna Grewal, Brighten Godfrey, and Justin Hsu. 2025. SafeTree: Expressive Tree Policies for Microservices. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 349 (Oct. 2025), 27 pages. https://doi.org/10.1145/3763127

[41] Karuna Grewal, P. Brighten Godfrey, and Justin Hsu. 2023. Expressive Policies For Microservice Networks. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 280–286. https://doi.org/10.1145/3626111.3628181

[42] Karuna Grewal, P. Brighten Godfrey, and Justin Hsu. 2026. SafeNom: Data-Aware Microservice Policies. arXiv:2609.30394 [cs.PL] https://arxiv.org/abs/2609.30394

[43] Karuna Grewal, P. Brighten Godfrey, Justin Hsu, and Umang Mathur. 2026. SafePar: Monitoring Asynchrony in Microservices. arXiv:2609.33081 [cs.PL] https://arxiv.org/abs/2609.33081

[44] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati

[45] Gunicorn Project. [n. d.]. Gunicorn Design. https://docs.gunicorn.org/en/stable/design.html. Accessed July 17, 2026.

[46] Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. 2025. Glia: A Human-Inspired AI for Automated Systems Design and Optimization. arXiv:2510.27176 [cs.DC] https://arxiv.org/abs/2510.27176

[47] Mor Harchol-Balter, Mark E. Crovella, and Cristina D. Murta. 1999. On Choosing a Task Assignment Policy for a Distributed Server System. *J. Parallel and Distrib. Comput.* 59, 2 (1999), 204–228. https://doi.org/10.1006/jpdc.1999.1577

[48] Mor Harchol-Balter, Alan Scheller-Wolf, and Andrew R. Young. 2009. Surprising results on task assignment in server farms with high-variability workloads. *SIGMETRICS Perform. Eval. Rev.* 37, 1 (June 2009), 287–298. https://doi.org/10.1145/2492101.1555383

[49] Vipul Harsh, Wenxuan Zhou, Sachin Ashok, Radhika Niranjan Mysore, Brighten Godfrey, and Sujata Banerjee. 2023. Murphy: Performance Diagnosis of Distributed Cloud Applications. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 438–451. https://doi.org/10.1145/3603269.3604877

[50] Instana. 2026. Stan's Robot Shop: Sample Microservice Application. https://github.com/instana/robot-shop. GitHub repository, accessed July 16, 2026.

[51] Keisuke Kamahori, Shihang Li, Simon Peter, and Baris Kasikci. 2026. VibeServe: Can AI Agents Build Bespoke LLM Serving Systems? arXiv:2605.06068 [cs.AI] https://arxiv.org/abs/2605.06068

[52] Donghyun Kim, Sriram Ravula, Taemin Ha, Alexandros G. Dimakis, Daehyeok Kim, and Aditya Akella. 2025. Large Language Models as Realistic Microservice Trace Generators. arXiv:2502.17439 [cs.SE] https://arxiv.org/abs/2502.17439

[53] Klee. 2025. Klee Symbolic Execution Engine. https://klee-se.org/

[54] Kubernetes. [n. d.]. GitHub - kubernetes/kube-state-metrics: Add-on agent to generate and expose cluster-level metrics. — github.com. https://github.com/kubernetes/kube-state-metrics. [Accessed 09-07-2025].

[55] Shihang Vic Li, Thomas Anderson, Ratul Mahajan, Simon Peter, Luke Zettlemoyer, and SDS Team. 2026. The Long Game: How Agents That Remember Resolve Operational Issues Faster. ACM SIGOPS Blog. https://www.sigops.org/2026/the-long-game-how-agents-that-remember-resolve-operational-issues-faster/ Accessed July 17, 2026.

[56] Gangmuk Lim, Aditya Prerepa, Brighten Godfrey, and Radhika Mittal. 2024. Opportunities and Challenges in Service Layer Traffic Engineering. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*. 352–359.

[57] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 1–28.

[58] Microservices Demo Project. 2026. Sock Shop: A Microservice Demo Application. https://github.com/microservices-demo/microservices-demo. Archived GitHub repository, accessed July 16, 2026.

[59] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg.
[60] Binary Ninja. 2025. Binary Ninja. https://binary.ninja/
[61] OpenAI. 2026. GPT-5.5 System Card. https://openai.com/index/gpt-5-5-system-card/. Accessed July 17, 2026.
[62] OpenAI. 2026. Running Codex Safely at OpenAI. https://openai.com/index/running-codex-safely/. Accessed July 17, 2026.
[63] OpenJS Foundation. [n. d.]. The Node.js Event Loop. https://nodejs.org/learn/asynchronous-work/event-loop-timers-and-nexttick. Accessed July 17, 2026.
[64] Jinwoo Park, Jaehyeong Park, Youngmok Jung, Hwijoon Lim, Hyunho Yeo, and Dongsu Han. 2024. TopFull: An Adaptive Top-Down Overload Control for SLO-Oriented Microservices. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 876–890.
[65] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*.
[66] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. 2006. WAP5: Black-Box Performance Debugging for Wide-Area Systems. In *Proceedings of the 15th International Conference on World Wide Web* (Edinburgh, Scotland) *(WWW '06)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/1135777.1135830
[67] RQ Project. [n. d.]. RQ: Workers. https://python-rq.org/docs/workers/. Accessed July 17, 2026.
[68] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems*, Vol. 36. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html
[69] Spring Petclinic. 2026. Spring Petclinic Microservices: Distributed Version of the Spring Petclinic Sample Application. https://github.com/spring-petclinic/spring-petclinic-microservices. GitHub repository, accessed July 16, 2026.
[70] Byung Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N. Chang. 2009. VPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (San Diego, California) *(USENIX'09)*. USENIX Association, USA, 19.
[71] The A2A Protocol Community. 2026. AgentCard. https://agent2agent.info/docs/concepts/agentcard/.
[72] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient performance prediction for {Large-Scale} advanced analytics. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*. 363–378.
[73] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (Milwaukee, WI, USA) *(MASCOTS '18)*. IEEE, 223–236. https://doi.org/10.1109/MASCOTS.2018.00030
[74] Wenhan Wang, Kaibo Liu, An Ran Chen, Ge Li, Zhi Jin, Gang Huang, and Lei Ma. 2024. Python Symbolic Execution with LLM-powered Code Generation. arXiv:2409.09271 [cs.SE] https://arxiv.org/abs/2409.09271
[75] Brett Wooldridge and HikariCP Contributors. 2026. HikariCP: A Solid, High-Performance JDBC Connection Pool. https://github.com/brettwooldridge/HikariCP. Project documentation, accessed July 16, 2026.
[76] Runhan Xie, Isaac Grosof, and Ziv Scully. 2024. Heavy-Traffic Optimal Size- and State-Aware Dispatching. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 1, Article 9 (Feb. 2024), 36 pages. https://doi.org/10.1145/3639035
[77] Jiali Xing, Akis Giannoukos, Paul Loh, Shuyue Wang, Justin Qiu, Henri Maxime Demoulin, Konstantinos Kallas, and Benjamin C. Lee. 2025. Rajomon: Decentralized and Coordinated Overload Control for Latency-Sensitive Microservices. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 21–36. https://www.usenix.org/conference/nsdi25/presentation/xing
[78] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*. OpenReview.net. https://openreview.net/forum?id=WE_vluYUL-X
[79] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 655–672. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou
[80] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 629–644.