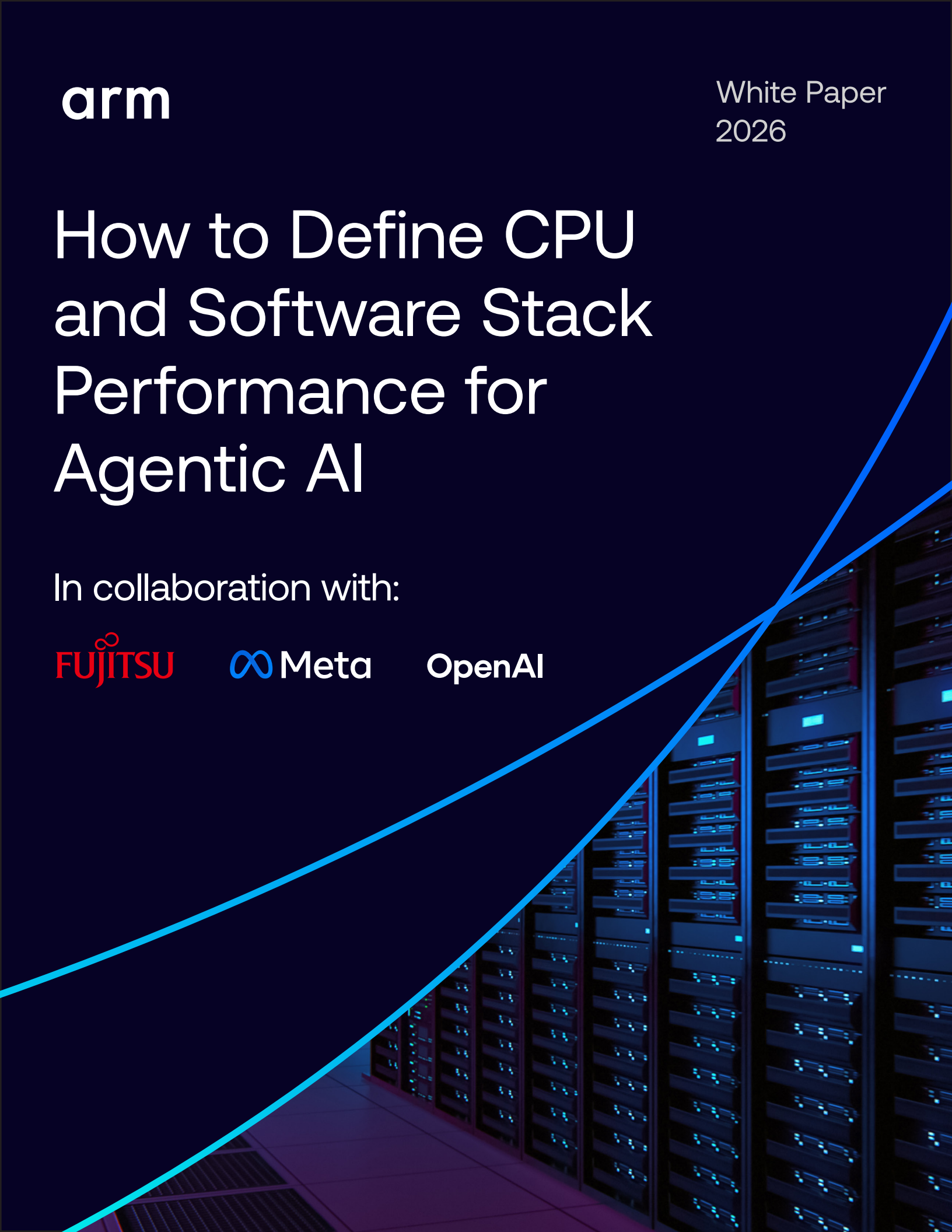
arm

White Paper
2026

How to Define CPU and Software Stack Performance for Agentic AI

In collaboration with:

FUJITSU Meta OpenAI



Executive Summary

Agentic AI changes the CPU performance problem. A completed task can involve repeated model calls, retrieval, state management, tool execution, application interactions, policy checks and validation loops. Accelerators remain central to model execution, while CPUs coordinate and execute much of the surrounding system path from user intent to completed work.

This paper proposes a use-case-driven framework for evaluating that CPU contribution. It separates CPU-visible work into two complementary areas: common agentic framework functions and use-case-specific tool workloads. Five representative workload archetypes are used to capture distinct deployment venues, tool classes and CPU stress profiles. The paper then describes two complementary measurement approaches: component benchmarking, which emphasizes reproducibility and attribution and end-to-end benchmarking, which emphasizes realistic task-level outcomes.

The objective is not to prescribe a finished benchmark suite or a single figure of merit. It is to define the measurement boundaries, workload categories and methodological gaps that need industry agreement. In particular, replayable agent trajectories offer a path to separating CPU work from model and network effects while reducing the run-to-run variability introduced by non-deterministic agent behavior.

"As agentic AI evolves, evaluating CPU performance requires benchmarks that go beyond traditional metrics to reflect real-world workloads. As we develop our next-generation high-performance, energy-efficient Arm-based CPU, FUJITSU-MONAKA, we recognize the importance of transparent, reproducible and representative measurement. Fujitsu welcomes Arm's initiative to foster industry-wide collaboration and help optimize the next generation of AI infrastructure."

Toshio Yoshida, SVP, Head of Advanced Computing Development Unit,
Fujitsu Research, Fujitsu Limited

"As agentic AI moves into production, the industry needs greater visibility into how different workloads exercise the infrastructure beneath them. We support Arm's work to establish a more consistent basis for comparison and believe this can help inform how systems evolve."

Abhishek Dhanotia, Software Engineer, Meta

"AI systems are increasingly moving beyond generating responses to completing complex tasks through reasoning, tool use and interaction with software environments. Understanding the performance of the infrastructure supporting these workflows, including orchestration and tool execution, is an important area for the industry as agents become more capable and widely deployed. Industry collaboration on representative benchmarking will be important for characterizing these emerging workloads and optimizing performance across the complete AI system."

Colin Jarvis, Global Head of Forward Deployed Engineering, OpenAI

1. Why Agentic AI Changes CPU Performance

Agentic AI systems have moved beyond single-turn model interactions toward applications that plan, use tools, retrieve information, act across software environments and maintain task state over time. A user request may therefore trigger a sequence of model calls, tool invocations, retrieval steps, data transformations, application interactions and validation loops before the task is complete.

The model remains a critical component, but the full workload is no longer limited to model inference. The CPU is active across orchestration, routing, state management, retrieval, tool execution, application serving, data transformation, security checks and observability. In deployments with accelerators, the accelerator may execute dense model operations while the CPU coordinates the broader execution path and the software environment around it.

For benchmarking, the CPU-visible workload can be organized into two broad parts. The first is the agentic framework: control flow, planning, context assembly, routing, session management, scheduling, retries, state, guardrails and telemetry. The second is the tool workload: the actual work performed outside the model, including compilers, runtimes, test systems, databases, search and retrieval, file and media processing, browser or shell automation, simulation and enterprise APIs. Both can affect end-to-end latency, throughput, cost and reliability.

Existing benchmark families cover only portions of this execution path. Traditional CPU suites can characterize important compute and service behaviors, while AI benchmarks are generally designed around model execution. End-to-end agent profiling captures the full application but makes CPU attribution difficult because model choice, network services and agent behavior can dominate wall-clock results. A practical framework therefore needs both representative CPU components and realistic task-level execution.

Use case is the organizing principle because it determines the tools an agent invokes, where execution occurs and what constrains the loop. The same foundation model can sit inside very different systems: a coding agent that compiles and tests software, an enterprise agent that processes transactions, a research agent that invokes simulation tools, a personal assistant that interacts with documents and applications or a robotics agent that coordinates perception and actuation. Those differences create materially different CPU requirements.

This framing also clarifies what a CPU benchmark should and should not claim. A component result can show how efficiently a processor executes a defined function, but it does not by itself establish how quickly an agent completes a goal. Conversely, a task-level result may capture user-visible performance while obscuring which part of the software stack or platform caused the difference. The framework therefore treats attribution and task realism as separate requirements that must eventually be connected.

The practical sequence is to begin with representative use cases, identify the CPU-visible framework and tool activity within each one, map that work to mature benchmarks where possible and use trace-based methods where existing proxies are inadequate. This creates a common language for discussing agentic CPU behavior across software, cloud, silicon and system communities without requiring every deployment to share the same architecture.

2. Five Representative Workload Archetypes

The framework uses five representative archetypes to avoid treating agents as a single generic workload class. The objective is not to define an exhaustive market taxonomy, but to span distinct combinations of deployment venue, dominant tool activity and CPU stress. Each archetype therefore contributes a workload signature that is not adequately represented by the others.

Use case	Deployment venue	Dominant tool class	Distinct CPU signature	Real world applications
Software engineering	Cloud developer infrastructure	Code editing, build, test execution	Compilation and sandboxed test execution at high churn	• Claude Code • OpenAI Codex • Github Copilot
Enterprise operations	Enterprise data center / SaaS	APIs, records, transactions	Latency-bounded transactional serving at high agent concurrency	• Salesforce Agentforce • ServiceNow AI agents • Sierra Agent Studio
Financial & scientific research	Secure cloud / HPC-adjacent facilities	Search, code, simulation, domain tools	Numerical work plus long-running agent loops	• Google DeepResearch • Microsoft Discovery • SpectrumIQ
Personal assistants	Client and edge devices plus cloud	Browsing, documents, media, local models	Interactive latency, parsing, local retrieval and rendering	• Meta Muse • OpenAI Dots • Google Gemini • Microsoft Copilot • Bytedance Dreamina
Robotics	Embedded systems, edge and fleet backends	Sensors, planners, simulation, actuation	Real-time loops, preprocessing and distributed coordination	• Figure Helix

Table 1. Representative agentic AI workload archetypes and CPU stress profiles.

The five archetypes were selected to span different execution locations, dominant tool classes and constraints on the agent loop. From a CPU perspective, those variables influence memory behavior, I/O, concurrency, scheduling, latency targets and the balance between interactive and batch execution. The categories are therefore best understood as representative stress profiles rather than market segments. A production application may combine characteristics from more than one archetype, but the benchmark design can still use the archetypes to identify which CPU behaviors need coverage.

SOFTWARE ENGINEERING

Software-engineering agents repeatedly search repositories, execute language runtimes, resolve dependencies, compile code, run tests, perform static analysis and coordinate repair loops. These activities stress CPU throughput, memory hierarchy, storage and runtime scheduling. At scale, parallel background agents and repeated test execution can make tool work a substantial part of task time and infrastructure cost. Repository-level agent benchmarks and terminal traces can expose the tool sequence needed to select representative CPU workloads.

The execution environment also matters. Coding agents may run in cloud sandboxes, CI systems, developer workstations or secure enterprise build environments. That placement changes the relative cost of storage access, sandbox startup, local versus remote tool calls and background concurrency. A useful benchmark

therefore needs to capture both the repeated tool activity itself and the infrastructure behavior that surrounds it, rather than treating the model call as the complete workload.

ENTERPRISE OPERATIONS

Enterprise agents operate across APIs, databases, workflow engines, identity systems, document stores and policy layers. CPU work includes application serving, transactional access, retrieval, serialization, policy enforcement, audit logging, routing and orchestration. The important performance profile is often high concurrency under latency and governance constraints, so service, database, OLTP, vector-search and compression workloads are useful component proxies while task-level traces preserve workflow realism.

Enterprise operations also illustrate why agentic performance cannot be reduced to raw throughput. Stateful workflows may involve retries, queueing, policy decisions, human approval and audit requirements. A platform can therefore look strong on an isolated database or service test yet still underperform at the workflow level if orchestration, serialization or policy checks become bottlenecks under concurrency. Component and end-to-end measurements are both needed to separate those effects.

FINANCIAL & SCIENTIFIC RESEARCH

Research agents combine evidence retrieval with code execution, numerical analysis, simulation and iterative validation. Much of the workflow around model inference remains general-purpose compute: document processing, database and graph operations, numerical kernels, simulation setup, policy evaluation, data movement and post-processing. Existing scientific, graph, database and simulation benchmarks provide component coverage; the open challenge is representing the full loop of planning, tool execution, evidence management and revision.

These research workloads are typically longer-running and more tool-intensive than interactive assistants. Minutes may be acceptable for some tasks, but provenance, reproducibility, validation and secure access to data become central requirements. That shifts the CPU profile toward sustained data preparation, code and simulation execution, artifact handling and coordination across long-running jobs, with human review often remaining part of the final decision path.

PERSONAL ASSISTANTS

Personal assistants often combine local and cloud execution and place greater weight on responsiveness. CPU work includes application control, document parsing, browser automation, media preprocessing, local retrieval, compression, synchronization and secure execution. Because user-perceived latency is central, computer-use traces and established media, database, retrieval and browser-related workloads provide a useful mix of task realism and component-level isolation.

Personal assistants also show why deployment is not synonymous with workload category. Some context construction, indexing, speech or media preprocessing and smaller-model stages may execute locally, while deeper reasoning occurs in the cloud. Benchmarking must identify which side of that split owns each CPU-visible function. Otherwise, a result can accidentally compare different system boundaries rather than different processor behavior.

ROBOTICS

Robotics agents span local control, edge infrastructure, simulation and fleet backends. CPU-visible work includes fleet orchestration, graph planning, simulation, sensor and video preprocessing, world-state management, retrieval, monitoring and backend services. These systems add strict latency, jitter, reliability and coordination requirements. Network simulation, graph analytics, media processing, databases and scientific simulation can

represent important components, while deployment traces are needed to preserve the timing and coordination characteristics of real systems.

Robotics adds another distinction: part of the workload is embodied and time-sensitive, while another part can run in edge servers, simulation clusters or cloud fleet infrastructure. Local control loops and sensor processing may be constrained by hard latency and jitter requirements, while simulation and fleet services emphasize throughput, concurrency and reliability. The benchmark boundary should therefore distinguish real-time local work from the cloud-side services that train, simulate, coordinate and monitor the system.

3. Defining the Benchmark Boundary

Agentic AI systems range from a single model using a few tools to distributed systems with planning, persistent state, human approval and multiple agents. Reactive, plan-and-execute and multi-agent patterns may all appear within the five workload archetypes. For CPU benchmarking, the important question is not the architectural label itself, but which software components execute, where they execute and which of those components fall inside the measurement boundary.

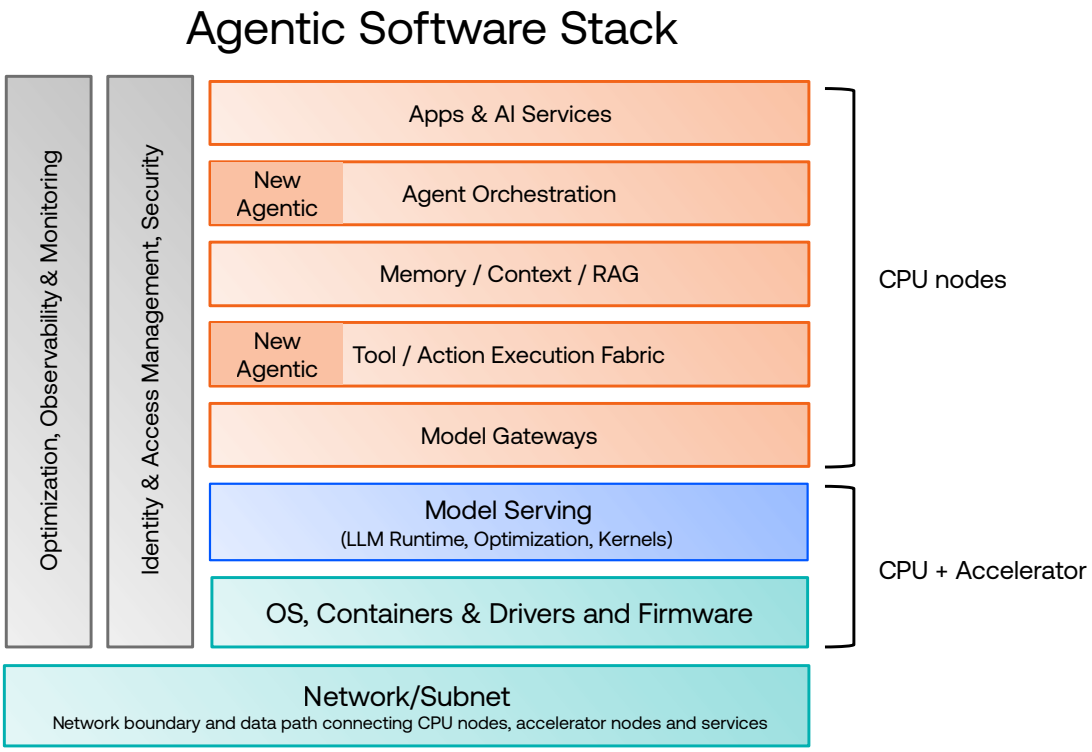


Figure 1. Agentic AI infrastructure components and services.

Figure 1 represents the logical stack as a benchmark-definition framework rather than a prescribed architecture. CPU-relevant layers include application and gateway services; agent orchestration; memory, context, retrieval and state; the tool execution fabric; model gateways; CPU-side model-serving functions; operating systems, containers and drivers; and cross-cutting security and observability. Functions may be combined or distributed across services, so benchmarks should define the function being measured rather than assume a fixed implementation.

The benchmark boundary should include CPU work that contributes directly to those layers. Examples include scheduling and graph traversal in orchestration; indexing, search and filtering in retrieval; schema validation, authentication, dispatch and sandbox lifecycle in tool execution; tokenization, request scheduling, sampling, streaming and KV-cache metadata management in model serving; and cryptographic processing, policy checks, logging and telemetry in the cross-cutting layers. Accelerator-resident neural-network execution should be reported separately when the purpose is to isolate CPU performance.

Within that stack, orchestration coordinates models, tools, memory and agents; memory and retrieval services maintain context, indexes and durable artifacts; state management records workflow progress and enables pause, resume and recovery; and the tool fabric validates, authenticates, dispatches and monitors tool calls. These functions can produce scheduling, serialization, graph-traversal, database, search and isolation work that is directly visible to the CPU.

Model gateways and serving engines add a second class of CPU activity around inference. Routing, prompt adaptation, tokenization, batching, sampling, streaming and cache management may all run on general-purpose cores even when neural-network execution is accelerator-resident. Operating systems, containers and drivers provide the resource and isolation boundaries underneath those services, while identity, policy enforcement, guardrails, audit and observability cut across the stack. These cross-cutting services should be included when they consume CPU inside the selected benchmark boundary and reported separately when they trigger additional model inference.

DEPLOYMENT PLACEMENT DEFINES WHAT IS MEASURED

The agent runtime, tools, data and model-serving stack do not necessarily execute in the same environment. The same application domain may use local tools with managed inference, split execution between edge and cloud or place the complete stack in an operator-controlled cloud. Component placement therefore determines which CPU workloads are measured locally, which are measured in the cloud and which must be represented as remote dependencies.

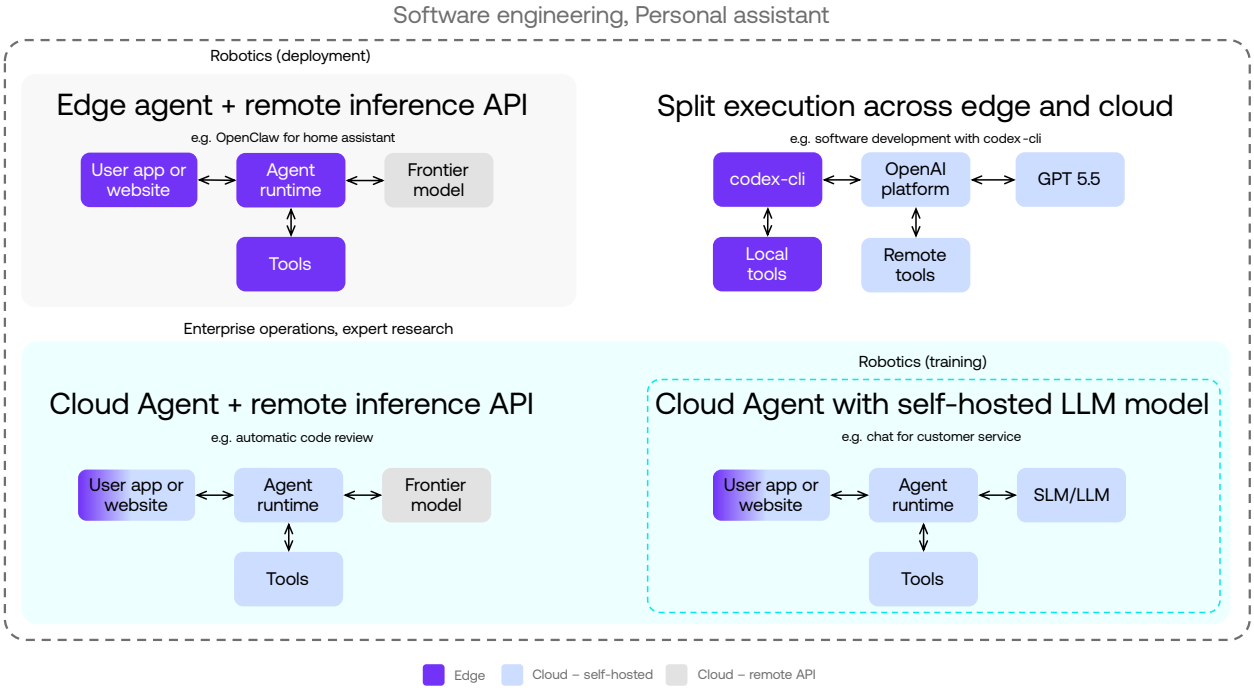


Figure 2. Cloud-centric deployment scenarios for agentic AI systems.

The rule is straightforward: classify the deployment by where individual components execute, not by assigning one deployment model to an entire application category. A coding workflow, for example, may execute repository and shell tools on a developer system while model reasoning occurs through a managed cloud service; a high-volume customer service system may place runtime, retrieval, databases, security, observability and model-serving functions inside the same operator-controlled environment. The benchmark should state component placement, trust and network boundaries and the treatment of managed or remote services.

Placement also changes the performance objective. Interactive assistants and coding agents can emphasize end-to-end and tail latency, time to first action and edge-cloud exchange overhead. Backend workloads can emphasize throughput, concurrency and cost per completed task. Self-hosted model serving exposes additional CPU-side serving work, while managed inference treats that work as an external dependency.

A cloud agent with self-hosted inference demonstrates the broadest boundary. A customer-service agent, for example, can place orchestration, memory, retrieval, databases, policy services, observability and the model-serving engine inside an operator-controlled public or private cloud. In that configuration, CPU-side tokenization, request scheduling, sampling, streaming and cache management are part of the measured system alongside the agent runtime and tools. With managed inference, those same serving functions may sit outside the boundary. The methodology therefore has to report the placement decision explicitly rather than assume that all agent deployments expose the same CPU work.

4. Measuring Agentic CPU Performance

4.1 TWO COMPLEMENTARY METHODS

Two measurement approaches are needed because they answer different questions. Component benchmarking represents agentic AI through the main CPU-visible parts of the execution path. It provides controlled, reproducible measurements, isolates CPU performance from model and network effects and helps attribute platform-level differences to specific functions. End-to-end benchmarking measures complete task execution across the model, agent runtime, tools, data systems and supporting infrastructure. It is closer to user-visible performance but is more sensitive to model choice, framework behavior, deployment topology, network effects and task quality.

Dimension	Component-based benchmarking	End-to-end benchmarking
Primary purpose	Attribute CPU performance to specific framework or tool components.	Measure task-level outcomes across the complete agent execution path.
Strengths	Controlled and reproducible; isolates CPU effects; can use mature benchmarks and proxies.	Application realism; exposes task completion time, throughput, cost and energy per successful task.
Limitations	Component weights/configurations may not match production agent behavior.	LLM non-determinism, waiting time, network services and inference can dominate wall-clock results.
Best use	CPU performance analysis and optimization.	Validation that local improvements translate into better completed-task outcomes.

Table 2. Complementary roles of component-based and end-to-end benchmarking.

4.2 COMPONENT METHODOLOGY AND CPU MEASUREMENT BOUNDARY

The component methodology has two pillars. The first is a set of platform components that recur across many agent architectures and application domains: gateways, orchestration, memory and knowledge retrieval, state, the tool harness, model routing, security, observability and CPU-side model serving. The second is use-case-specific tool work, such as compilation and testing for software engineering or numerical analysis and simulation for research. Together, these two pillars represent both the common agentic software stack and the work that makes individual applications distinct.

– The methodology is intentionally limited to CPU workloads. Inputs normally generated by a model, remote agent or external service can be replayed from fixed traces. CPU-side serving functions such as tokenization, sampling, structured-output processing, scheduling and KV-cache metadata management can be exercised without including accelerator inference throughput. Communication-related CPU work - request parsing, serialization, encryption, compression, protocol handling and telemetry - remains in scope, while the underlying network should not constrain the result.

The initial framework should favor validated, packaged industry benchmarks and representative proxies where they exist. Portable, reproducible and correctly validated benchmarks take substantial engineering effort and premature agent-specific suites can encourage optimization for benchmark artifacts rather than production behavior. The goal at this stage is therefore to identify the workload categories that need coverage and progressively replace proxies with more faithful agentic workloads as those workloads mature and gain acceptance.

There is also a useful distinction between microbenchmarks and component benchmarks. Microbenchmarks isolate specific operations that recur across the stack - for example, serialization, tokenization, cryptographic checks, caching or context-switching overhead. They are efficient for studying individual functions, but asynchronous systems can make a single operation's execution time a poor predictor of total component latency. Component benchmarks evaluate a complete service or software component and therefore capture more of its scheduling, queueing and interaction effects. The two levels are complementary rather than interchangeable.

For tool workloads, use-case traces should guide which established benchmarks are selected and how they are configured. A software-engineering trajectory can reveal the relative importance of compilation, interpreters, static analysis and sandboxed test execution; an enterprise trajectory can expose databases, service layers and policy checks; research and robotics traces can identify numerical or simulation kernels. This trace-to-component mapping is intended to prevent the benchmark suite from becoming a disconnected collection of familiar tests.

4.3 REPRESENTATIVE BENCHMARK COVERAGE

Table 3 shows a mapping of our representative agentic AI use cases to potential sources of agent trajectories. Agent benchmarks are useful not only as end-to-end tests, but as sources of representative execution trajectories. Those trajectories reveal which CPU-visible tools, runtimes and services occur in each use case and therefore guide component-benchmark selection.

Use case	Source of agent trajectories for end-end benchmarking	CPU visible work exposed by these trajectories
Software Engineering	• SWE-bench • multi-SWE-bench • Terminal-bench	Compilation/build, language runtimes, static analysis, shell/filesystem operations, sandbox execution
Enterprise Operations	• tau2-bench • CRMArena • WorkArena	Databases/KV stores, API/RPC processing, application runtimes, retrieval/data processing, policy/business logic
Financial and Scientific Research	• ScienceAgentBench • DiscoveryWorld • FinanceBenchmark	Data/document retrieval, databases, Python/runtime execution, numerical computation, scientific libraries, simulation, sandbox/code execution
Personal Assistant	• GAIA • AndroidWorld • VisualWebArena • AssistantBench	Application/browser interaction, language runtimes, databases, text/media processing, local retrieval
Robotics	• EmbodiedBench • Behavior1K	Numerical solvers, physics simulation, image/video processing, control runtimes, networking/RPC

Table 3: Mapping to sources of agent trajectories for end-end benchmarking

4.4 METRICS AND RACK-LEVEL SCALING

No single metric is sufficient because agentic AI combines latency-sensitive sequential operations with concurrent backend work. Relevant dimensions include latency and throughput; tail latency and variability; energy per operation and performance per watt; cost per operation or cost at a specified throughput; scaling efficiency across threads, cores, sockets and concurrent instances; performance under different utilization or power limits; and cold-start versus steady-state behavior. Reporting rules should reflect the characteristics of each workload rather than force heterogeneous components into a premature common score.

Component results provide workload-specific insight, while server- and rack-level analysis can show how those results translate into deployable capacity under a fixed power envelope. For a clearly defined set of representative workloads, reporting both individual results and an aggregate summary provides a useful view of overall platform behavior, provided the system power, density, scaling assumptions and aggregation method are disclosed. The resulting summary characterizes the tested workload set rather than defining a universal Agentic AI score.

Because different components expose different performance characteristics, workload-level results should remain visible alongside any aggregate summary. An orchestration service, vector-search workload and compiler naturally have different units and scaling behavior. Results should document configuration, utilization, concurrency and power conditions, and clearly state how any aggregate metric is constructed.

Although open-source implementations of these components are generally available, traditional CPU benchmarks do not typically cover them, making it difficult to compare different CPU architectures and microarchitectures. This section reviews the approaches considered to address this gap:

- **Functions to Characterize:** Microbenchmarks provide an efficient way to evaluate specific features or workloads that may be common across multiple components. For example, semantic caching in an AI gateway shares similarities with prompt caching in an inference serving engine. However, with asynchronous scheduling mechanisms employed by many components, the execution time of an individual task is not necessarily correlated with its contribution to the overall component latency. While microbenchmarks are well suited to isolating specific functions, they can be difficult to design certain component features.
- **Component benchmarks:** Component benchmarks complement microbenchmarks by evaluating the performance of an entire software component. Benchmarking a complete component therefore provides a more representative assessment of its overall performance. Benchmarks built in components, already packaged in benchmarks suites or custom-built benchmarks are different options.

To illustrate the approach, Table 4 presents a preliminary list of the types of functions to characterize and components currently under consideration. The specific benchmarks to be selected are still under investigation at this stage. As the evaluation framework continues to evolve, the list will be refined and expanded. Consequently, it is intentionally kept generic.

Stack component	Functions to characterize		Example component benchmarks
Apps	• Python/Java/JS execution • Compilation • Text processing	• Document/media processing • Application-specific logic	• Language-runtime benchmarks • Compiler/build workloads • Representative application pipelines • FFmpeg • OCR/ASR preprocessing
AI Gateway / AI Services	• HTTP/API processing • Request parsing • JSON/protobuf serialization • gRPC/RPC	• Routing • Concurrency • Rate limiting • Caching	• Nginx/Envoy + wrk/h2load • gRPC/ghz • Serialization benchmarks • Representative AI API-gateway workloads
Agent Orchestration	• Agent runtime execution • Workflow/graph traversal • Task scheduling • Queues	• State transitions • Tool dispatch • Asynchronous execution • Retries	• Agent-framework workloads where available • Representative LangGraph/LangChain-style workflows • Task/queue/scheduling benchmarks
Memory / Context / RAG	• SQL/KV operations • State and caching • Parsing/chunking	• Vector indexing/ search • Graph traversal • Retrieval/ranking	• Redis + memtier • MySQL/PostgreSQL • MongoDB • FAISS/Milvus/VectorDBBench • Elasticsearch
Tool / Action Execution Fabric	• Tool invocation and dispatch • Shell/code execution • Authentication/ validation around tool calls	• Sandbox lifecycle • Tool-to-tool communication	• Representative agent tool traces • MCP/tool-server workloads • Sandboxed tool execution • Use-case-specific tool benchmarks
Model Gateways, Serving	• Routing • Request handling • Scheduling • Tokenization	• Data processing • Batching, streaming • Caching and KV-cache management	• Model/inference router tests • fastText • Tokenizer benchmarks • vLLM/SGLang/TensorRT-LLM host-side tests • CPU embedding/reranking workloads
OS / Containers / Drivers	• Sandbox startup/ teardown process creation • Filesystem activity	• Context switching • Isolation • Container/micro-VM lifecycle	• Firecracker/Kata/gVisor/container Runtime • Cold/warm startup • Scaling vs. concurrency • Sandboxes/tasks per second

Identity & Access Management / Security	• Authentication • Credential validation • Encryption • Authorization	• Policy checks • Guardrails • Redaction	Best measured in context within representative agentic workloads
Observability & Monitoring	• Logging • Tracing • Telemetry collection/ export	• Serialization • Compression • Metrics processing	Individual microbenchmarks may isolate pieces, but there is no meaningful standalone benchmark that represents the full function

Table 4: Representative CPU-visible Stack components and benchmark coverage

4.5 END-TO-END MEASUREMENT AND AGENT TRAJECTORIES

End-to-end benchmarking introduces two related problems. First, wall-clock task time includes CPU execution as well as waiting for networked services and model inference. Second, non-deterministic models can choose different strategies for the same prompt, producing markedly different tool sequences and runtimes. The source experiments behind this framework observe run-to-run variation of one to two orders of magnitude for the same task. A coding goal, for example, may be reached through web search in one run and repeated code-test cycles in another.

A practical response is to capture an agent trajectory: a sequence of tool calls together with their command inputs. Once captured, that trajectory can be replayed exactly, allowing the CPU-visible work to be measured without requiring the model to regenerate the same strategy on every run. This also separates CPU execution from time spent waiting on external services.

A single trajectory is not necessarily representative. For a given task, many runs may be needed to reveal the different strategies a model can take. Those observations can then be reduced to a smaller representative set of traces with similar tool behavior. Existing task-level agent benchmarks, such as software-engineering or computer-use suites, can provide sources of trajectories, while other workload classes will require comparable trace-generation methods.

This is an area where industry consensus is particularly valuable. Shared methods for capturing, selecting and replaying representative trajectories would make end-to-end CPU evaluation more reproducible without removing the real tool behavior that differentiates agentic workloads.

The size of the representative trace set is an open methodological question. The work behind this paper has found that many thousands of runs may be needed for a task before the space of model strategies is well represented. If those strategies can be grouped by similar tool use, the full population may be pruned to a smaller set of trajectories that preserves the relevant CPU behavior while making repeated benchmarking practical. That selection process needs clear validation rules so that convenience does not distort the workload.

Trajectory replay should also preserve the inputs needed to reproduce each tool operation. The goal is not simply to record that a database, compiler or search tool was called, but to capture enough of the command and input context to exercise comparable CPU work on each platform. This offers a path to reproducibility while keeping the benchmark grounded in tool sequences that real agents generated rather than hand-constructed synthetic mixes.

5. Conclusion: Toward Industry Consensus

Agentic AI changes the useful unit of performance from a model invocation to a completed task. The CPU participates across that path through orchestration, state, retrieval, tool execution, data movement, policy enforcement, observability and coordination with accelerators and external systems.

Evaluating that contribution requires two complementary views. Component benchmarking provides the control and attribution needed to understand CPU behavior and guide optimization. End-to-end measurement provides the task-level realism needed to determine whether those improvements translate into better latency, throughput, cost or energy per successful task. Use-case-specific tool workloads and replayable agent trajectories connect the two.

The value of this methodology is ultimately realized through optimization: By attributing bottlenecks to specific stages of the agentic execution path and validating whether local improvements translate into better completed-task latency, throughput, cost or energy efficiency.

The framework should remain deliberately evolvable rather than become a fixed specification too early. Agentic software stacks, deployment patterns and tool behavior are changing quickly. Its value will depend on validation against production behavior and participation from cloud providers, software developers, silicon vendors, enterprises and benchmark communities. Shared workload traces, deployment constraints, service objectives and observed bottlenecks will be essential to refine coverage and build common measurement practices for the agentic era.