Azure Functions as Scalable Agent MCP Tools: Architecture and Model-based Evaluation of Elastic, Serverless Tool Invocation for AI Agents

Roshan Kumar Choudhary
Independent Researcher
Fuquay-Varina, NC 27526, USA

ABSTRACT
Agentic AI systems call tools through the Model Context Protocol (MCP), producing bursty, latency-critical traffic that serverless platforms must absorb. This paper presents a reference architecture and a Java 21 implementation pattern for hosting MCP tools on Azure Functions, and evaluates the Consumption, Flex Consumption and Premium hosting plans using a discrete-event simulation with a Markov-modulated workload and an analytical monthly cost model. Under the stated assumptions, Flex Consumption with a small always-ready floor reached a P99 latency of about 1.1 s at roughly 8 USD per million calls, while a Premium EP2 plan reached about 1.3 s at about 89 USD per million calls; the legacy Consumption plan had a P99 of 4.7 s. Instance-allocation rate and per-instance concurrency, rather than the plan label, dominated tail latency and cost respectively, and per-call tail latency compounds to a 13–18% chance of a slow call in a 10-step agent workflow. The results are model-based, not measurements of a live deployment; they are accompanied by sensitivity and break-even analyses and by security and observability guidance.

General Terms
Cloud Computing, Serverless Computing, Performance Evaluation, Agent Systems

Keywords
Azure Functions, Model Context Protocol, MCP, serverless computing, tail latency, discrete-event simulation, Java 21, agentic AI, cost modelling

1. INTRODUCTION
LLM-based agents increasingly act through tools: they query databases, call APIs and trigger workflows during a single reasoning loop [1, 2]. The Model Context Protocol (MCP) standardises how an agent discovers and invokes such tools [3], which moves the engineering problem from "how does the model call a tool" to "where does the tool run, and what happens to it under agent-shaped traffic". That traffic is unusual. A single user prompt can fan out into tens of parallel or sequential tool calls, workloads alternate between long idle periods and sharp bursts, and the end-to-end latency of an agent is the sum of its slowest tool calls.

Azure Functions is a natural candidate host: a tool is a short, stateless request/response handler, the platform scales it automatically [4, 5], and Microsoft now ships a first-class MCP extension with a tool trigger [6]. Practitioner guidance, including an earlier version of this paper, tends to answer the hosting question qualitatively: "use the Premium plan to avoid cold starts". This paper tests that advice instead of repeating it.

Research questions. RQ1: how do the Azure Functions hosting plans (Consumption, Flex Consumption, Premium) differ in tail latency and cost under bursty agent traffic? RQ2: which platform parameters (scale-out rate, instance start-up time, per-instance concurrency, always-ready floor, idle timeout) dominate those outcomes? RQ3: how does per-call tail latency compound across multi-step agent workflows?

Contributions. (i) A reference architecture and Java 21 implementation pattern for MCP tools on Azure Functions (Sections 3–5). (ii) A transparent evaluation framework: a discrete-event simulator driven by a Markov-modulated workload, and an analytical monthly cost model, both parameterised from public documentation (Section 6). (iii) Results with 95% confidence intervals, a burst-response study, five sensitivity analyses and a cost break-even analysis (Section 7). (iv) Revised, evidence-qualified hosting guidance (Section 8).

Scope and honesty note. All latency and cost numbers in this paper are produced by simulation and an analytical model under stated assumptions. They are not measurements of a live deployment. Section 8.3 lists what that does and does not support.

2. BACKGROUND AND RELATED WORK
MCP and tool use. MCP defines a JSON-RPC based protocol in which servers expose tools with JSON Schema input definitions, and clients (agents) list and call them over a standard transport [3]. Surveys of the MCP ecosystem identify the tool server as a distinct security boundary, with threats such as tool poisoning and over-privileged servers [7]; Section 9 returns to this.

Serverless platform behaviour. Production traces show that most serverless functions are invoked rarely and that cold starts are a first-order cost for them [8]. Black-box measurements of commercial platforms quantify instance lifetime, idle reclamation and scaling behaviour [9], and the Berkeley view identifies cold start and limited per-function concurrency control as recurring weaknesses [10]. Our simulator adopts the structural ingredients of that literature (boot delay, idle scale-in, bounded scale-out rate) rather than any one vendor's undisclosed internals.

Tail latency. When a request fans out or chains, the probability that at least one component is slow grows quickly, so rare per-call delays dominate the user-visible latency [11]. Agent workflows are chains, which is why we report P99 and workflow-level violation probabilities, not just means.

Gap. Vendor documentation describes plans and limits [12–14] but does not quantify how those mechanisms trade tail latency against cost for MCP-style traffic. That quantification is the aim here.

3. AZURE FUNCTIONS AS AN MCP TOOL HOST

3.1 Requirements and Capabilities

Table 1 maps MCP tool requirements to platform capabilities. Two claims in common guidance need correcting: concurrency is bounded (per-instance concurrency limits and a plan-specific instance cap, never "unlimited" [12, 13]), and schema enforcement is something the tool code or middleware must do; the platform only binds JSON.

Table 1. MCP requirements versus Azure Functions capabilities

Requirement	Capability and caveat
Stateless execution	Each invocation is independent; keep tools session-less even if the transport permits sessions [3]
JSON in/out	HTTP trigger with JSON binding; or the MCP extension tool trigger [6]
Concurrency	Bounded by per-instance concurrency and the plan's instance cap [12, 13]
Schema checks	Not automatic: validate in code or middleware
Versioning	Slots on Premium/Dedicated; use blue-green with separate apps on Flex (no slots at the time of writing) [13]
Authentication	Entra ID tokens, Managed Identity, APIM `validate-jwt` [15]
Tracing	Application Insights, W3C Trace Context [16, 17]

3.2 Hosting Plans

Table 2 summarises the plans evaluated. Limits are as documented at the time of writing and change; they should be re-checked against current documentation before any capacity decision [12–14].

Table 2. Hosting plans compared (documented behaviour)

Plan	Scaling and start-up	Billing basis
Consumption (legacy)	Scale to zero; event-driven scale-out; up to 200 instances on Windows; cold start on new instances [12]	Per execution plus GB-s while executing [18]
Flex Consumption	Scale to zero; per-function scaling; configurable instance cap up to 1,000; optional always-ready instances; HTTP concurrency per instance (default 16) [13]	Per execution plus GB-s on-demand; always-ready billed for baseline and execution [18]
Premium (EP1–EP3)	At least one always-ready instance, pre-warmed buffer instances, instance cap by configuration; no scale to zero by default [14]	Per vCPU-s and GB-s of allocated instances, including idle [18]
Container Apps + KEDA	Scale on queue depth and other scalers; scale to zero optional [19]	Container Apps consumption/dedicated pricing

3.3 Trigger Choice

Tools can be exposed as plain HTTP-triggered functions (used in Section 5 for clarity) or through the Functions MCP extension, whose tool trigger became generally available in late 2025 and handles the MCP transport and tool metadata for the developer [6]. For clients that speak the MCP protocol directly, the extension is the better default; the scaling and cost results below depend on the hosting plan, not on the trigger type.

4. ARCHITECTURE

4.1 Five-Layer Stack

The reference architecture (Fig 1) has five layers. (1) The **agent orchestrator** (for example AutoGen, LangGraph, Semantic Kernel or a hosted agent service [20]) runs the reasoning loop and emits tool calls. (2) An **API gateway** (API Management [15]) centralises OAuth 2.0 validation, rate limiting and routing; it is optional but removes duplicated security logic from every tool. (3) **Function app hosts** run related tools as individual functions. (4) A **backend integration layer** reaches SQL, Cosmos DB, search, storage and messaging. (5) An **observability plane** collects traces, metrics and logs.

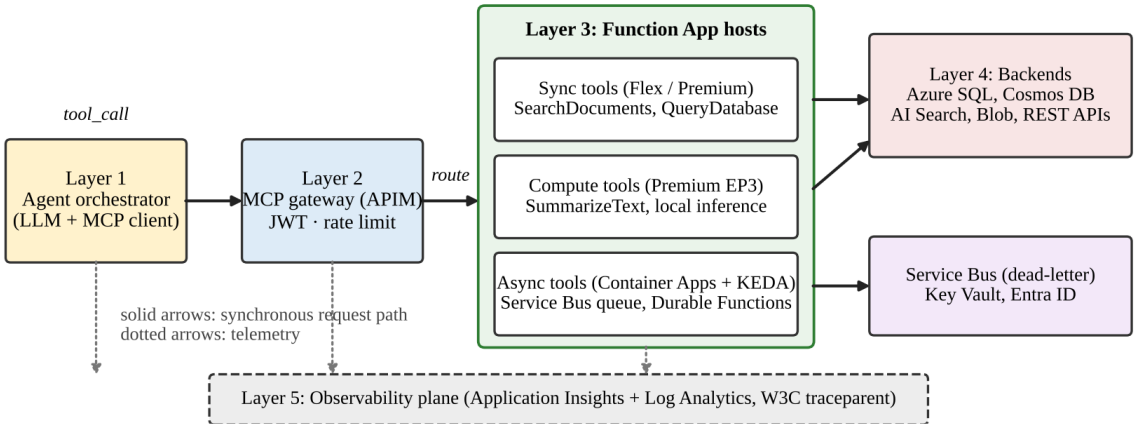

Fig 1: Reference architecture for MCP tools on Azure Functions. Solid arrows: synchronous request path; dotted arrows: telemetry.

4.2 Tool Discovery and Request Flow

Discovery follows MCP's `tools/list` semantics [3]. For a stable inventory, the gateway can serve a static catalogue from configuration. For per-tenant or feature-flagged tool sets, a dedicated discovery function returns only the tools the caller is entitled to use. A synchronous call then proceeds: the agent sends the tool name and arguments; the gateway validates the

token and applies rate limits; the Functions host binds the body to a typed Java model; the tool validates input, calls its backend and returns a JSON result; telemetry is linked to the agent's trace through W3C Trace Context headers [16, 21].

One function, one tool. Multi-command functions with an `action` parameter defeat per-tool scaling, per-tool alerting and per-tool authorisation. Single-purpose functions keep these independent, at the price of more deployable units per app.

5. IMPLEMENTATION (JAVA 21)

Each tool is one class in a standard Maven layout (`tools/`, `middleware/`, `models/`, `registry/`, `config/`), with business logic in injected services so that it can be unit-tested without the Functions runtime. Listing 1 shows the shape of a tool function, using Java 21 records for the typed input.

Listing 1. Tool function (condensed).

```
@FunctionName("SearchDocuments")
public HttpResponseMessage run(
  @HttpTrigger(name = "req",
    methods = {HttpMethod.POST},
    route = "tools/search_documents",
    authLevel = AuthorizationLevel.ANONYMOUS)
  HttpRequestMessage<Optional<String>> req,
  ExecutionContext ctx) {
  var in = McpJson.parse(req.getBody(),
                         SearchInput.class);
  var errs = validator.validate(in);
  if (!errs.isEmpty())
    return McpResponse.error(req, 400, errs);
  var hits = searchService.search(
                in.query(), in.topK());
  return McpResponse.ok(req, hits);
}

record SearchInput(String query, int topK,
                   String filter) {}
```

`ANONYMOUS` authorisation is acceptable only when the gateway is the sole ingress (private endpoint plus token validation upstream); otherwise use function-level keys as a secondary check. The tool's JSON Schema (type, bounds such as `top_k` in [1, 50], required fields) is registered with the discovery layer, and the same bounds are enforced in `validator`.

Concurrency configuration. On Premium and Dedicated plans the per-instance request cap is set in `host.json` (`extensions.http.maxConcurrentRequests`); on Flex Consumption per-instance HTTP concurrency is a platform setting with a documented default of 16 [13]. Section 7.3 shows this single parameter is among the strongest cost levers, so it should be set deliberately and not left at a default. Heavy or long-running tools belong in Durable Functions orchestrations [22], which return a correlation identifier instead of holding an agent call open, or behind a queue with KEDA-driven consumers [19]. Downstream calls should sit behind circuit breakers and bulkheads [23] so that scale-out cannot overwhelm a database.

6. EVALUATION METHODOLOGY

6.1 Approach and Rationale

No public dataset describes agent-to-tool traffic against Azure Functions, and plan internals (placement, allocation policy) are not public. We therefore built a discrete-event simulator of a function app on each plan and an analytical cost model, and we report sensitivity to every parameter we could not take from documentation. The value of the results is in the relative behaviour and the sensitivities, not in the absolute numbers.

6.2 Workload

Agent traffic is modelled as a three-state Markov-modulated Poisson process [24] with states *idle* (0.05 req/s), *work* (10 req/s) and *burst* (300 req/s), mean dwell times of 1800, 600 and 60 s, and transitions idle→work, work→burst ($p = 0.4$) or work→idle ($p = 0.6$), burst→work. Each run covers 6 h (about 209,000 requests; mean rate 9.7 req/s) and every configuration is run with 20 independent seeds. A separate step workload (5 req/s, jumping to 400 req/s for 120 s) isolates burst response.

6.3 Service and Platform Model

Service time for a request executing alongside a concurrent requests on one instance is

$$S = X \cdot (1 + \kappa(a - 1)) + \delta \quad (1)$$

where X is lognormal with median 0.25 s and shape 0.5, $\kappa = 0.03$ models contention and $\delta = 0.03$ s covers gateway and network. A request is dispatched to a ready instance with spare concurrency; otherwise it waits in a FIFO queue while the platform starts instances. Table 3 lists the platform parameters. Instance start-up times are lognormal (shape 0.5). Premium is modelled with one always-ready instance plus one pre-warmed buffer instance that is replenished 30 s after use; Flex with 0 or 2 always-ready instances. Idle instances are reclaimed after 600 s except always-ready ones.

Table 3. Simulation parameters (assumptions, not measurements)

Parameter	Cons.	Flex	Premium
Instance cap	200	1000	100
Concurrency per instance	16	16	16
Median start-up (s)	2.0	1.5	2.0
Min. gap between new instances (s)	1.0	0.1	0.1
Always-ready / floor	0	0 or 2	1 or 3
Idle timeout (s)	600	600	n/a

The start-up medians and especially the scale-out gap are assumptions: public documentation states that Flex scales faster than Consumption but gives no figure for allocation rate [5, 13]. We equalise Flex and Premium at 0.1 s and treat the gap as a sensitivity variable (Section 7.3).

6.4 Cost Model

For N calls per month at mean rate $\lambda = N/(730 \cdot 3600)$ and mean duration $d = 0.25$ s, Little's law [25] gives the mean number of requests in service λd. A busy instance cannot be packed densely at low volume, so the effective per-instance concurrency is

$$c_{eff} = \min(c_{max}, \max(1, r\lambda d)) \quad (2)$$

with a peak-to-mean ratio $r = 10$ and c_{max} the configured concurrency. The on-demand Flex cost is then

$$C_F = (N - N_0) \cdot p_e + (N \cdot d \cdot m / c_{eff} - G_0) \cdot p_g \quad (3)$$

with $m = 2$ GB instance memory, free grants N_0, G_0 (floored at zero) and execution and GB-second prices p_e, p_g. Consumption uses 0.5 GB billed over execution time. Always-ready Flex adds a baseline charge per GB-s. Premium is sized for a target utilisation $\rho = 0.5$:

$$n = \max(1, \lambda d / (c_{max} \cdot \rho)), \quad C_P = n \cdot P_{inst} \quad (4)$$

where P_{inst} ≈ USD 300 per month for an always-on EP2 instance. Prices are list-price approximations (US region) taken from the pricing page [18] at the time of the study; they vary by region and over time, so only the structure and the break-even behaviour should be relied upon. The simulator-derived costs in Section 7.1 apply the same prices to simulated busy and allocated instance-seconds and ignore free grants.

6.5 Metrics

We report P50, P95, P99 and P99.9 end-to-end latency (queueing plus start-up wait plus service), the fraction of requests delayed by scale-out, the fraction exceeding a 1 s service-level objective (SLO), peak instance count, and compute cost per million calls. Intervals are 95% Student-*t* intervals over 20 seeds. Assuming independent calls, a workflow of k sequential calls violates the 1 s bound with probability

$$P_w(k) = 1 - (1 - q)^k \quad (5)$$

where q is the per-call violation probability [11]. Independence is optimistic when bursts correlate delays, so this is a lower bound on workflow-level risk under correlated load.

7. RESULTS

7.1 Baseline Comparison

Table 4. Baseline results (MMPP workload, 6 h, 20 seeds; P99 shown with 95% CI)

Plan	P50 (s)	P95 (s)	P99 (s)	P99.9 (s)	Delayed by scale-out (%)	Peak inst.	Cost per M calls (USD)
Consumption	0.33	2.52	4.72 ± 0.14	5.35	8.1	18.9	3.06
Flex	0.31	0.71	1.26 ± 0.08	1.85	1.8	31.4	7.03
Flex + 2 always-ready	0.31	0.70	1.14 ± 0.03	1.66	1.6	28.4	7.88
Premium EP2 (min 1)	0.31	0.72	1.32 ± 0.07	1.97	2.0	34.8	88.61
Premium EP2 (min 3)	0.31	0.70	1.08 ± 0.02	1.56	1.8	27.1	93.91

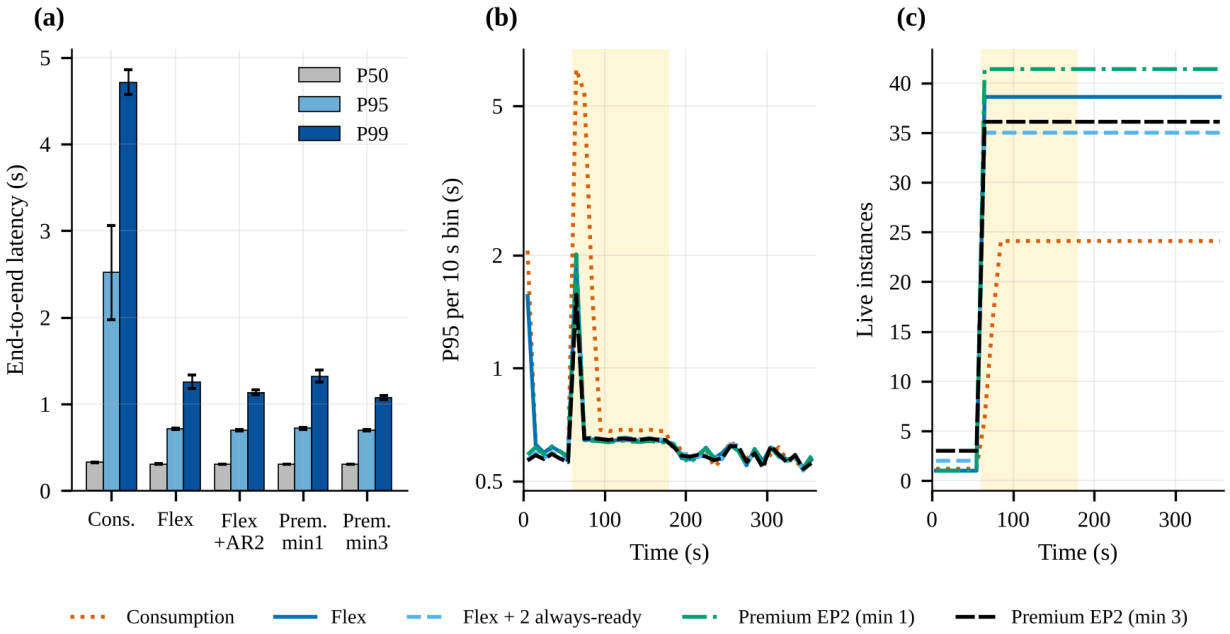

Fig 2: Latency and burst response. (a) Latency percentiles per plan (error bars: 95% CI). (b) P95 per 10 s bin when load steps from 5 to 400 req/s at t = 60 s. (c) Live instances during the same step.

Table 4 and Fig 2a show three results. First, the legacy Consumption plan is clearly worst: P99 of 4.72 s versus 1.26 s on Flex, with 8.1% of requests delayed by scale-out. Second, and contrary to the common advice, **Premium with one always-ready instance is not better than Flex without any always-ready instance**: P99 is 1.32 ± 0.07 s against 1.26 ± 0.08 s (overlapping intervals), because both are limited by how fast new instances can be added once a burst exceeds the warm capacity, not by the first instance. Third, the cost gap is large: Premium costs about 12.6 times more per million calls than on-demand Flex at this load (USD 88.61 against 7.03), because an always-on EP2 instance is billed whether or not it is busy.

Adding always-ready capacity helps modestly: two always-ready Flex instances reduce P99 from 1.26 s to 1.14 s for 12% more cost, and a three-instance Premium floor reaches 1.08 s. Neither removes the tail, since the SLO violation rate stays at 1.4–2.0% for all non-Consumption configurations.

7.2 Burst Response

Fig 2b,c shows the step from 5 to 400 req/s. In the 20 s after the step, the worst 10 s-bin P95 is 6.27 s on Consumption, 1.92 s on Flex, 1.64 s on Flex with two always-ready instances, 2.01 s on Premium (min 1) and 1.57 s on Premium (min 3). Every plan breaches a 1 s objective during onset. The Flex and Premium variants return to a P95 near 0.65 s within 20 s, whereas Consumption needs about 60 s more (bin P95 still up to 1.5 s between 80 and 120 s). Pre-warmed or always-ready capacity shortens and softens the transient but does not remove it; Premium's "no cold starts" claim holds only for load that fits within the warm instances.

7.3 Sensitivity Analysis

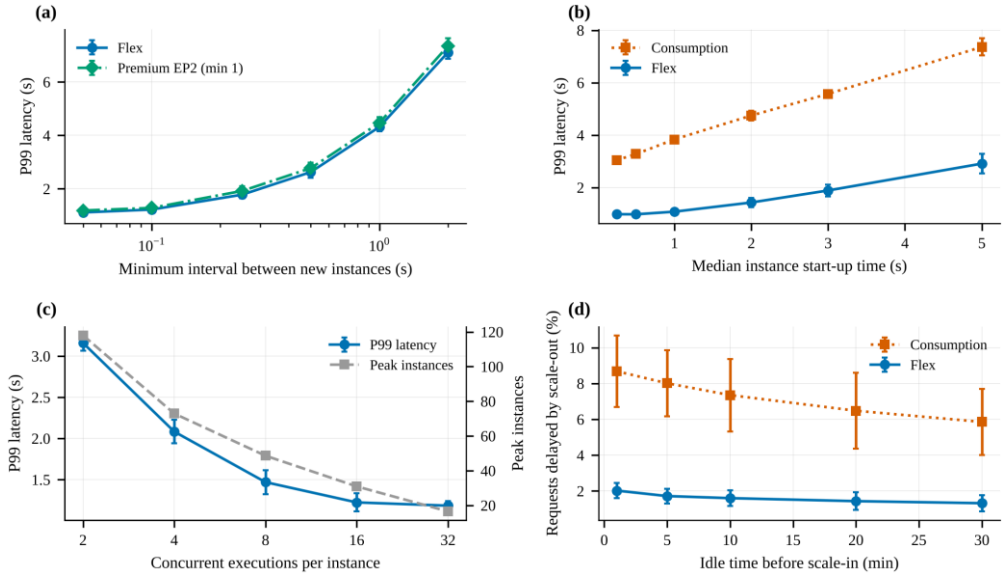

Fig 3: Sensitivity of P99 latency and cost. (a) Minimum interval between new instances. (b) Median instance start-up time. (c) Per-instance concurrency (left axis P99, right axis peak instances). (d) Idle time before scale-in.

Scale-out rate dominates (Fig 3a). Raising the minimum gap between new instances from 0.05 s to 2 s increases Flex P99 from 1.12 s to 7.11 s and Premium P99 from 1.19 s to 7.35 s, while compute cost falls by 15% (Flex) to 25% (Premium) as scaling slows. In this model the allocation rate matters more than the plan label, and it is exactly the parameter that vendors do not publish. Anyone depending on tail latency should measure it on their own deployment.

Start-up time matters, but sub-linearly (Fig 3b). Flex P99 rises from 0.98 s at a 0.25 s median start-up to 1.43 s at 2 s and 2.91 s at 5 s; on Consumption it rises from 3.05 s to 7.36 s. For Java, where JVM start-up and class loading are significant, this is the parameter most likely to differ from our default.

Concurrency drives cost (Fig 3c). On Flex, going from 2 to 16 concurrent executions per instance cuts cost per million calls from USD 16.08 to 7.36 and P99 from 3.16 s to 1.22 s; going from 16 to 32 saves another 13% of cost for almost no latency change in this model. The same change on Premium reduces cost from 240 to 87 USD per million. The caveat is that tools with heavy CPU or memory use per call cannot safely run at high concurrency; our contention coefficient is a simple linear approximation.

Idle timeout is a weak lever (Fig 3d). Extending the idle timeout from 1 to 30 min reduces delayed requests on Consumption from 8.6% to 6.0% and Flex P99 from 1.29 s to 1.13 s, but raises Flex cost from 6.58 to 8.22 USD per million (+25%). Always-ready instances give a better return: on Flex, 1, 2 and 4 always-ready instances give P99 of 1.23, 1.12 and 1.00 s at 7.48, 8.06 and 9.28 USD per million. The Premium floor saturates: raising it from 5 to 8 instances brings no further P99 gain (0.98 s and 1.01 s) while cost rises from 98.59 to 109.86 USD per million.

7.4 Cost Analysis

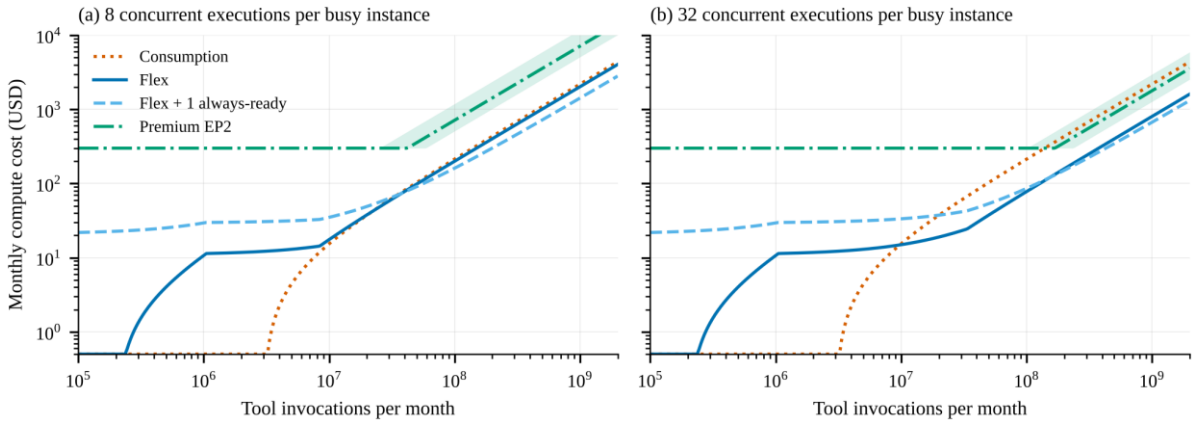

Fig 4: Analytical monthly compute cost versus call volume with (a) 8 and (b) 32 concurrent executions per busy instance. Shaded band: Premium at utilisation targets 0.3–0.7.

Table 5. Monthly compute cost (USD) from the analytical model

Calls per month, c_{max}	Cons.	Flex	Flex + AR1	Prem. EP2
10^7, 8	15	18	35	300
10^8, 8	213	200	161	714
10^9, 8	2,193	2,022	1,421	7,135
10^8, 32	213	78	86	300

Calls per month, c_{max}	Cons.	Flex	Flex + AR1	Prem. EP2
10^9, 32	2,193	804	671	1,784

Fig 4 and Table 5 give the monthly cost at three volumes. Below about 10^7 calls per month (roughly 4 calls/s on average) the fixed cost of an always-on Premium instance dominates and Premium is 17–20 times more expensive than Flex or Consumption. At 10^8 calls with 8 concurrent executions per instance, Premium costs USD 714 against 200 for Flex. In the range 10^5–$10^{10.5}$ calls per month, Premium undercuts Consumption only when instances can be driven at 32 or more concurrent executions (above about 1.4×10^8 calls per month) and never undercuts on-demand Flex or Flex with always-ready capacity for any concurrency up to 64 in this model. Flex with one always-ready instance is cheaper than on-demand Flex from roughly 5×10^7 calls per month at c_{max} = 8, because the always-ready execution rate per GB-second is lower than the on-demand rate, but it is about twice as expensive at low volume.

7.5 Cost–Latency Trade-off and Workflow Impact

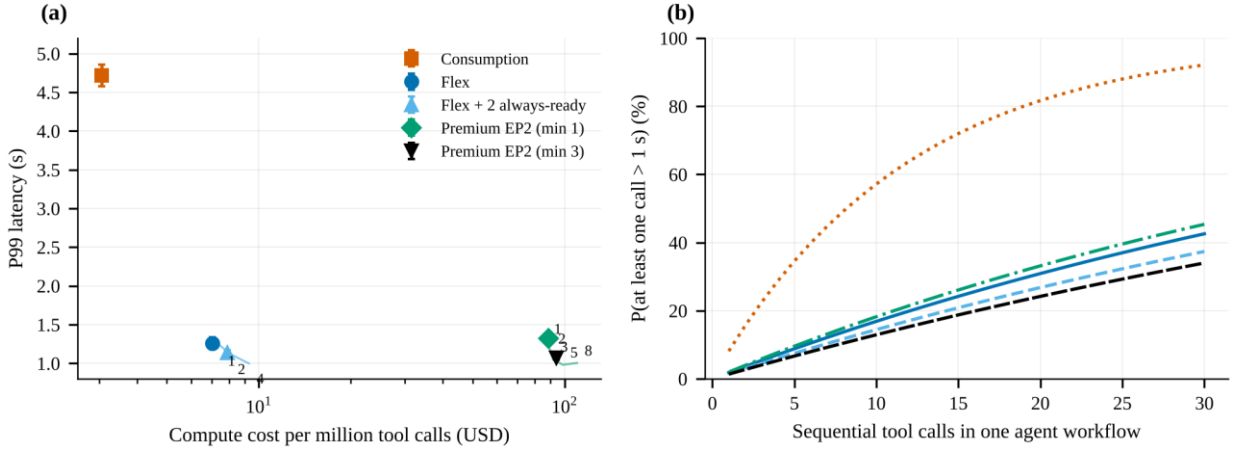

Fig 5: Trade-off and compounding. (a) P99 latency versus compute cost per million calls (log axis); numbered trails show increasing always-ready floor. (b) Probability that a workflow of k sequential calls contains at least one call slower than 1 s.

Fig 5a places the configurations on a cost–latency plane. The Pareto front is formed by Consumption (cheapest, slowest) and the Flex variants; Premium points lie about an order of magnitude to the right at no better latency. Fig 5b applies Eq. (5) to the measured per-call violation rates. For a 10-step workflow the probability that at least one call exceeds 1 s is 57% on Consumption, 17% on Flex, 14% on Flex with two always-ready instances, 18% on Premium (min 1) and 13% on Premium (min 3); at 30 steps the figures rise to 92%, 43%, 37%, 45% and 34%. Per-call tail latency that looks acceptable (P99 near 1 s) is therefore not acceptable for long agent chains, which argues for fewer sequential tool calls, parallel calls where the agent can do so, and client-side timeouts and retries with jitter.

8. DISCUSSION

8.1 Revised Hosting Guidance

The results change the guidance of the earlier version of this paper. The original recommendation (Premium for any synchronous tool) is not supported on cost or latency grounds for the workload studied. Table 6 gives the evidence-qualified alternative.

Table 6. Hosting guidance supported by the results

Situation	Suggested plan	Evidence / caveat
Async, batch or latency-tolerant tools	Flex (or Consumption if cost-critical)	Cheapest at low volume; P99 4.7 s on Consumption
Interactive tools, bursty traffic	Flex + 1–2 always-ready	P99 1.1–1.2 s at 7–8 USD per million; Premium no better
Sustained high volume, dense packing	Flex + always-ready; Premium only if features require it	Premium cheaper only above c≈32 vs Consumption
Needs Premium-only features	Premium EP2/EP3	Not modelled: e.g. slots, larger instance sizes, execution limits [14]
Queue-driven async tools	Container Apps + KEDA or Durable	Scale tied to backlog [19, 22]

8.2 Practical Implications

Three practices follow. (1) Treat per-instance concurrency as a first-class design parameter, tested against the tool's real CPU and memory profile. (2) Prefer a small always-ready floor on an on-demand plan to a whole Premium plan, then add floor capacity only until the P99 curve flattens (here at 2–4 instances). (3) Measure the platform's instance-allocation rate and start-up time for your runtime and region, since those two parameters dominate the tail in our sensitivity study.

8.3 Threats to Validity and Limits

Simulation, not measurement. The simulator reproduces structural mechanisms but not Azure's actual scale controller, placement or networking. The scale-out gap, start-up medians and contention coefficient are assumptions; their effect is bounded by the sensitivity study but their true values are unknown. **Workload.** The MMPP is a stylised stand-in for agent traffic; real agent traffic will have correlated retries, heavy-tailed tool durations and diurnal patterns. **Scope.** We model compute only. Backend latency, gateway cost, data egress, networking, storage and observability ingestion are excluded, and in many deployments exceed function compute cost. **Independence.** Eq. (5) assumes independent calls and understates correlated risk. **Prices.** List prices change and exclude discounts and committed-use agreements. **Plan coverage.** Dedicated (App Service) plans were not modelled. The conclusions are therefore hypotheses that a deployed benchmark should confirm; a replication on live Flex and

Premium apps with a load generator replaying the MMPP trace is the obvious next step.

9. SECURITY AND GOVERNANCE

Tool servers sit at a trust boundary between a model that may be manipulated and systems that hold real data, a risk highlighted in MCP threat analyses [7]. Controls should be layered (Table 7): token validation at the gateway, per-tool role checks inside the app, and network isolation so that the function app is not reachable except through the gateway. Treat all agent-provided arguments as untrusted input: validate against the JSON Schema, use parameterised queries, bound input sizes, and log every invocation (with sensitive fields redacted) for audit.

Table 7. Layered security controls for MCP tool hosts

Layer	Control
Identity	Entra ID OAuth 2.0 tokens scoped to the tool API; gateway `validate-jwt` [15]; app roles per tool
Workload identity	Managed Identity to reach Key Vault and data stores; no secrets in code
Network	VNet integration for outbound, private endpoints for inbound, private DNS for backends
Secrets	Key Vault references in app settings, resolved at runtime
Input	Strict schema validation, parameterised queries, size limits, audit log
Resilience	Rate limits at the gateway, circuit breakers on downstream calls [23]

10. OBSERVABILITY

Metrics (invocation count, failure rate, duration, instance count), structured logs (tool name, input summary, duration, outcome) and distributed traces are all needed to diagnose agent failures. W3C Trace Context headers [16, 21] propagated by Application Insights and OpenTelemetry [17] link each tool call to its parent agent step. The simulation results suggest alert thresholds (Table 8): in particular, a cold-start or delayed-request rate above 5% should first prompt a review of the always-ready floor and concurrency setting, not an automatic plan upgrade.

Table 8. Suggested production alert thresholds

Metric	Threshold and action
Duration P99	> 2 s: investigate downstream dependencies
Failure rate	> 1% over 5 min: start incident response
Instance count	At plan cap: raise cap or add regions
Delayed-request rate	> 5%: review always-ready floor, concurrency, start-up time
Queue depth (async)	> 1,000 messages: scale-out is lagging
Token failures	Any spike: possible credential misuse

11. CONCLUSION

Azure Functions is a workable host for MCP tools because tools are short and stateless, but the simple rule “Premium removes cold starts” does not survive quantitative testing. In a simulated bursty workload, Flex Consumption with a small always-ready floor matched or beat a Premium plan on P99 latency (about 1.1 s versus 1.3 s) at roughly one-tenth of the compute cost, while the legacy Consumption plan was cheapest and slowest (P99 4.7 s). Instance allocation rate and per-instance concurrency, not plan label, dominated tail latency and cost respectively, and per-call P99 near 1 s compounds into a 13–18% chance of a slow call in a 10-step workflow. These findings come from a model whose key parameters are assumptions; validating them on a live deployment, including Java cold-start behaviour and real agent traces, is the main piece of remaining work. The simulation code and raw results are available from the author on request.

12. REFERENCES

[1] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, “ReAct: Synergizing Reasoning and Acting in Language Models,” in Proc. ICLR, 2023.

[2] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, “Toolformer: Language Models Can Teach Themselves to Use Tools,” in Proc. NeurIPS, 2023.

[3] Anthropic, “Model Context Protocol Specification,” Anthropic PBC, San Francisco, CA, 2025. [Online]. Available: https://modelcontextprotocol.io

[4] Microsoft, “Azure Functions Overview,” Microsoft Azure Documentation, Redmond, WA, 2025. [Online]. Available: https://learn.microsoft.com/azure/azure-functions

[5] Microsoft, “Event-driven Scaling in Azure Functions,” Microsoft Azure Documentation, Redmond, WA, 2025. [Online]. Available: https://learn.microsoft.com/azure/azure-functions/event-driven-scaling

[6] Microsoft, “Azure Functions — Ignite 2025 Update: MCP Tool Trigger General Availability,” Microsoft Tech Community, Nov. 2025. [Online]. Available: https://techcommunity.microsoft.com/blog/appsonazureblog/azure-functions-ignite-2025-update/4469815

[7] X. Hou, Y. Zhao, S. Wang, and H. Wang, “Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions,” arXiv preprint arXiv:2503.23278, 2025.

[8] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, “Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider,” in Proc. USENIX Annual Technical Conf. (ATC), 2020, pp. 205–218.

[9] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking Behind the Curtains of Serverless Platforms,” in Proc. USENIX Annual Technical Conf. (ATC), 2018, pp. 133–146.

[10] E. Jonas et al., “Cloud Programming Simplified: A Berkeley View on Serverless Computing,” arXiv preprint arXiv:1902.03383, 2019.

[11] J. Dean and L. A. Barroso, “The Tail at Scale,” Commun. ACM, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[12] Microsoft, “Azure Functions Scale and Hosting,” Microsoft Azure Documentation, Redmond, WA, 2025. [Online]. Available: https://learn.microsoft.com/azure/azure-functions/functions-scale

[13] Microsoft, “Azure Functions Flex Consumption Plan Hosting,” Microsoft Azure Documentation, Redmond, WA, 2025. [Online]. Available: https://learn.microsoft.com/azure/azure-functions/flex-consumption-plan

[14] Microsoft, “Azure Functions Premium Plan,” Microsoft Azure Documentation, Redmond, WA, 2025. [Online]. Available: https://learn.microsoft.com/azure/azure-functions/functions-premium-plan

[15] Microsoft, "Azure API Management Documentation," Microsoft Azure Documentation, Redmond, WA, 2025. [Online]. Available: https://learn.microsoft.com/azure/api-management

[16] W3C, "Trace Context," World Wide Web Consortium Recommendation, Feb. 2020. [Online]. Available: https://www.w3.org/TR/trace-context

[17] OpenTelemetry Authors, "OpenTelemetry Specification," Cloud Native Computing Foundation, 2025. [Online]. Available: https://opentelemetry.io/docs/specs/otel

[18] Microsoft, "Azure Functions Pricing," Microsoft Azure, Redmond, WA, 2025. [Online]. Available: https://azure.microsoft.com/pricing/details/functions

[19] KEDA Authors, "KEDA: Kubernetes Event-Driven Autoscaling," Cloud Native Computing Foundation, 2025. [Online]. Available: https://keda.sh

[20] Microsoft, "Azure AI Foundry," Microsoft Azure, Redmond, WA, 2025. [Online]. Available: https://ai.azure.com

[21] W3C, "Trace Context Level 2," World Wide Web Consortium Working Draft, 2024. [Online]. Available: https://www.w3.org/TR/trace-context-2

[22] Microsoft, "Durable Functions Overview," Microsoft Azure Documentation, Redmond, WA, 2025. [Online]. Available: https://learn.microsoft.com/azure/azure-functions/durable/durable-functions-overview

[23] Resilience4j Authors, "Resilience4j: Fault Tolerance Library for Java," GitHub, 2025. [Online]. Available: https://resilience4j.readme.io

[24] W. Fischer and K. Meier-Hellstern, "The Markov-Modulated Poisson Process (MMPP) Cookbook," Performance Evaluation, vol. 18, no. 2, pp. 149–171, 1993.

[25] J. D. C. Little, "A Proof for the Queuing Formula: L = λW," Operations Research, vol. 9, no. 3, pp. 383–387, 1961.