

CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

InferSuite: A Microarchitectural Characterization and System Profiling of LLM Serving and Agentic Inference

CPU/GPU Time and TMA-Style Breakdown Across Serving and Agentic Workloads

Master's thesis in Computer Science and Engineering

Mohamad Al Shaar

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

InferSuite: A Microarchitectural Characterization and System Profiling of LLM Serving and Agentic Inference

CPU/GPU Time and TMA-Style Breakdown Across Serving and Agentic Workloads

Mohamad Al Shaar



UNIVERSITY OF GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

InferSuite: A Microarchitectural Characterization and System Profiling of LLM Serving and Agentic Inference
CPU/GPU Time and TMA-Style Breakdown Across Serving and Agentic Workloads
Mohamad Al Shaar

© Mohamad Al Shaar, 2026.

Supervisor and examiner: Ahmad Ali-Eldin Hassan, Department of Computer Science and Engineering
Industrial supervisor: Javier Picorel

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Performance characterization of a deployable GenAI inference stack and agentic workloads.

Typeset in LaTeX
Gothenburg, Sweden 2026

InferSuite: A Microarchitectural Characterization and System Profiling of LLM Serving and Agentic Inference
CPU/GPU Time and TMA-Style Breakdown Across Serving and Agentic Workloads
Mohamad Al Shaar
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Deployed LLM systems are commonly treated as GPU workloads. In practice, retrieval, cache lookup, storage, and agent tool execution also consume CPU and memory resources. It remains unclear where time and machine capacity are spent across the full system. This thesis addresses that question with InferSuite, a deployable LLM service and measurement suite for quantifying time and resource use across retrieval, semantic caching, and model serving. Separately from the service, the same measurement method is applied to two existing agentic workloads, a repository-repair agent and a browser agent, driven through their own unmodified harnesses. Everything runs on one bare-metal workstation and is measured through container-scoped hardware counters, Top-Down Microarchitecture Analysis, and GPU profiling, with every measurement split into what happens during inference and outside it.

The two studies give opposite answers. In the service, GPU generation holds 95 to 99 percent of request time, and the largest CPU consumer is the serving engine's synchronization busy-wait, about two cores that look healthy on every counter while doing no useful work, most of which a blocking wait policy hands back. In the agents, the CPU does the real work outside inference, executing the delegated tool payload and the orchestration harness, whose cost compounds with conversation length. The agent machine averages under two percent utilization of its partition yet briefly demands all of it. On the GPU, quantized matrix kernels dominate, compute-bound in prefill and memory-bound in decode.

The serving CPU should therefore wait more cheaply rather than be tuned, agent machines should be provisioned for bursts and shared rather than dedicated, and the harness, the serial term of the agentic loop, is the natural next target for accelerating agentic systems.

Keywords: LLM inference, LLM serving, Agents, Agentic AI, RAG, semantic cache, agents, CPU profiling, GPU profiling, top-down analysis.

Acknowledgements

I would like to thank my supervisor and examiner, Ahmad Hassan, for guidance, feedback, and support throughout this thesis. I would also like to thank my industrial supervisor, Javier Picorel, for discussions, technical direction, and help shaping the systems questions behind this work.

Mohamad Al Shaar, Gothenburg, 2026-10-07

Contents

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**
1.1 Thesis Focus . . . 2
1.2 Hypotheses . . . 2
1.3 Contribution . . . 2
1.4 System View . . . 3
1.5 Structure . . . 5

2 Background and Related Work **7**
2.1 LLM Serving, Retrieval and Caching . . . 7
2.2 Inference Phases . . . 8
2.3 Top-Down Microarchitecture Analysis . . . 9
2.4 Agentic Workloads . . . 10
2.5 Related Work and Gap . . . 11

3 Methods **13**
3.1 Artifact . . . 13
3.2 CPU Measurement . . . 13
3.3 Measurement Campaigns . . . 14
3.4 Service Workloads . . . 15
3.4.1 Service Deployment and Capture . . . 16
3.5 Agentic Workloads . . . 17
3.6 GPU Measurement . . . 18
3.7 Accounting . . . 21
3.8 Limits . . . 21

4 Results **23**
4.1 Service: Where the Time Goes . . . 23
4.2 Service: What the CPU Runs . . . 24
4.3 Service: How the Pods Run . . . 25
4.4 Agents: Where the Time Goes . . . 27
4.5 Agents: The Delegated Work . . . 31
4.5.1 What Runs Inside the Tool Fence . . . 31

4.5.2 The Counter View . . . 33
4.5.3 The Microarchitecture . . . 34
4.6 The GPU in the Same Language . . . 36

5 Discussion **41**
5.1 During Inference Versus Outside Inference . . . 41
5.2 Service Implication . . . 41
5.3 Agent Implication . . . 42
5.4 The Harness as an Acceleration Target . . . 42
5.5 The Orchestration Layer Across Agents . . . 44
5.6 Batchability and Exactness . . . 46
5.7 GPU Top-Down Interpretation . . . 46
5.8 Limits . . . 46

6 Conclusion **47**

Bibliography **49**

A Supplementary Service Figures **I**

B The Browser Agent: OpenClaw **III**

List of Figures

1.1 The InferSuite service path: a request passes the semantic cache, optionally retrieval, and then model serving. Each stage has its own CPU or GPU cost. 3

3.1 The deployed service stack on a single node. Red dashes mark the per-pod cgroup measurement fences, and all fenced scopes are measured in the same windows. Generation runs on the GPU, and every fenced pod runs on the node CPU. 17
3.2 The agent loop on a single node: the harness orchestrates, the sandbox executes the delegated work, and the served model generates. Red dashes mark the three cgroup measurement fences. 18

4.1 Where a request's time goes per output tier 23
4.2 Steady-state CPU usage per cell, stacked by pod (isolated capture, mean of three repeats) 24
4.3 CPU cost per 1,000 engine tokens (prompt plus generated), by pod and cell (isolated capture). The engine's busy-wait dominates the normalized cost. 25
4.4 Per-pod micro-architectural signature by input bucket (isolated capture, median run per cell, absolute scales). 26
4.5 Top-down Level-1 pipeline slots per pod (isolated capture, median run per row). 26
4.6 Where each SWE-agent episode's wall-clock time went 28
4.7 Who consumed each episode's CPU core-seconds 29
4.8 Orchestration timeline per episode, white gaps are model round-trips 30
4.9 Cumulative CPU work per fence over episode time 31
4.10 Tool-call structure per episode: call count and how many calls are heavy (left), median call duration (middle), and the share of episode wall-clock in which the tool fence is active at all (right). The right panel is the utilization statement: the fence doing the task's real computation is busy for 6 to 14 percent of the episode. 32
4.11 Average CPU utilization over the full episode, stacked by fence, read on two scales: cores (left axis) and share of the 20-core measured partition (right axis). Every episode averages 0.3 to 1.5 percent of the partition, and which fence dominates the average follows the task. . . 33

4.12 Per-side hardware signatures on absolute scales: the tool payload against the agent harness for each episode. The tool side is low-IPC irregular work, and the harness is a high-IPC single-core loop. 34
4.13 Top-down Level-1 pipeline slots per side. The tool payload is front-end-bound with substantial bad speculation, and the harness is retiring-dominated. 35
4.14 The full top-down tree per fence 36
4.15 GPU warp-scheduler slots re-binned into Intel-style top-down buckets 36
4.16 Microarchitectural signature of the dominant GPU kernels 37
4.17 GPU Speed-of-Light regimes for prefill, decode, and agent prompts . 38
4.18 GPU time by kernel class per regime. The quantized matrix kernels hold 79 to 90 percent of GPU time in every regime, and attention is a minority everywhere. 39
4.19 The dominant matrix kernel's bottleneck flip: compute-bound as the matrix-matrix product of prefill, memory-bound as the matrix-vector product of decode. 40

5.1 Harness CPU work against conversation length across fourteen live episodes on four repositories (log-log). The power-law fit has exponent 2.7 and explains 99.8 percent of the variance. Repository size, spanning 68 to 531 thousand lines, does not set the cost. 43

A.1 Per-pod CPU usage over the capture window. The engine holds its two-core spin continuously while the CPU-side RAG stages fire per request. I
A.2 Wall-clock split between GPU generation and CPU-side work across the twelve cells (isolated capture). This is the tier-donut data in per-cell bar form. I

B.1 Where each OpenClaw episode's wall-clock time went: model round-trip, tool execution, and the agent harness. IV
B.2 Who consumed each OpenClaw episode's CPU core-seconds. The compute-carrying tasks (jigsaw, PDF) are tool-dominated. On the fetch-only web tasks the API proxy is the largest consumer. IV
B.3 Average CPU utilization over the full OpenClaw episode, stacked by fence, in cores and as a share of the 20-core measured partition. Every episode averages at or below about 0.2 percent of the partition. . . . V
B.4 Orchestration timeline per OpenClaw episode: instantaneous CPU cores in the tool and harness fences. As in the main text, white gaps are model round-trips. VI
B.5 Cumulative CPU work per fence for the OpenClaw episodes, read as in the main text: flat stretches are model round-trips, and the slope of a rising curve is that fence's instantaneous usage in cores. VII
B.6 Tool-call structure per OpenClaw episode: call counts, median duration, and tool-active share of wall-clock. VIII
B.7 Per-side hardware signatures for OpenClaw, read as in the main text: the tool payloads against the Node.js harness on absolute scales. . . . VIII

B.8 Top-down Level-1 pipeline slots per side for OpenClaw. The Node.js harness is not retiring-dominated, unlike the Python harness of the code agents. IX
B.9 The full top-down tree per fence for OpenClaw, as shares of pipeline slots. Starred segments are labeled estimates, and the pale segment is unattributed fetch-latency cost. IX

List of Tables

1.1 Service components, their role in a request, and the reason for each choice. 4

3.1 Measurement setups. Everything is measured on one machine, the local workstation (Intel Xeon w5-3425, bare metal, full TMA, with an NVIDIA RTX A2000, 12 GB), in three roles. 15
3.2 Service paths. 16
3.3 Mapping from GPU warp-scheduler outcomes to the TMA-style categories used in this thesis. 20

4.1 Two ways to count agent activity, per SWE-agent episode: tool calls (actions in the agent's log) against CPU bursts (contiguous measured activity in each fence). 27

1

Introduction

LLM systems are often described as GPU workloads. This is true but incomplete. A deployed system also has retrieval, cache lookup, storage, routing, tokenization, and agent tool execution, and how such a system behaves is a property of the whole deployment, not of any single kernel inside it. A hot kernel can be profiled in isolation, but where the time goes, which component holds the machine, and what the hardware actually does while it runs only become visible when the running system is measured as a whole. That is the lesson of whole-system workload studies such as *Clearing the Clouds* and *Profiling a Warehouse-Scale Computer*, which showed that deployed services behave unlike the isolated benchmarks that stand in for them [1], [2]. This thesis takes that approach to a deployed LLM system and profiles the full system around inference rather than only the model kernel.

We build InferSuite, a containerized LLM service with retrieval augmentation, semantic caching, and model serving, together with a load generator and measurement suite for profiling it. Building the service rather than measuring an existing one is deliberate, because it makes every stage of the pipeline instrumentable, from the request-facing orchestrator down to the serving engine's host thread. Separately from the service, the same measurement method is applied to two existing agentic workloads, SWE-agent and OpenClaw, driven through their own unmodified harnesses, so the agent results describe the agents as they ship rather than a system built for the thesis. Due to time limitations, the agentic analysis focuses primarily on software engineering tasks.

The central question of this thesis is therefore: *Where are time and resources spent in a deployed LLM system, and which stages consume them?* To answer this question, the measurements separate inference from the work outside it. During inference, the model-serving engine drives the GPU. Outside inference, the CPU handles retrieval, cache lookup, and agent tool execution.

Three main findings emerge:

- **CPU behavior differs across stages.** Outside inference, the CPU performs workload-specific work such as retrieval, embedding, and agent tool execution. During inference, much of the serving CPU activity instead comes from synchronization with the GPU.
- **Agent CPU demand is low on average but highly bursty.** Prior studies report server utilization of roughly 10–50% [3], [4], [5], [6]. In our measurements,

agent execution averages below 2% utilization of the measured partition, while briefly demanding the full partition.

- **The agent harness is itself a significant CPU consumer.** Separating the harness from the tool payload shows that orchestration contributes substantial CPU work and lies on the serial path between model calls and tool execution, making it a potential target for acceleration.

1.1 Thesis Focus

This thesis tracks four things:

1. Where the time goes in a deployed system, how wall-clock and capacity split between CPU-side stages and GPU generation.
2. How much of the machine is actually utilized while each workload runs, from steady service load to the "idle with bursts" profile of the agents.
3. The CPU and GPU microarchitectural breakdown in a common TMA-style view.
4. Whether an agent's CPU cost is more than its tools.

1.2 Hypotheses

H1. During inference, the serving CPU sits idle.

H2. An agent's CPU cost is not only exhausted by its tools, the harness is a distinct consumer.

H3. The CPU character of the agent loop follows the language and runtime it is written in, on both sides of the fence, interpreted, just-in-time-compiled, and native implementations of the harness and of the delegated tasks leave distinct microarchitectural signatures.

H4. A machine dedicated to agent execution sits below reported server utilization, its demand arriving as brief many-core bursts rather than a steady draw.

1.3 Contribution

The contribution is a characterization, not a new serving mechanism. The thesis combines:

- CPU/GPU time accounting for cache, RAG, direct generation, and agent loops.
- Container-level CPU profiling that includes the process coordinating GPU inference.
- CPU TMA during inference and outside inference [7].

- GPU Nsight profiling mapped into a TMA-style breakdown [8], [9].
- A lineage-fenced separation of the agent harness from its tool payload, identifying the orchestration loop as the agentic loop's serial term and an acceleration target for algorithms and hardware.

Every measurement comes from one machine, a bare-metal workstation whose full performance-monitoring unit gives the same top-down lens to every result (Chapter 3). The agent workloads are driven by GLM-5.2 over its API, while the serving engine is self-hosted locally so its CPU is visible during inference.

1.4 System View

This thesis follows the workload-characterization style of *Clearing the Clouds* and *Profiling a Warehouse-Scale Computer*: measure the full deployed system, not only the obvious hot kernel [1], [2].

InferSuite has the service path shown in Figure 1.1. RAG adds retrieval before generation [10], [11]. Semantic caching can skip generation but still costs CPU-side lookup and validation [12], [13]. The model-serving system manages autoregressive generation and key-value cache memory [14].

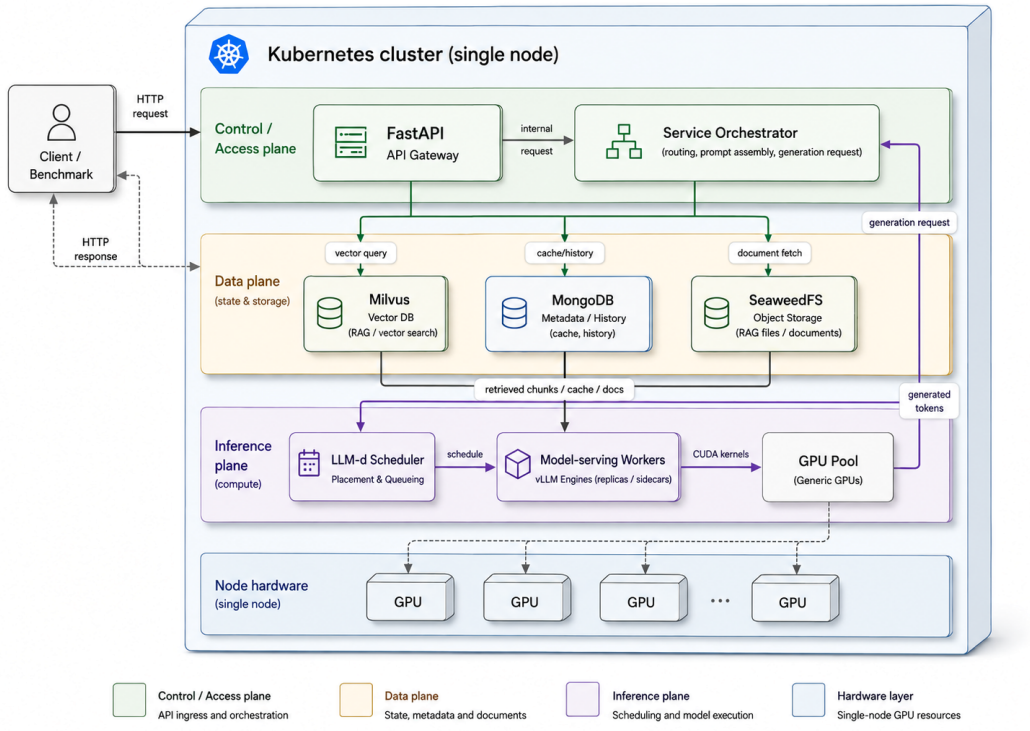


Figure 1.1: The InferSuite service path: a request passes the semantic cache, optionally retrieval, and then model serving. Each stage has its own CPU or GPU cost.

The components were chosen on one principle, each stage uses what a production

deployment would actually use, rather than a lightweight stand-in that would be easier to run but would not carry a representative cost. A characterization is only as meaningful as the system it characterizes, so the goal was a stack that an engineer would recognize as ordinary. Table 1.1 lists each stage, the component that implements it, and why it was selected.

Table 1.1: Service components, their role in a request, and the reason for each choice.

Stage	Component	Role and reason
Orchestration	FastAPI service	Accepts requests and drives the pipeline. An asynchronous Python service is the common shape for this layer, and it is the component that assembles the prompt, so it sits on the request-critical path [15].
Embedding	BGE base	Turns a query into a vector for both caching and retrieval. A widely used open embedding model, small enough to serve on the host CPU, which is where embedding commonly runs when the accelerator is reserved for generation [16].
Vector index	Milvus	Finds the nearest chunks among the indexed vectors. A purpose-built vector database running as its own service, rather than an in-process similarity library, because a real deployment separates the index from the application and that separation is where its cost becomes visible [17].
Object store	SeaweedFS	Holds the chunk text that retrieval returns. Deployments keep payload text in object storage and only vectors in the index, so every query performs a real object fetch rather than reading from memory [18]. The store follows the small-object design established for photo storage at scale [19], which matches the access pattern of retrieval, where each query fetches a handful of small chunks by key.
Metadata and history	MongoDB	Stores semantic-cache entries and conversation history. A document store is the usual choice for this shape of data [20].
Serving	llm-d gateway and vLLM	Routes to the serving engine and generates tokens [21]. vLLM is the reference open serving engine for paged attention and continuous batching [14], and the gateway in front of it is the Kubernetes-native way such engines are exposed.

Two consequences of these choices matter for the measurements. First, every stage runs as its own containerized service, which is how these systems are deployed

and, at the same time, exactly what gives each stage its own measurement fence. Production practice and measurement boundary coincide here, which is why the deployment is orchestrated by Kubernetes [22] rather than assembled as a single process, a monolithic implementation would have been simpler to run and impossible to attribute. Second, all inter-service traffic is real network traffic inside the cluster, so retrieval, lookup, and object fetch carry their genuine cost rather than a function-call cost.

The service also carries the parts a deployment needs but that this study does not exercise under load, such as authentication and request tracing. They are present so the artifact is complete, and they are outside the measured path.

1.5 Structure

Chapter 2 gives the minimum background. Chapter 3 describes the artifact and measurements. Chapter 4 is ordered as follows: service and CPU, agentic workloads and CPU, then GPU behavior. Chapter 5 and Chapter 6 summarize the implications.

2

Background and Related Work

This chapter defines the terms needed for the measurements and positions the thesis relative to prior work.

2.1 LLM Serving, Retrieval and Caching

A deployed LLM is typically accessed through a *serving system* rather than by invoking the model directly. The serving system accepts incoming requests, prepares them for inference, schedules model execution on the GPU, manages GPU memory, and returns generated tokens to the client. Unlike offline inference, where inputs can be processed as a fixed batch, an online serving system must handle requests that arrive at different times and have different prompt and output lengths. Efficient scheduling and memory management are therefore central to LLM serving.

A major source of GPU memory consumption during autoregressive inference is the key–value (KV) cache. Transformer attention computes key and value tensors for each processed token. During decoding, these tensors are retained so that previous tokens do not need to be recomputed every time a new token is generated. The KV cache therefore grows with the sequence length and with the number of requests being served concurrently. Poor KV-cache management can waste GPU memory and limit the number of requests that can be processed together [14].

vLLM is an LLM serving engine designed to execute autoregressive model inference efficiently on accelerators [14]. It receives inference requests, schedules their model execution, manages their KV caches, and performs token generation on the GPU. A central contribution of vLLM is *PagedAttention*, which manages the KV cache using fixed-size blocks inspired by virtual-memory paging. The blocks belonging to one sequence do not need to occupy one contiguous region of GPU memory. This reduces memory fragmentation and allows GPU memory to be allocated as a sequence grows.

More efficient KV-cache management increases the number of requests that can reside on the GPU simultaneously and therefore enables more effective batching. The serving engine can interleave work from multiple requests instead of waiting for an entire request to finish before admitting another one. In this way, vLLM sits between the request-facing application and the GPU: the application submits generation requests, while vLLM controls when and how the corresponding model computation is executed.

Retrieval-augmented generation (RAG) extends the request path with an external knowledge source [10]. Instead of sending the user's query directly to the language model, the system first uses the query to identify relevant information from a collection of documents. A common implementation maps the query and document chunks into an embedding space, searches an index for nearby vectors, retrieves the corresponding text, and inserts that text into the model prompt before generation.

The resulting request path therefore contains several stages that do not belong to the language model itself. Query embedding performs neural-network inference to obtain the search vector. Vector search identifies relevant entries from the index. The associated document content must then be fetched and inserted into the prompt before the request reaches the serving engine. RAG can consequently introduce CPU computation, memory accesses, storage accesses, and communication before GPU generation begins [10], [11].

Semantic caching attempts to avoid repeated model generation when a sufficiently similar request has already been processed. In contrast to an exact cache, which requires the same key to be repeated, a semantic cache represents requests using embeddings and compares them in a semantic space. If an incoming query is sufficiently similar to a cached query, the previous result can be returned without performing another complete model invocation [12], [13].

A semantic-cache lookup therefore also introduces work before model serving. The query must first be embedded, the cache must be searched, and the candidate match must be validated. A cache hit can remove the much larger cost of model generation, whereas a cache miss adds this lookup cost before continuing to the normal inference path. RAG and semantic caching therefore change not only the latency of an LLM request but also where computation and resource usage occur within the system.

2.2 Inference Phases

Autoregressive transformer inference is commonly divided into two phases: *prefill* and *decode* [23], [24]. The distinction follows from how the model processes the input sequence and how subsequent output tokens are generated.

During *prefill*, the complete input prompt is processed through the model. The tokens can be handled together within each transformer layer, allowing the large linear operations to be expressed as matrix–matrix computations. At the same time, the key and value tensors produced by the attention layers are stored in the key–value (KV) cache. These cached states represent the previously processed context and are retained for use during later generation [14], [23].

During *decode*, the model generates output autoregressively, one token at a time. Each newly generated token depends on the previously generated sequence, so one decoding step must complete before the next can begin. The model reuses the KV cache instead of recomputing the key and value tensors for all preceding tokens. However, the model weights still need to be accessed for each decoding step, and the KV cache grows as new tokens are added [23], [24].

The two phases consequently expose different forms of parallelism. Prefill processes many prompt tokens together and typically makes better use of the GPU's compute resources. Decode performs much less work per iteration and has less parallelism within an individual request, making data movement and memory bandwidth increasingly important. This asymmetry has motivated serving systems that schedule, chunk, or even place prefill and decode on separate resources [23], [24], [25].

The distinction is also reflected in serving latency. Prefill largely determines the time to first token (TTFT), while repeated decode iterations determine the time between subsequent output tokens, commonly expressed as time per output token (TPOT) [23]. Prompt length therefore primarily changes the amount of prefill work, whereas output length determines how many decode iterations must be executed.

The model used in this thesis is weight-quantized. Weight quantization represents model parameters at reduced numerical precision, reducing their storage and memory-transfer requirements while introducing quantized matrix kernels into the inference path [26], [27]. The distinction between matrix–matrix-like prefill execution and matrix–vector-like decode execution is therefore particularly relevant to the GPU measurements presented later in this thesis.

2.3 Top-Down Microarchitecture Analysis

Top-Down Microarchitecture Analysis (TMA) provides a hierarchical way of characterizing how a processor uses its available pipeline capacity [7], [28]. A modern out-of-order CPU can be viewed broadly as a front end and a back end. The front end fetches architectural instructions and decodes them into lower-level micro-operations (μops). The back end schedules these micro-operations, executes them when their operands and execution resources are available, and finally retires them, committing their results to the architectural state [29].

Because a superscalar processor can handle several micro-operations in one cycle, TMA expresses its available execution capacity in terms of *pipeline slots*. A pipeline slot represents one opportunity for a micro-operation to make progress through the pipeline. Over a measurement interval, TMA accounts for these slots and assigns each one to one of four top-level categories [7].

- **Retiring:** the slot contains a micro-operation that successfully completes and retires. This represents pipeline capacity used for completed execution.
- **Bad Speculation:** pipeline capacity is spent on work that is subsequently discarded or on recovery from incorrect speculation. Branch mispredictions are a common cause, although events such as machine clears can also contribute.
- **Front-End Bound:** the back end could accept additional work, but the front end does not deliver micro-operations quickly enough. The limitation therefore occurs before execution.
- **Back-End Bound:** micro-operations have reached the back end, but cannot proceed because their operands or required execution resources are not available.

The four categories form the first level of the TMA hierarchy and together account for the processor's pipeline slots [7]. The percentages reported by TMA should therefore be interpreted as shares of pipeline capacity, rather than as direct fractions of application wall-clock time.

Each top-level category can be decomposed further. Front-End Bound separates into *fetch latency* and *fetch bandwidth*. Fetch latency represents periods in which instruction delivery is interrupted, for example by instruction-cache or instruction-TLB misses. Fetch bandwidth instead describes sustained under-delivery of micro-operations even when the front end is not completely stalled [28].

Back-End Bound separates into *memory bound* and *core bound*. Memory-bound execution is delayed by the cache and memory hierarchy, whereas core-bound execution is limited by resources inside the processor core, such as execution ports, functional units, or scheduling capacity [7], [28]. Bad Speculation can similarly be divided into branch mispredictions and machine clears, while Retiring can be separated into lighter and heavier classes of retired operations on processors that expose these metrics [28].

The hierarchy is intended to be interpreted from the top down. A lower-level metric is most useful when its parent category is itself significant. For example, a high cache-miss rate does not necessarily indicate that memory is the performance bottleneck if Back-End Bound contributes little to the overall slot breakdown. The method therefore first identifies which top-level category dominates and then uses the lower levels to determine the underlying cause [7].

One interpretive rule is particularly important in this thesis. *Retiring* means that micro-operations completed successfully; it does not mean that they performed useful application-level work. A polling or synchronization loop can retire instructions efficiently while making almost no progress toward completing the request. TMA therefore describes *how* the processor executed, but not by itself *what* the software was doing. For this reason, the TMA results in this thesis are interpreted together with software samples that identify the code responsible for the measured CPU activity.

2.4 Agentic Workloads

Agents alternate between model generation and external actions [30], [31]. Each turn, the model reads the conversation so far and proposes one action, the harness executes that action on the machine, and the result is appended to the conversation for the next turn. The loop repeats, growing the conversation, until the model declares the task done. This thesis measures two agents whose layouts on the machine differ in exactly the way that matters for CPU measurement.

SWE-agent performs repository and test work on real software issues [32]. Its harness is a Python process running directly on the host. At the start of an episode it loads the task and starts an isolated per-task sandbox container holding the project's repository and toolchain. Each turn it runs the model's single proposed command

inside that sandbox, a file view, an edit, a build, or a test run, and returns the output to the model, until the model submits its fix. The sandbox is the tool side of the loop, so the harness and the delegated work naturally live in separate containers.

OpenClaw performs general browser and computer-use tasks, such as digesting web pages and documents, crawling, and image work. It ships as a single container in which a Node.js gateway runs the loop and spawns every tool it needs, browsers, interpreters, and converters, as child processes beside itself. Harness and tools therefore share one container, which makes its measurement the more delicate case. How each layout is fenced for measurement follows in Chapter 3.

2.5 Related Work and Gap

Research on LLM serving has largely focused on improving model-serving throughput, memory efficiency, and latency. PagedAttention [14] improves KV-cache memory management, DistServe [23] separates prefill and decode execution, and Sarathi-Serve [24] uses chunked prefill scheduling. These works primarily optimize the model-serving path.

RAG research, beginning with retrieval-augmented generation [10], studies how external retrieval can augment model generation, while RAGBench [11] focuses on evaluating RAG systems. Semantic-cache systems such as MeanCache [12] and the approach of Kim et al. [13] aim to reuse previous results and reduce repeated model execution. These lines of work motivate the retrieval and caching components considered in this thesis, but do not provide a whole-system microarchitectural characterization of the resulting deployment.

Top-Down Microarchitecture Analysis [7] provides an established framework for understanding CPU pipeline behavior. On GPUs, Nsight Systems [9] and Nsight Compute [8] expose system-level and kernel-level behavior, respectively. The CPU and GPU measurement methodology used to relate these views is described in Chapter 3.

The closest methodological inspirations are whole-system workload characterizations. *Clearing the Clouds* [1] and *Profiling a Warehouse-Scale Computer* [2] demonstrate the value of studying complete deployed systems rather than only isolated kernels. Related datacenter studies have also examined how much of provisioned server capacity is actually used [3], [4], [5]. In this thesis, we ask the same utilization question with a focus on machines executing AI-agent workloads.

Agent research such as ReAct [30] and SWE-agent [32] studies reasoning, tool use, software-engineering tasks, and agent evaluation. These works focus on agent behavior and task performance rather than the CPU and GPU resource behavior of the complete agent loop.

The gap addressed in this thesis is the system-level behavior across the inference boundary. Prior serving work primarily targets model inference, while retrieval, caching, orchestration, and agent tool execution introduce additional host-side work.

This thesis studies these components together and distinguishes CPU behavior during inference from CPU work performed outside inference.

3

Methods

InferSuite was developed by the author for this thesis and is measured as a working artifact, following prior whole-system workload characterization [1], [2]. The deployment manifests, benchmark drivers, measurement scripts, and figure generators are all part of the artifact.

The methodological contribution is scope correction. CPU measurements are attributed to the whole container or pod cgroup, not only to the visible request-facing process. This matters because the process accepting API requests can appear almost idle while the worker coordinating GPU inference consumes the CPU. In this artifact, matching only the API process undercounts model-serving CPU use, while matching the full cgroup includes the model-serving worker.

3.1 Artifact

InferSuite contains:

- A service orchestrator with RAG, semantic cache, history, tracing, and model-serving calls.
- Deployment manifests for the service orchestrator, vector index, metadata store, object store, routing layer, and model-serving system.
- Service benchmarks for cache, RAG, pure-fetch, and LLM-direct paths.
- Agentic workloads for SWE-agent and OpenClaw.

The public code repository is `https://github.com/MohamadAlshaar/InferSuite`.

3.2 CPU Measurement

CPU profiling uses container-scoped Linux performance counters rather than process-local measurements of only the request-facing service [33]. This is required because the process accepting a request is not necessarily the process consuming the CPU. In the local serving setup, for example, a separate vLLM worker coordinates and synchronizes GPU inference, while the request-facing process itself can appear nearly idle.

The counter measurements characterize instruction throughput, instruction delivery, branch behavior, the cache hierarchy, memory latency, and vector floating-point activity. The main metrics used in the results are described below.

Instructions per cycle (IPC) measures the number of retired instructions relative to elapsed processor cycles. It provides a compact measure of instruction throughput. A high IPC does not necessarily imply useful application work; a tight polling loop can retire instructions quickly while making little application-level progress [28], [34].

Micro-op cache coverage. Modern Intel processors can supply previously decoded micro-operations from the Decoded Stream Buffer (DSB), commonly referred to as the micro-op cache, instead of repeatedly fetching and decoding the corresponding instructions. The DSB or micro-op-cache coverage reported in this thesis is the share of micro-operations supplied through this path. High coverage indicates that the front end can largely bypass repeated instruction decoding [29].

Branch MPKI. Branch MPKI reports branch mispredictions per thousand retired instructions. It normalizes branch-prediction failures by the amount of executed code and is used to identify workloads with substantial control-flow irregularity [28].

Cache MPKI. Cache behavior is reported as misses per thousand instructions (MPKI) at the L1 instruction cache, L1 data cache, L2 cache, and last-level cache (LLC). The metric indicates how frequently instruction or data accesses fail at each level of the hierarchy relative to the amount of executed work [28], [34].

Average memory-access time (AMAT), reported in processor cycles, estimates the average latency of memory accesses across the cache and memory hierarchy [34].

Memory-level parallelism (MLP) describes the degree to which multiple memory operations can be outstanding concurrently. Higher MLP allows the processor to overlap memory accesses and hide part of their individual latency [34].

Packed floating-point activity. Packed floating-point activity indicates whether floating-point work is executed using SIMD vector instructions rather than scalar operations. It is used here primarily to distinguish vectorized numerical stages, such as the embedding computation, from scalar control and orchestration workloads [29].

The service counters are collected in four passes covering instruction throughput and micro-operation delivery, the cache hierarchy, stalls and address translation, and memory and floating-point behavior. The agent captures use the same counter families and additionally report AMAT and MLP. The passes are required because all desired hardware events cannot be counted simultaneously [33], [35].

3.3 Measurement Campaigns

Every measurement in this thesis comes from one machine, the local workstation, used in the three roles summarized in Table 3.1. Its bare-metal Intel PMU provides the full Top-Down hierarchy for every CPU result, and its GPU permits direct Nsight access, so the whole thesis carries one hardware baseline and one analytical lens.

Table 3.1: Measurement setups. Everything is measured on one machine, the local workstation (Intel Xeon w5-3425, bare metal, full TMA, with an NVIDIA RTX A2000, 12 GB), in three roles.

Role	Model	Measures
Agent runs, API-driven	GLM-5.2 via API, harness and tools on the local CPU	agent harness and tool-execution CPU incl. TMA
Local service (k3s)	vLLM 0.24 (llm-d), Qwen2.5-7B-Instruct-AWQ, full storage stack	service time grid, per-pod attribution and TMA, idle control

Two constraints force this design and neither is a preference.

First, **model capability**. The agent benchmarks are hard by construction. Strong models resolve only a small fraction of SWE-bench issues [36], and open-weight models lag frontier models, failing through exceeded turn limits and prematurely self-judged completion [37]. A model capable enough to resolve these tasks does not fit the workstation's 12 GB GPU, so the agent runs are driven by GLM-5.2 over its API while the agent harness and every tool run on the local CPU. Full tool execution is reached, both the harness and the tool payload get full TMA, and the only cost is that the serving CPU is remote and therefore invisible in the agent runs. That serving CPU is measured separately, in the two local-serving roles below.

Second, **seeing the serving CPU during inference requires hosting the model locally with a full PMU**. Cloud GPU machines are virtualized, and virtualization removes the measurement capability this thesis needs. Rentable GPU instances expose either no performance counters at all or no top-down slots, and no rentable bare-metal machine offered both a full PMU and a GPU that current serving stacks support. The one machine available to this work that combines a full PMU with a usable GPU is the local workstation, which self-serves a 7B model for the during-inference and service roles. Everything in Chapter 4 is therefore measured there.

3.4 Service Workloads

The service workload is driven along four request paths, each isolating a different stage cost (Table 3.2), RAG-standard, RAG pure-fetch, semantic cache, and LLM-direct. RAG-standard includes retrieval, prompt augmentation, and generation. RAG pure-fetch isolates retrieval with no generation. Semantic cache measures embedding and lookup, covering both cache misses and hits. LLM-direct measures generation with no retrieval or cache path.

The measurements in Chapter 4 exercise the RAG-standard path, the one that touches every stage of the stack. For each path, inputs are grouped into short, medium, long, and very-long prompt buckets. Output length is fixed to three tiers of 64, 192, and 320 generated tokens. Every measured request is forced to emit

exactly its tier's token count, by disabling the model's end-of-sequence stopping and setting a minimum generation length. This decouples generation cost from sampling variance and makes the CPU/GPU split comparable across paths, buckets, and runs. The benchmark uses 20 measured requests per service cell, issued one at a time (concurrency 1), so per-request stage costs are not confounded by queueing.

Table 3.2: Service paths.

Path	Purpose
RAG-standard	OpenRAGBench corpus. Retrieval, prompt augmentation, and generation [10], [11].
RAG pure-fetch	Bare questions. Retrieval without generation.
Semantic cache	QQP pairs. Embedding, lookup, validation, and hit/miss behavior [12], [13].
LLM-direct	ShareGPT52K. Generation without RAG or cache hit.

3.4.1 Service Deployment and Capture

The service runs on the workstation under single-node k3s, from the same manifests as any cluster deployment. vLLM 0.24 (the llm-d serving image) serves Qwen2.5-7B-Instruct-AWQ on the A2000, the embedding model runs on the CPU, and the full storage stack (Milvus, MongoDB, SeaweedFS) runs alongside. The corpus is 328 papers from the OpenRAGBench dataset, chunked into 14,419 indexed vectors, and every request fetches its chunk text from the object store. Two measurement modes share this deployment. The timing grid runs the benchmark protocol, 20 requests per cell at concurrency 1 with exact-token forcing, and records per-request stage timings. The counter capture holds a continuous sequential load per tier and measures every pod cgroup, one 25-second sampling window for software attribution, and seven 20-second counter groups, the five portable groups plus two top-down groups (Level 1 and Level 2). An idle control with the load generator scaled to zero gives the no-load baseline. Every window is validated against multiplexing, zeroed counters, empty captures, and unverified load before use. Figure 3.1 shows the deployed stack and where the fences sit.

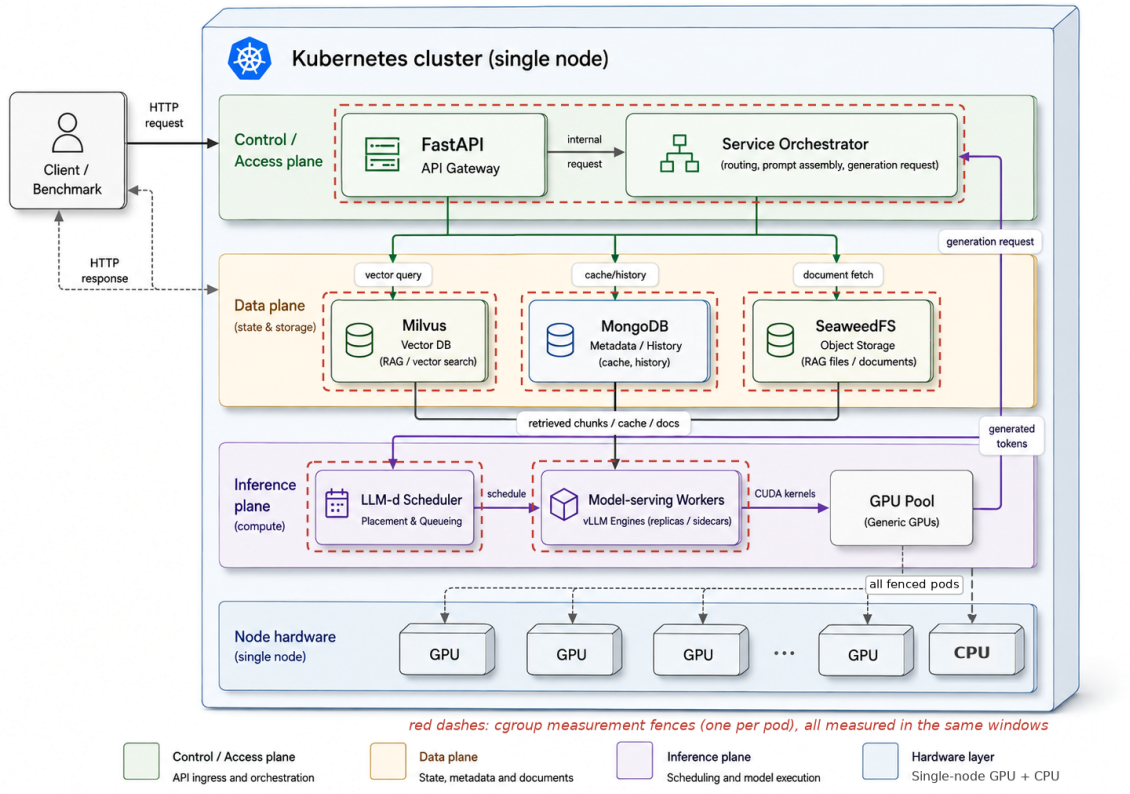


Figure 3.1: The deployed service stack on a single node. Red dashes mark the per-pod cgroup measurement fences, and all fenced scopes are measured in the same windows. Generation runs on the GPU, and every fenced pod runs on the node CPU.

3.5 Agentic Workloads

Agent time is split into inference time and local tool-execution time. The agent benchmarks are chosen to span agentic work rather than one task type, one coding workload and one general workload. SWE-bench tasks, driven by the SWE-agent harness, represent repository-level software engineering, where the agent navigates a real codebase, edits files, and runs the project's tests to fix a reported issue [32]. WildClawBench/OpenClaw represents general non-coding tasks, such as browser and tool orchestration over web pages, PDF documents, wikis, and images [30].

Within each benchmark, diverse tasks are selected so the CPU tool-execution side spans file and repository operations, numeric and library code, browser control, and I/O. Each task runs as a single long live episode driven by GLM-5.2, and every episode is captured under cgroup measurement fences in the same windows. The two harnesses are laid out differently on the node, and the fencing follows the layout. SWE-agent is a Python process running directly on the host. Every tool action executes inside a per-task sandbox container that holds the project's repository and toolchain, and a small local proxy relays the model calls to the API, so the harness, the sandbox, and the proxy each sit behind their own fence. OpenClaw instead ships

as a single container that holds both its Node.js gateway and every tool it spawns, so there is no container boundary between harness and tools to fence on. The separation instead rests on a process-lineage watcher that tags every descendant process by which process spawned it, populating the two fences by ancestry rather than by name. Both fences get the full counter suite including TMA. The main text reports four SWE-agent episodes, and the four OpenClaw episodes are in Appendix B. Figure 3.2 shows the loop and the fences.

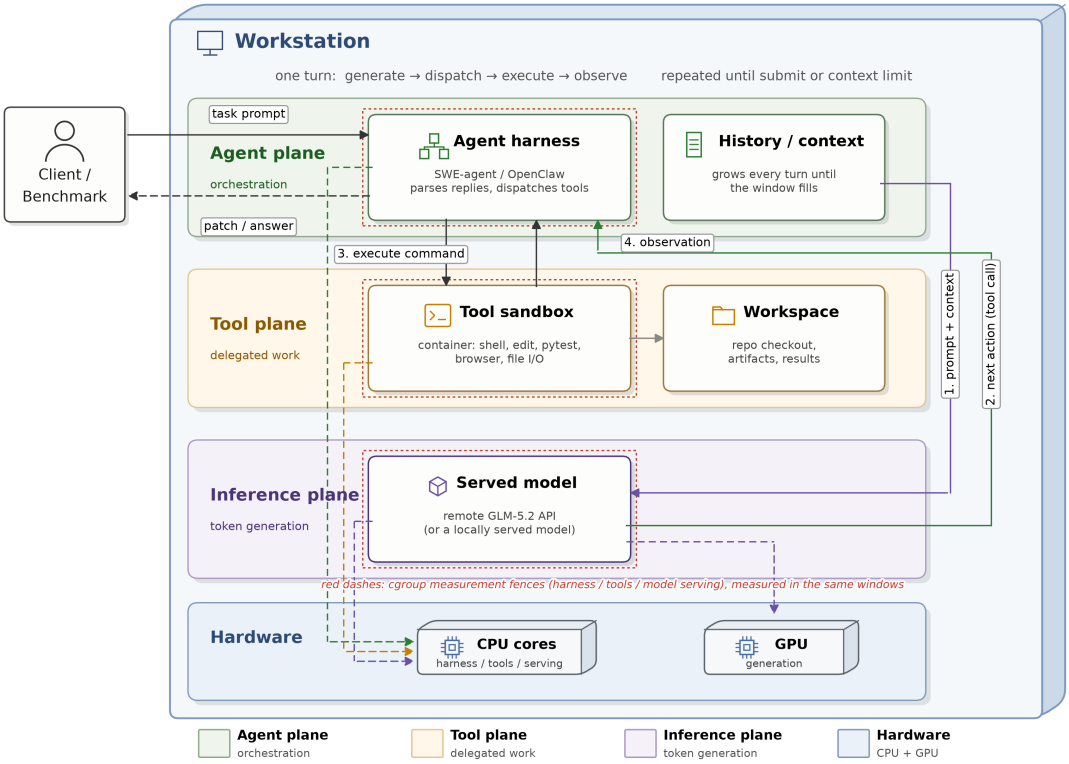


Figure 3.2: The agent loop on a single node: the harness orchestrates, the sandbox executes the delegated work, and the served model generates. Red dashes mark the three cgroup measurement fences.

3.6 GPU Measurement

GPU execution in this thesis uses NVIDIA's CUDA programming model. CUDA provides the software and execution interface through which a host CPU launches work on an NVIDIA GPU. GPU functions, called *kernels*, are launched by the host and executed by many lightweight GPU threads in parallel. Threads are organized into thread blocks, and blocks are distributed across the GPU's streaming multiprocessors (SMs) [38].

CUDA execution is generally asynchronous with respect to the host. After launching a kernel, the CPU can continue executing until it reaches an operation that requires the GPU result or explicitly synchronizes with the device [38], [39]. This distinction

is important in this thesis because CPU activity observed during inference can represent kernel submission, runtime bookkeeping, or synchronization rather than the model's numerical computation itself.

Within an SM, threads execute in groups called *warps*. On NVIDIA GPUs, a warp contains 32 threads and is the basic unit scheduled for execution. Multiple warps can reside on an SM simultaneously, allowing the scheduler to issue work from another warp when one is unable to proceed [38]. This is one of the main mechanisms by which the GPU hides instruction and memory latency.

GPU behavior is profiled using NVIDIA Nsight Systems and Nsight Compute. The two tools operate at different levels and are used together in this thesis [8], [9].

Nsight Systems provides a system-level timeline of the interaction between the host and the GPU. It records events such as CUDA API calls, kernel launches, GPU execution, memory operations, and synchronization. It is therefore used to determine when GPU work occurs and how that work interacts with CPU-side execution [9].

Nsight Compute profiles individual CUDA kernels and exposes hardware metrics describing how those kernels execute. These include warp scheduling, occupancy, execution-pipeline activity, cache behavior, memory throughput, and warp-stall reasons [8]. The two tools are therefore complementary: Nsight Systems identifies where GPU time is spent in the execution timeline, while Nsight Compute characterizes the kernels responsible for that time.

Several metrics are used to describe the resulting GPU behavior.

Instruction throughput. GPU instruction throughput provides a compact view of how quickly instructions are being issued and executed. In this thesis, it is used together with warp scheduling and execution-pipeline metrics rather than as a stand-alone measure of performance [8].

Occupancy and eligible warps. Occupancy measures the fraction of the hardware's maximum resident warps that are active on an SM. Higher occupancy provides more warps that may be used to hide execution and memory latency. An *eligible warp* is a resident warp whose next instruction is ready to issue. The number of eligible warps therefore indicates how much immediately runnable work is available to the scheduler [8], [38].

Compute-pipe activity. The GPU contains several execution pipelines for different classes of instructions. Tensor-pipe activity captures the use of Tensor Cores for matrix operations, while FMA and ALU activity describe pressure on more general arithmetic pipelines. These metrics are used to distinguish kernels dominated by dense matrix computation from those relying on other arithmetic resources [8].

Active lanes. CUDA follows a single-instruction, multiple-thread (SIMT) execution model. Threads within a warp follow the same instruction stream, but not every lane must be active for every instruction. The active-lane fraction therefore indicates how much of the warp participates in execution. Reduced lane activity can result from control-flow divergence or predication [8], [38].

Memory hierarchy. GPU memory accesses pass through a hierarchy that includes on-chip caches and off-chip device memory. L1 and L2 hit rates indicate how often accesses are satisfied at those cache levels, while DRAM bandwidth describes pressure on device memory. Together, these metrics indicate whether a kernel benefits from cache reuse or relies more heavily on off-chip memory traffic [8].

Registers per thread. Registers are allocated from a finite register file on each SM. High register usage per thread can therefore reduce the number of warps that can reside on an SM at the same time and, in turn, limit achievable occupancy [38].

Speed-of-Light analysis. Nsight Compute's Speed-of-Light analysis compares achieved compute and memory throughput with the corresponding hardware limits [8]. The values are reported relative to the available throughput and provide a high-level indication of whether execution is primarily limited by compute resources, memory bandwidth, or other latency effects.

TMA-style GPU view. There is no direct GPU equivalent of Intel's Top-Down Microarchitecture Analysis. To make the CPU and GPU results comparable, this thesis therefore groups warp-scheduler outcomes into a TMA-style hierarchy. This is an analytical mapping, not an implementation of Intel TMA on the GPU [7], [8].

The unit of accounting is a scheduler slot. In each scheduler cycle, an eligible warp may issue an instruction, which is treated as retiring-like progress. When no instruction issues, Nsight Compute reports reasons describing why the candidate warps could not proceed. These reasons are grouped according to whether progress is limited by instruction delivery, memory, execution resources, or synchronization. Table 3.3 summarizes the mapping used in this thesis.

Table 3.3: Mapping from GPU warp-scheduler outcomes to the TMA-style categories used in this thesis.

Warp-scheduler outcome	TMA-style category
Warp issues an instruction	Retiring-like
Instruction not yet fetched or decoded	Front-end
Waiting on far memory (DRAM, model weights, or KV cache)	Back-end: memory
Waiting on near memory (shared memory or L1)	Back-end: memory
Math pipeline saturated	Back-end: core
Memory input/output pipeline saturated	Back-end: core
Dependency on a previous instruction's result	Back-end: core
Warps waiting at synchronization points	Synchronization
Warp ready but another warp selected	Not a stall
Speculative rollback	Not applicable

Instruction-delivery stalls are therefore treated as front-end-like, while memory stalls are assigned to back-end memory and arithmetic-pipeline or dependency stalls to back-end core pressure. Synchronization remains a separate category. Bad speculation has

no direct counterpart in this mapping because GPU control flow does not use the same speculative execution and rollback mechanism as an out-of-order CPU. The per-kernel slot breakdown is weighted by kernel execution time before aggregation so that short kernels do not contribute disproportionately to the overall result.

The GPU is measured in three regimes. The *prefill-focused* regime uses a long input prompt and little fixed generation, emphasizing the work required to process the prompt and construct the KV cache. The *decode-focused* regime uses a short prompt and long generation, emphasizing repeated autoregressive token generation. Finally, the *agent-prompt* regime uses the real prompts produced by the measured agent workloads. These three regimes allow the GPU behavior of prefill, decode, and realistic agent traffic to be compared under the same profiling method.

Prefix caching is disabled for the corrected prefill captures so that previously computed prompt states do not remove the prefill work being measured.

3.7 Accounting

The thesis reports CPU-side time/capacity and GPU generation time. CPU core-seconds are used when CPU capacity matters more than elapsed duration. GPU time is then paired with Nsight Compute counters to produce a TMA-style view of GPU execution.

3.8 Limits

The main limits are the small GPU and model (the A2000 caps the locally servable model at 7B), batch-1 or single-stream interactive runs for the CPU measurements, and modest agent sample size (one long episode per task). The reported percentages are therefore mechanism-revealing measurements for this artifact, not universal constants.

4

Results

This chapter reads the measurements in one order for each workload family. First where the wall-clock time goes, then which software the CPU actually runs, then what the hardware counters say about how it runs. The service comes first and the agent workloads second. The chapter closes with the deep dive the earlier sections keep pointing at, the GPU itself. Dense plots are shown full-width so that the axes and labels remain readable.

4.1 Service: Where the Time Goes

The service is measured on the RAG path across three output tiers of 64, 192, and 320 generated tokens and four input buckets (short to long queries), with three isolated repeats per cell and exact output-token forcing, on the workstation described in Chapter 3. Every request emitted exactly its tier's token count and passed through live retrieval, and the capture is validated end to end (366 cross-checks, zero failures, engine IPC reproducing to within 0.5% across repeats). Figure 4.1 shows the wall-clock split per tier. GPU generation dominates elapsed time, at 95%, 98%, and 99% for the three tiers, and generation time is set by the output tier alone: about 23 ms per output token, roughly 1.9 s, 4.2 s, and 7.3 s per request, flat across input buckets.

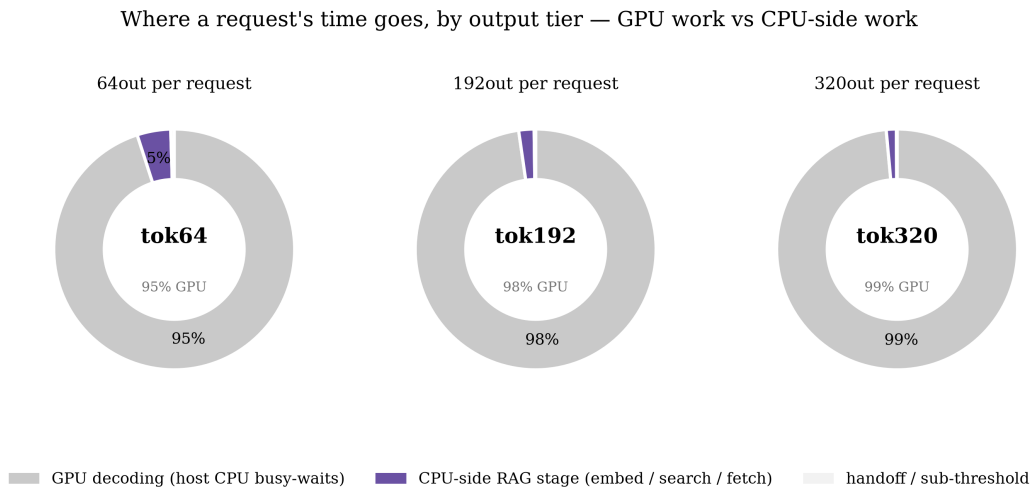


Figure 4.1: Where a request's time goes per output tier

The input bucket moves only the CPU side. The CPU-side cost per request is dominated by the query embedding, which the FastAPI pod runs in-process on the BGE encoder. Because the encoder's cost scales with query length, the CPU-side time grows with the input bucket while the vector search stays constant and retrieval returns the same number of chunks in every cell. The two benchmark axes therefore map cleanly onto the two engines: the output tier sets GPU time and the input bucket sets CPU time, independently of each other.

Implication: the service is not homogeneous CPU/GPU co-processing. It is a heterogeneous pipeline in which CPU-side stages feed a dense GPU generation stage, and the stages run serially: embed, search, fetch, then generate. The CPU-side work is on the request-critical path even when generation dominates the clock, and overlap is limited by causality, because the prompt cannot be assembled before retrieval returns. The per-pod timeline and the per-cell wall-clock split behind these figures are in Appendix A.

4.2 Service: What the CPU Runs

Figure 4.2 breaks the CPU work down per cell, stacked by pod. The vLLM host holds about 1.9 to 2.0 CPUs in every one of the twelve cells, at IPC 3.59 with a 99% uop-cache rate and zero vectorized floating point. The GPU does the arithmetic while the host threads busy-wait on it, so the engine's CPU cost is synchronization, not computation, and it does not vary with the workload. The only pod that sees a difference in CPU usage is the FastAPI pod, which runs the BGE query embedding in-process. It grows from about 0.1 cores on the shortest queries to about 0.5 cores on the longest, tracking the input bucket, and bursts to roughly ten cores while the encoder fires. Milvus (nearest-neighbour search over 14,419 chunk vectors), MongoDB (semantic-cache metadata), and the SeaweedFS object store are each a few hundredths of a core.

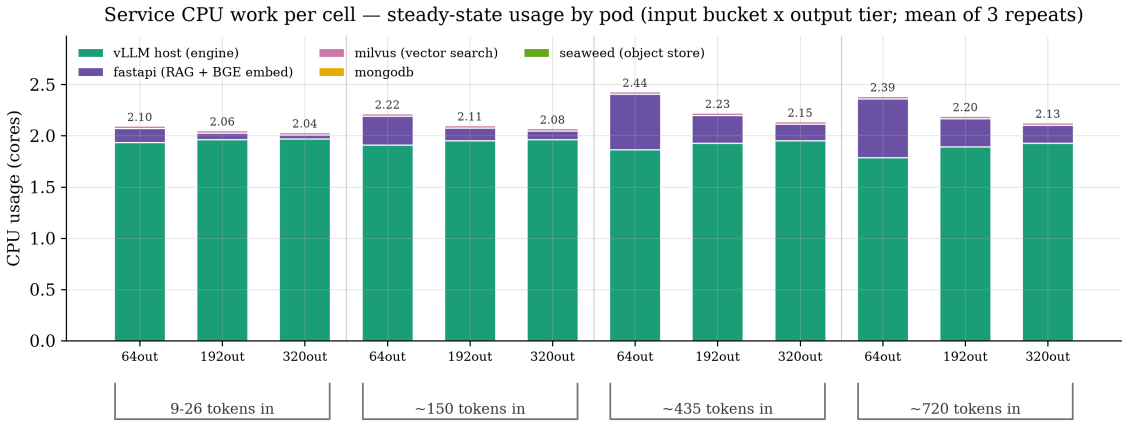


Figure 4.2: Steady-state CPU usage per cell, stacked by pod (isolated capture, mean of three repeats)

Normalized per token, the picture is the same engine (Figure 4.3): the CPU cost

per 1,000 engine tokens is almost entirely the engine spin, rising from about 4 to 11 core-seconds across the output tiers, with the CPU-side RAG stage a small, input-bucket-dependent addition on top.

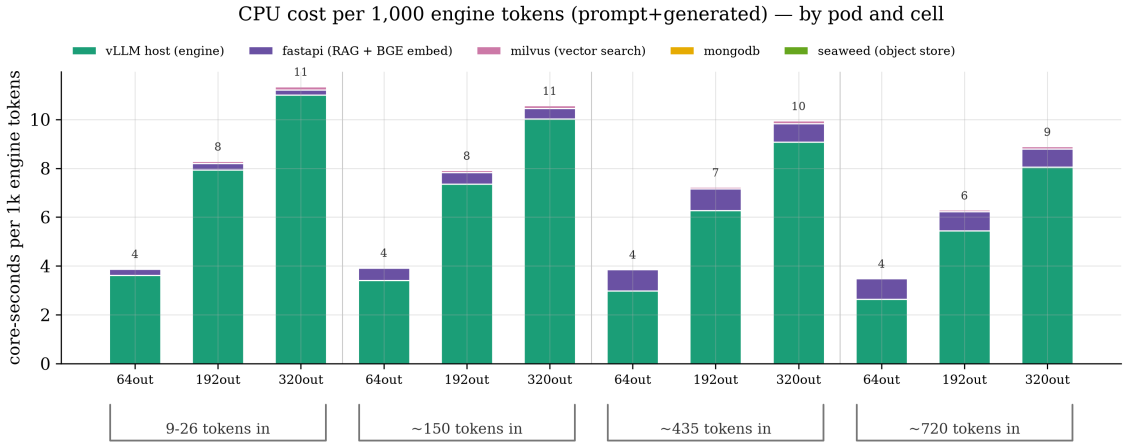


Figure 4.3: CPU cost per 1,000 engine tokens (prompt plus generated), by pod and cell (isolated capture). The engine's busy-wait dominates the normalized cost.

An idle control ties the engine's spin to load. With the load generator scaled to zero, the same engine cgroup drops from about 1.9 CPUs to 0.02 CPUs, parked in the kernel's event-polling path with no spin. The two-core busy-wait is therefore about 90 times load-induced, a per-inference cost rather than an always-on engine tax.

4.3 Service: How the Pods Run

Figure 4.4 shows the counter view. The engine runs at IPC 3.59 with a 99% uop-cache rate, zero cache misses at every level, and zero vectorized floating point: a register-resident spin that looks healthy on every counter. The FastAPI pod inverts the picture. All of its floating point is packed AVX-512 because it is streaming the BGE embedding GEMM, and its IPC climbs from 0.59 to 1.21 across the input buckets as the embedding grows the pod's useful work. Milvus is a vectorized but front-end-pressured search (IPC 0.85, high instruction-cache misses). MongoDB is an integer, branchy store.

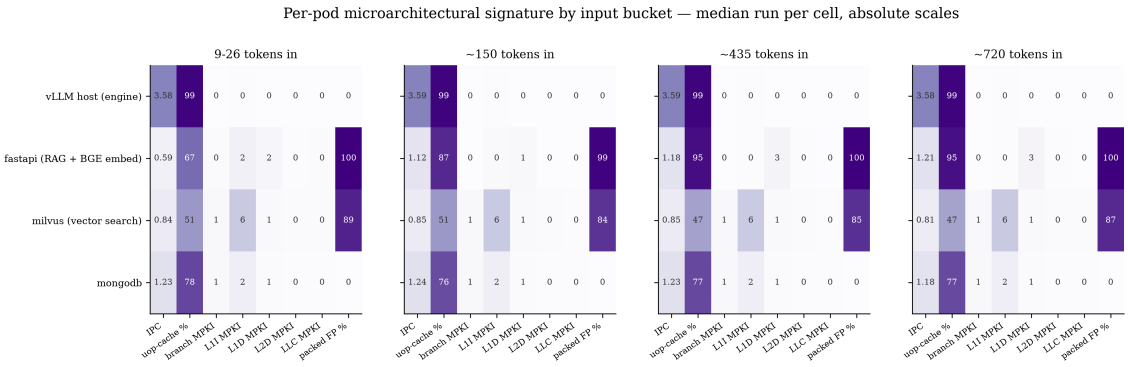


Figure 4.4: Per-pod micro-architectural signature by input bucket (isolated capture, median run per cell, absolute scales).

Figure 4.5 gives the top-down view, read as described in section 2.3. The engine is the only retiring-dominated pod, at 57% retiring against 39% backend-bound, stalling on execution ports while it retires trivial spin instructions and never touches the memory hierarchy. The FastAPI pod is 70% backend-bound at 25% retiring, which is what a real GEMM streaming data through saturated vector ports looks like. Milvus and MongoDB are front-end-bound, the regime typical of large branchy server binaries, and the micro-op supply in Figure 4.4 confirms it, the engine and FastAPI are served almost entirely from the micro-op cache, while Milvus falls back to legacy decode for about half of its supply. These signatures are tier-invariant. The engine reads about 57% retiring at every tier, so one tier's figures stand for all three.

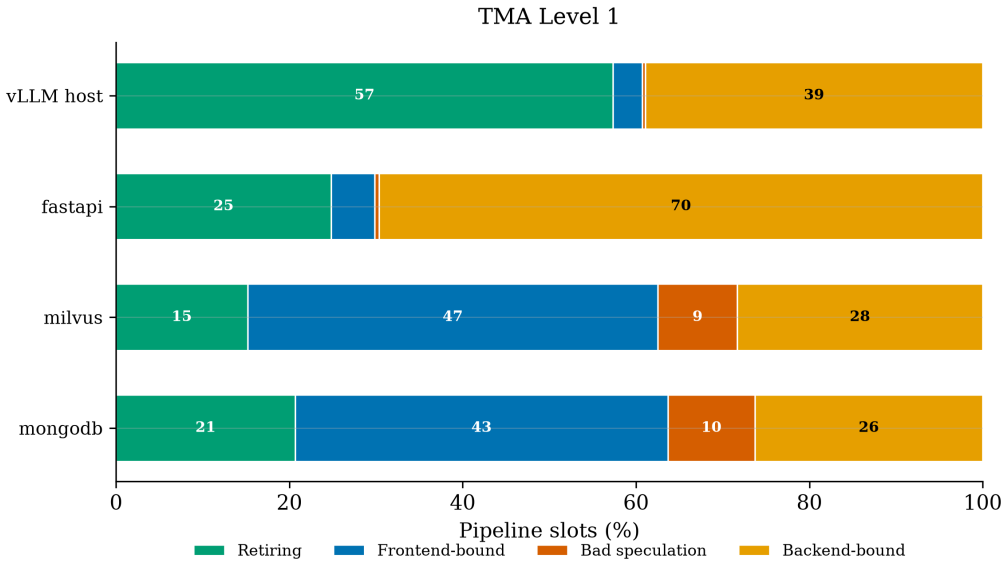


Figure 4.5: Top-down Level-1 pipeline slots per pod (isolated capture, median run per row).

4.4 Agents: Where the Time Goes

Where it ran: the agent harness and every tool run on the workstation described in Chapter 3, and the model is served by GLM-5.2 over its API. Four SWE-bench issues were resolved, spanning three languages, django and sympy (Python), babel (JavaScript), and fmt (C++). Each is a single long episode, isolated under the same CPU fences as the service, with a lineage watcher separating the agent harness from the tool payload it launches. The browser agent, OpenClaw, is measured the same way and is reported in Appendix B.

Reading the CPU numbers. Two units recur throughout this section. *Cores* is the rate of CPU time consumed relative to wall-clock time. A value of 1 means that the fence accumulates one CPU-second of execution per second of wall-clock time, summed across all of its threads; it does not imply execution on one specific physical core. Values above one indicate that CPU execution overlaps across multiple threads or cores.

Core-seconds is the total CPU time accumulated over the episode. It is related to the average core rate by

$$\text{core-seconds} = \text{average cores} \times \text{wall-clock seconds}.$$

Two counts also recur, and they come from different places. A *tool call* is one action in the agent's own log, the record of what the agent chose to run. A *burst* is one contiguous stretch in which a fence's measured CPU usage rises above a small detection floor of a few thousandths of a core, with gaps shorter than 0.4 s bridged so a single action is not split in two. The lightest actions never clear that floor. A directory change, a small edit, or a command dominated by waiting on disk uses too little processor time to register, so a fence's tool bursts run somewhat below its tool-call count (Table 4.1), while the harness, which fires once per turn, tracks the turn count closely. The shortfall is the featherweight actions, not a measurement gap. The per-burst figures later in this section plot one point per burst, not per logged call.

Table 4.1: Two ways to count agent activity, per SWE-agent episode: tool calls (actions in the agent's log) against CPU bursts (contiguous measured activity in each fence).

Task	Tool calls	Tool bursts	Harness bursts
django (Python)	127	109	138
sympy (Python)	389	389	568
babel (JavaScript)	94	74	82
fmt (C++)	114	112	130

Figure 4.6 shows where each episode's wall-clock time went. The loops are inference-dominated. The model round-trip, during which the local CPU only waits, is 80 to

89 percent of every episode, tool execution occupies 6 to 14 percent of the wall-clock, and the agent harness 5 to 15 percent. The episodes run from about seven minutes (babel) to 81 minutes (sympy). The longest episode spends the most on the harness, because the harness fires once between every one of the hundreds of model turns. The CPU side, though a minority of wall-clock, is on the critical path, since each tool result is the input to the next prompt.

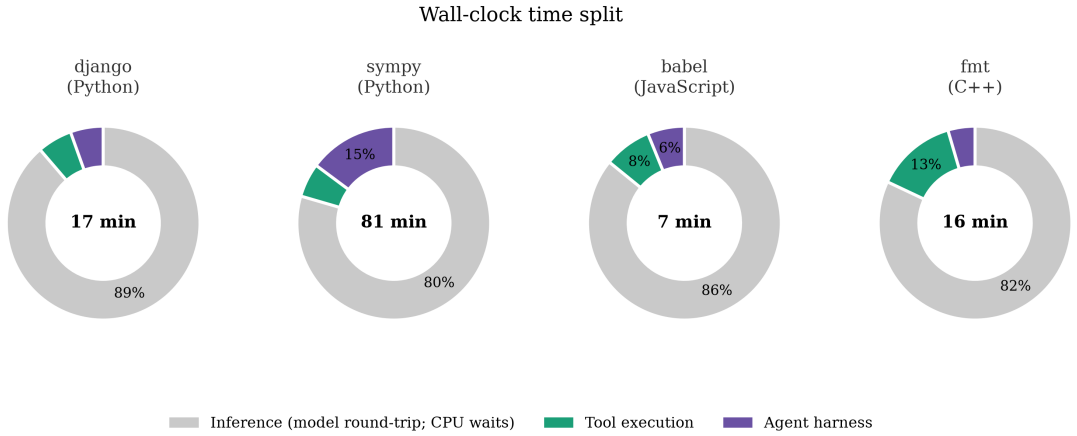
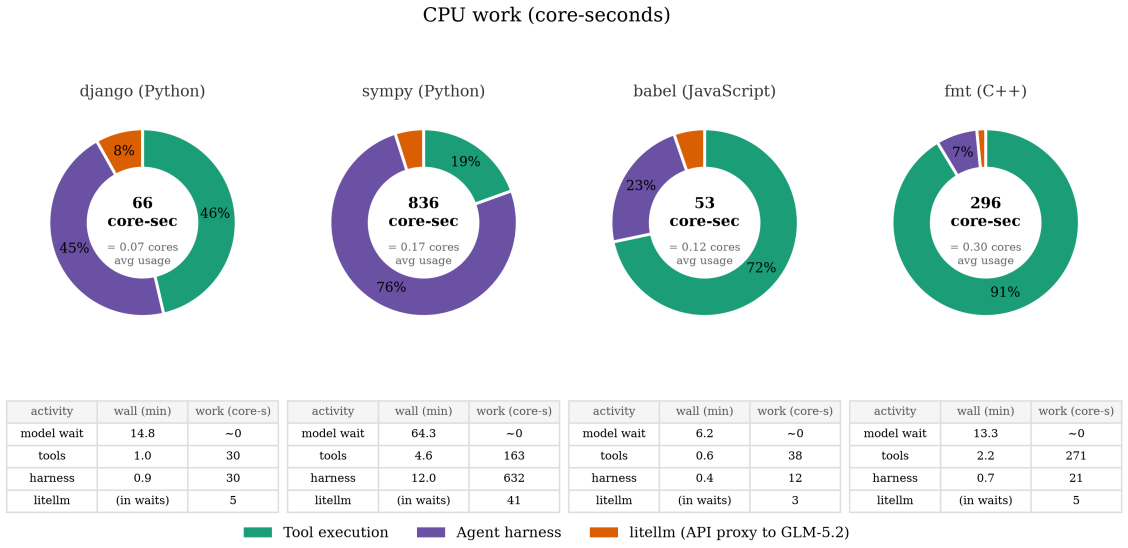


Figure 4.6: Where each SWE-agent episode's wall-clock time went

Figure 4.7 asks who did the episode's actual computation, in core-seconds. The totals are small: 53 to 836 core-seconds over episodes lasting many minutes, an average of 0.05 to 0.30 busy cores, because the machine spends most of the episode idle, waiting on the remote model. How that CPU work divides between the harness and the tool payload is task-dependent. The C++ episode (fmt) is 91 percent tool payload, dominated by compiling and linking the library and its tests, while the longest Python episode (sympy) is 76 percent harness, the cost of running hundreds of ever-longer agent turns. The same agent loop can therefore be harness-dominated or payload-dominated depending only on what the tools do.



activity	wall (min)	work (core-s)
model wait	14.8	~0
tools	1.0	30
harness	0.9	30
litellm	(in waits)	5

activity	wall (min)	work (core-s)
model wait	64.3	~0
tools	4.6	163
harness	12.0	632
litellm	(in waits)	41

activity	wall (min)	work (core-s)
model wait	6.2	~0
tools	0.6	38
harness	0.4	12
litellm	(in waits)	3

activity	wall (min)	work (core-s)
model wait	13.3	~0
tools	2.2	271
harness	0.7	21
litellm	(in waits)	5

Figure 4.7: Who consumed each episode's CPU core-seconds

Figure 4.8 traces both fences over episode time. The white gaps, when neither fence is busy, are the model round-trips that dominate the clock. The tool fence spikes to several cores on the heavy build and test calls, while the harness stays pinned near one core between turns. The episodes differ in shape, from the short bursty builds to sympy's near-continuous harness activity across 81 minutes, but every one is mostly white.

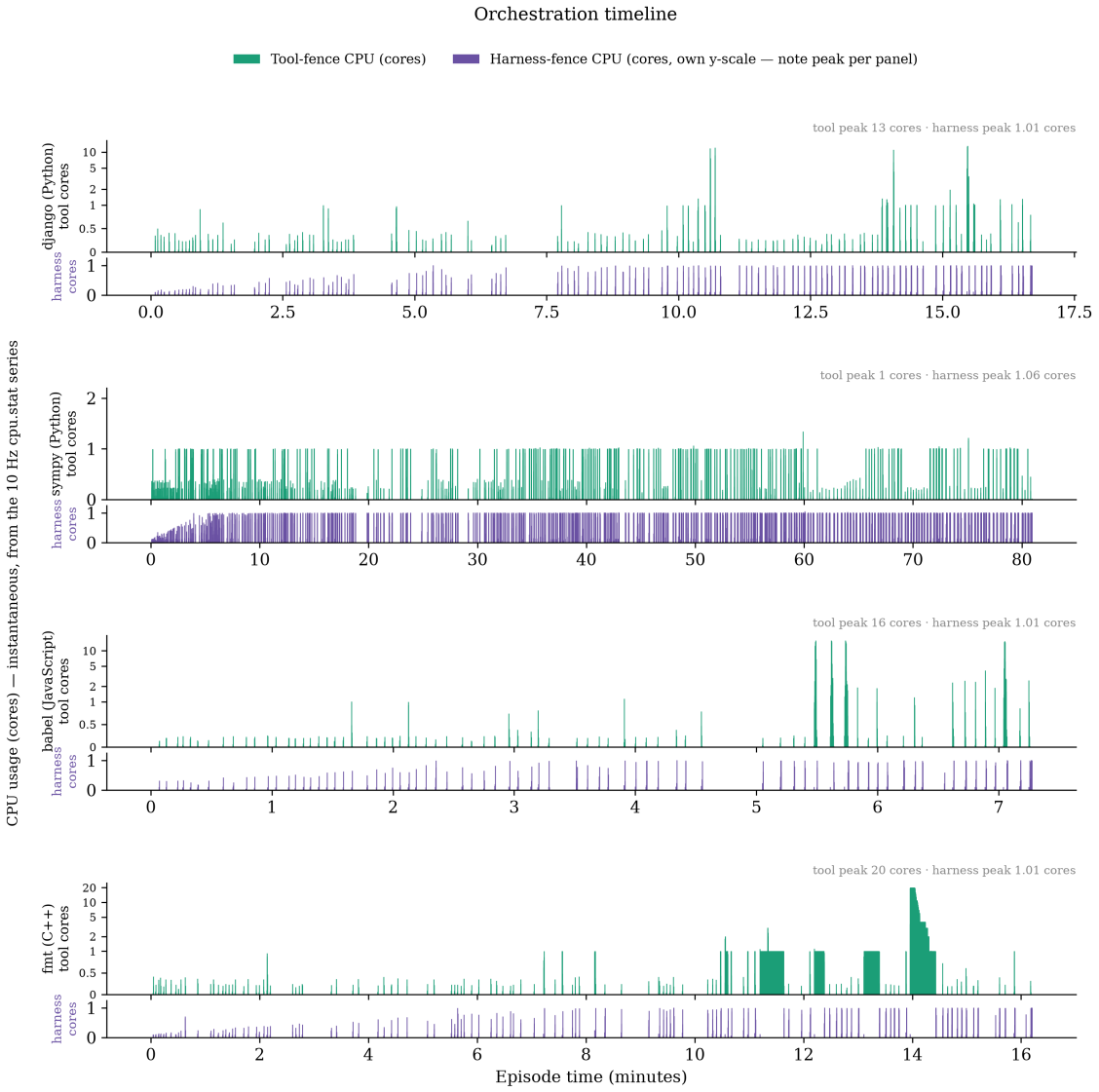


Figure 4.8: Orchestration timeline per episode, white gaps are model round-trips

Figure 4.9 shows the same series accumulated: each fence's core-seconds as they build over the episode. Flat stretches are the model round-trips, where nothing accrues, and the slope of a curve while it rises is that fence's instantaneous usage in cores. The two growth patterns separate cleanly. The harness accrues smoothly, once per turn, and on the sympy episode its curve visibly steepens as the conversation lengthens, the compounding cost of re-processing an ever-longer history. The tool payload accrues in steps. fmt's build lands most of the episode's total CPU in under a minute near the end, after fourteen almost-flat minutes.

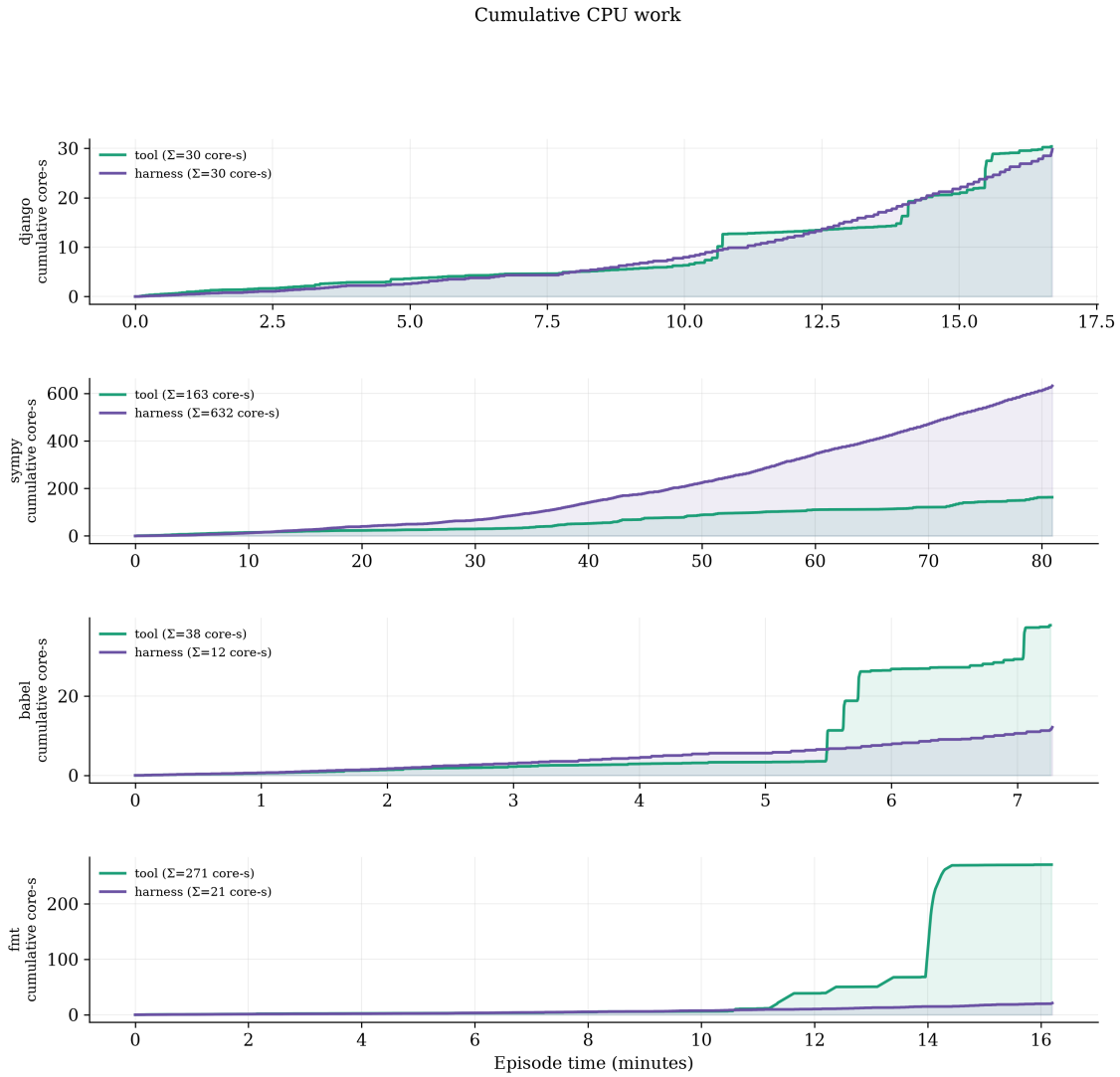


Figure 4.9: Cumulative CPU work per fence over episode time

4.5 Agents: The Delegated Work

This section reads the delegated CPU work, everything the agent does outside the model round-trip, through three lenses in turn: what programs run inside the tool fence, what the hardware counters say about them, and where they sit in the top-down hierarchy.

4.5.1 What Runs Inside the Tool Fence

The CPU outside inference is overwhelmingly the project's own build and test suite, run by the agent. The agent's built-in machinery, the file editor, the repository viewer, and the submit step, costs only a few core-seconds per episode. The delegated work is the codebase's own toolchain, not the agent framework.

Figure 4.10 shows how that payload is issued, and its right panel carries this section's

most important number. Each episode makes 94 to 389 tool calls, most of them light, and only a fraction leave a heavy burst of measured CPU above 0.3 cores (median heavy-burst durations of 0.1 to 0.3 seconds). The heavy calls fan out, as the timeline's per-panel peaks show: a single build or test call reaches 13 cores on django, 16 on babel, and saturates the 20-core partition on fmt, while the harness, bounded by the interpreter lock, never leaves one core.

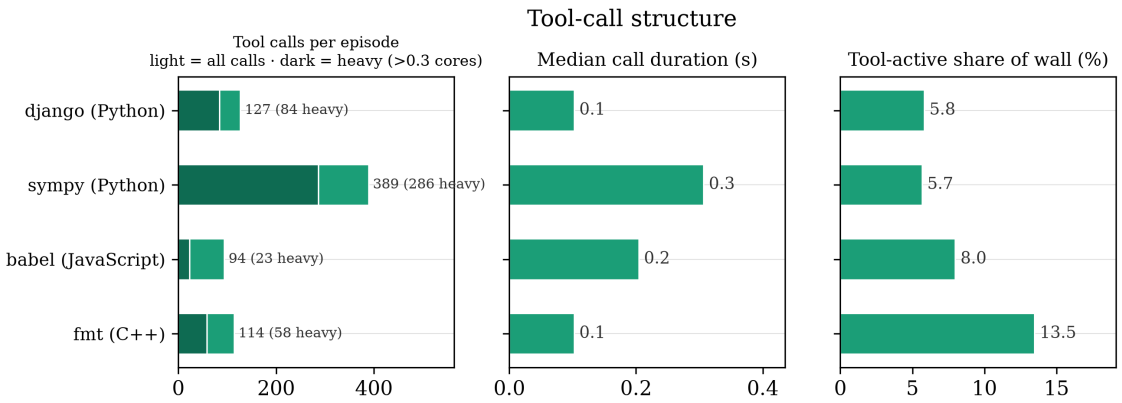


Figure 4.10: Tool-call structure per episode: call count and how many calls are heavy (left), median call duration (middle), and the share of episode wall-clock in which the tool fence is active at all (right). The right panel is the utilization statement: the fence doing the task's real computation is busy for 6 to 14 percent of the episode.

The right panel states how much of the machine all of this actually occupies. The tool fence, the place where the task's real computation happens, is active for only 5.8 to 13.5 percent of episode wall-clock. Figure 4.11 completes the utilization picture by averaging over the full episode: all three fences together hold 0.07 to 0.30 cores, which is 0.3 to 1.5 percent of the 20-core measured partition, and the stacking repeats the task dependence of Figure 4.7, mostly tool payload on the C++ build and mostly harness on the long sympy conversation. The utilization profile is therefore extreme at both ends: an average below two percent of the machine, sustained for minutes at a time, against brief demands for the full partition width when a build or a test suite fires.

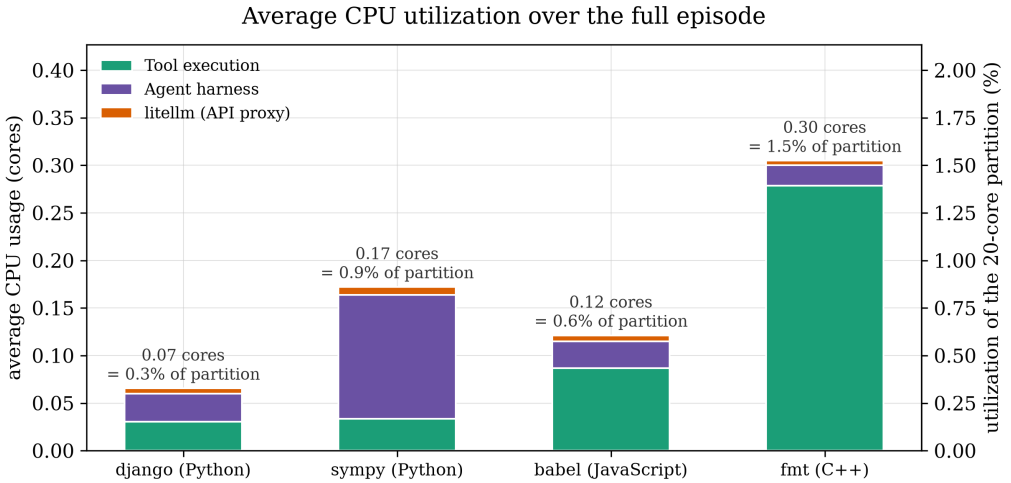


Figure 4.11: Average CPU utilization over the full episode, stacked by fence, read on two scales: cores (left axis) and share of the 20-core measured partition (right axis). Every episode averages 0.3 to 1.5 percent of the partition, and which fence dominates the average follows the task.

These numbers sit well below anything the utilization literature reports for conventional server workloads. Datacenter studies report that production servers spend most of their time between roughly 10 and 50 percent of maximum utilization, rarely fully idle and rarely fully busy [3]. Trace analyses of production clusters likewise find sustained CPU usage well below provisioned capacity, with reported aggregates below 20 percent [4], [5]. The agent machine measured here averages one to two orders of magnitude below even those figures, and unlike the steadily loaded servers in those traces, its demand arrives as rare, brief, many-core spikes rather than a stable draw. What this asymmetry implies for provisioning agent machines is discussed in Chapter 6.

4.5.2 The Counter View

Figure 4.12 places the two fences side by side on absolute hardware scales. The tool payload is irregular real work: IPC of 1.2 to 1.9 and instruction-cache miss rates of 8 to 24 per thousand instructions, the instruction-supply territory reported for warehouse-scale code [2]. The floating-point counters add that across all four tasks the payload executes essentially zero packed floating point. These SWE workloads are compiler, interpreter, and test-runner code, not linear algebra, and none of them reaches a vectorized numeric kernel. The harness is the opposite shape: IPC 2.8 across all four tasks, most of its micro-ops served from the micro-op cache (70 to 76 percent), and it holds close to a single core, the signature of the Python agent loop bounded by the global interpreter lock.

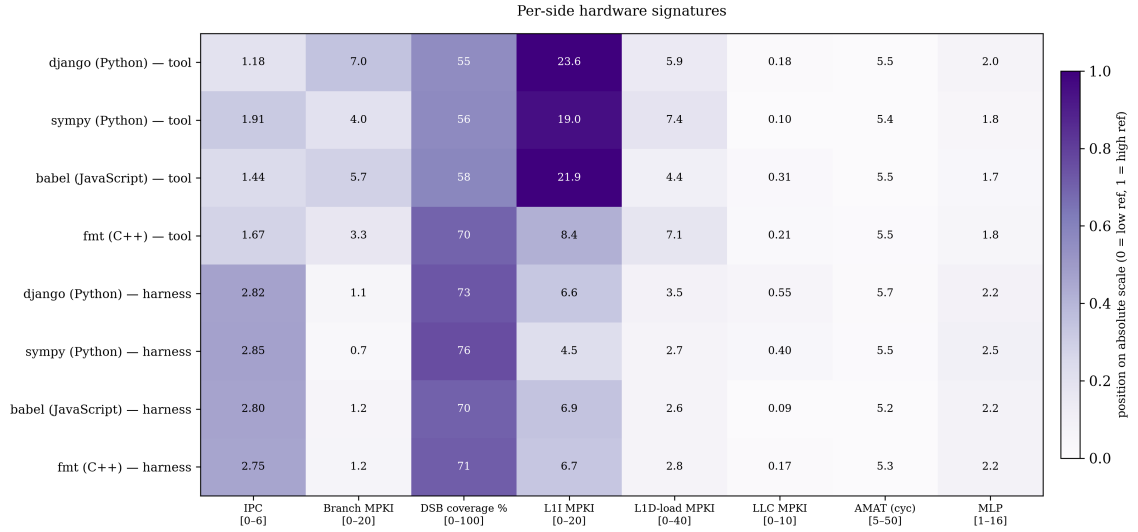


Figure 4.12: Per-side hardware signatures on absolute scales: the tool payload against the agent harness for each episode. The tool side is low-IPC irregular work, and the harness is a high-IPC single-core loop.

4.5.3 The Microarchitecture

Figure 4.13 gives the top-down view. The tool side of each task is a single signature averaged over that task's heterogeneous tool programs, the compiler, the test runner, version control, and shell commands. A single average is not strictly correct, since each program has its own profile, but it gives a faithful per-task overview because the fence is dominated by the project's own build and test toolchain. The tool payload is front-end-bound (32 to 40 percent of pipeline slots), retires only 27 to 32 percent, and carries real bad speculation (16 to 19 percent) with a third or more of its micro-ops coming from legacy decode rather than the cache, the classic profile of branchy, cold, irregular code. The harness inverts every one of these, about 42 percent retiring, 70 to 76 percent from the micro-op cache, low bad speculation. It looks microarchitecturally healthy, and it is doing almost none of the task's real work. This is the same caution restated on the agent side, high IPC and high retiring do not certify useful work. The harness retires efficiently while the low-IPC, front-end-bound tool payload is where the delegated computation actually happens, the mirror image of the serving engine, which retires just as efficiently while it spins on a GPU synchronization.

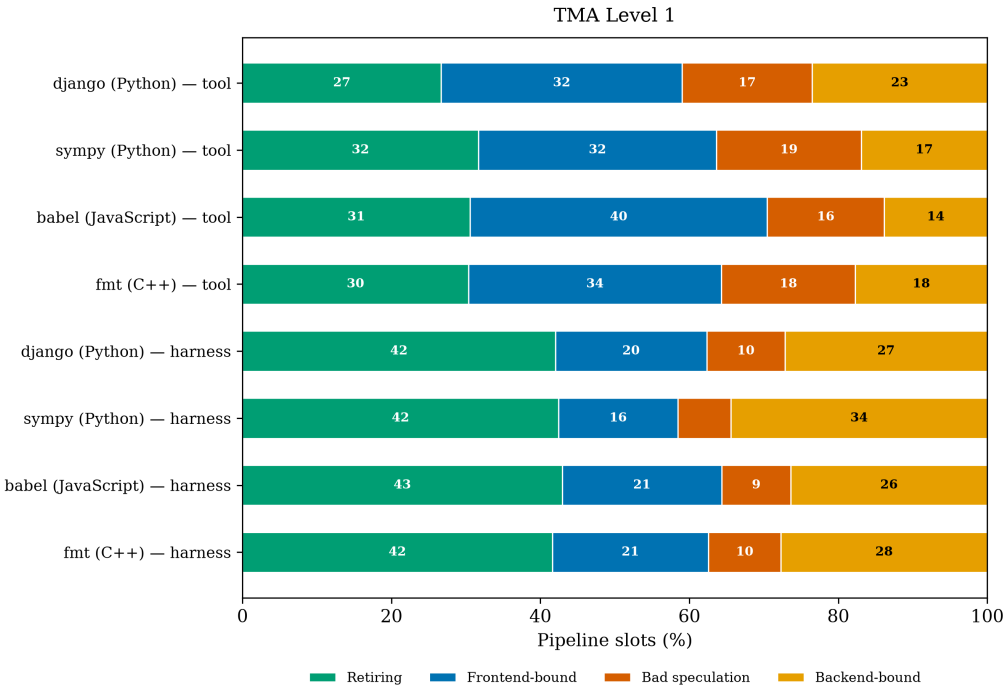


Figure 4.13: Top-down Level-1 pipeline slots per side. The tool payload is front-end-bound with substantial bad speculation, and the harness is retiring-dominated.

Figure 4.14 drills the same slot accounting down through the deeper levels of the hierarchy in a single view. The exactly measured children explain most of what the top levels report. The front-end's losses split between fetch latency, whose measured causes are instruction-cache misses, address-translation misses, and the restart bubble after mispredicted branches, and fetch bandwidth, the steady under-supply of decoded micro-ops that dominates for the tool payloads. On the memory side the tool work is overwhelmingly bound in the first cache level: data arrives quickly, but the branchy code cannot consume it any faster, so almost none of the pressure reaches main memory. Two subdivisions in the figure are marked as estimates rather than measurements, the split of fetch bandwidth between the two supply paths and the attribution of memory time across cache levels, and one slice of fetch latency remains explicitly unattributed. The figure shows it as such rather than absorbing it into a measured category.

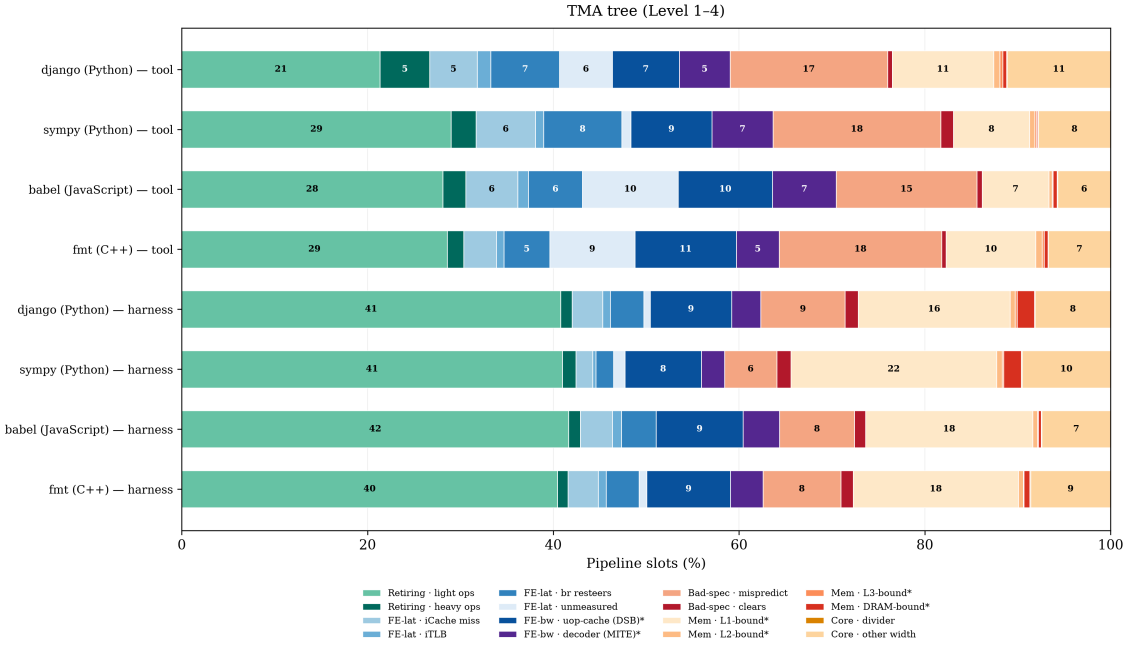


Figure 4.14: The full top-down tree per fence

4.6 The GPU in the Same Language

Figure 4.15 maps GPU issue behavior into the Intel-style buckets defined in Chapter 3. This is not a literal Intel TMA implementation, but it makes the CPU/GPU comparison explicit, both processors are read as slots that either delivered work or stalled for a stated reason.

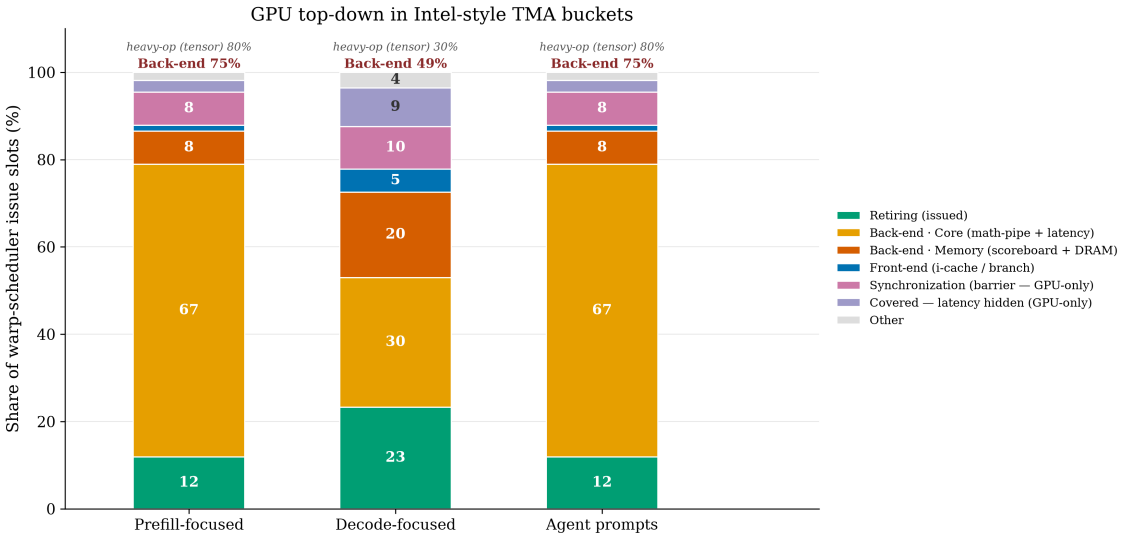


Figure 4.15: GPU warp-scheduler slots re-binned into Intel-style top-down buckets

The re-binning treats issued work as retiring-like. Memory, math-pipe, dependency, and throttle stalls are back-end-like, and instruction-delivery limits are front-end-

like. Bad speculation is not applicable because the GPU does not execute down mispredicted paths and roll back. Its closest analogues, SIMT divergence and instruction replay, are both negligible here, with lanes at 100%. Covered latency hiding is also small because occupancy is low, around 16 to 18 percent, so stalls remain exposed as genuine back-end pressure rather than being hidden by spare warps.

Figure 4.16 gives the corresponding GPU signature heatmap for the dominant kernels, the GPU analogue of the CPU counter view in Figure 4.12. It supports the same conclusion as the TMA-style mapping, GPU behavior changes by kernel and inference phase, with back-end pressure as the dominant signature rather than control-flow divergence. Prefill matrix kernels are tensor-pipe heavy, with the dominant prefill GEMM at 86% tensor activity, while decode kernels shift toward memory pressure.

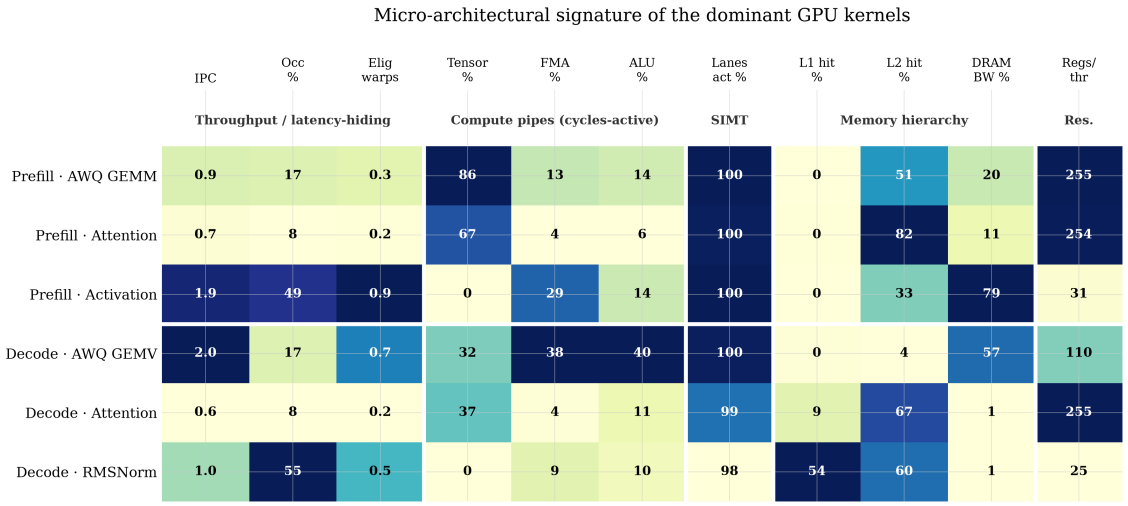


Figure 4.16: Microarchitectural signature of the dominant GPU kernels

Figure 4.17 shows the phase split at the Speed-of-Light level: prefill is compute-bound and decode is memory-bandwidth-bound. Prefill saturates the SM math pipes at 81% of peak, while decode is weight-bandwidth-bound at 61% memory throughput. The agent-prompt regime lands on the prefill point, at 81% compute and 28% memory throughput, because the measured agent turns are dominated by their long prompts. A real agent turn is therefore prefill-shaped, the GPU bottleneck is the compute-bound matrix work over the long prompt, not the memory-bound decode tail. In the dominant-kernel view, the decode matrix-vector kernel is still 67% memory-bound. The GPU optimization target consequently depends on prompt shape.

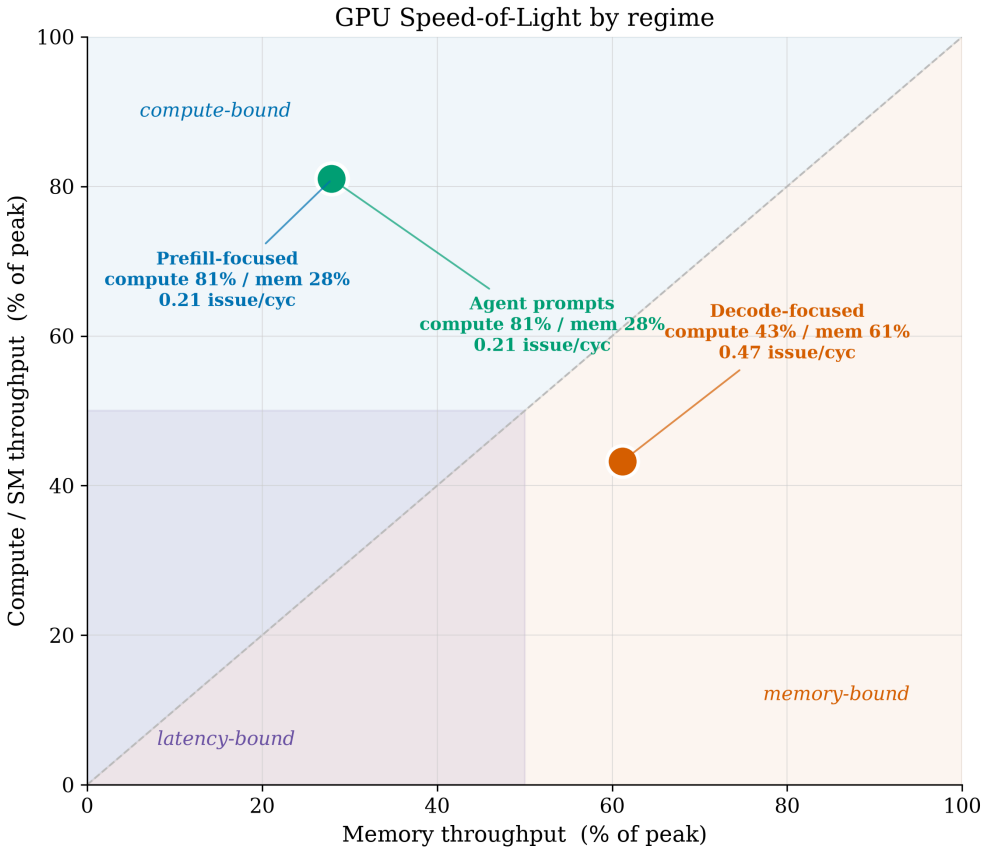


Figure 4.17: GPU Speed-of-Light regimes for prefill, decode, and agent prompts

Figure 4.18 and Figure 4.19 identify where that time is spent and why. Quantized matrix kernels dominate GPU time in every regime, at 79% of prefill-focused GPU time, 90% of decode-focused time, and 79% of the agent-prompt regime, with attention a minority everywhere. The second figure shows the same dominant kernel flipping bottleneck between phases. As a matrix-matrix product in prefill it is compute-bound at 86% SM activity, and as a matrix-vector product in decode it becomes memory-bound at 67%, re-streaming the model weights for every token.

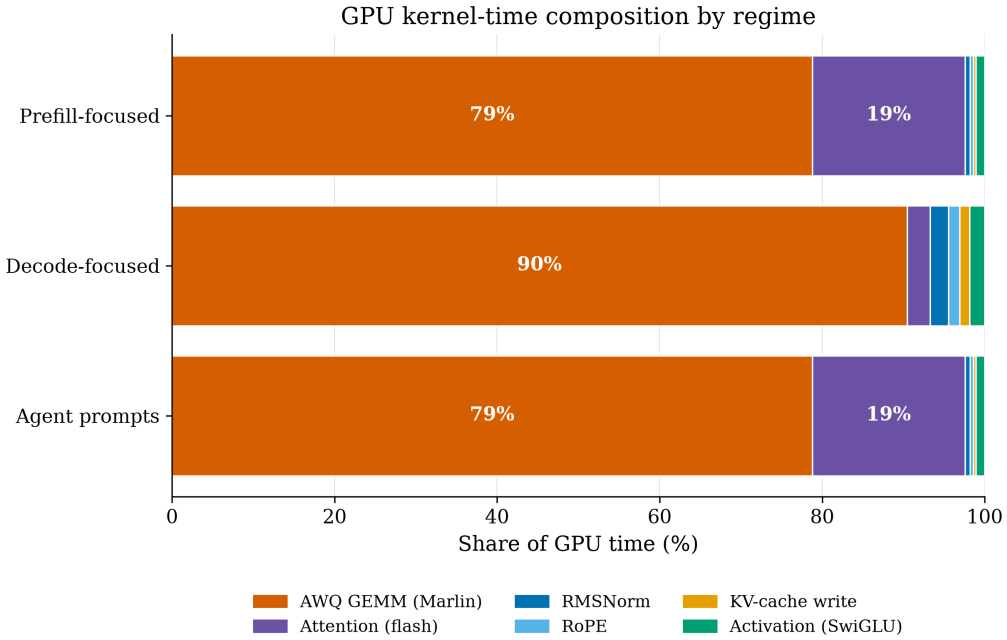


Figure 4.18: GPU time by kernel class per regime. The quantized matrix kernels hold 79 to 90 percent of GPU time in every regime, and attention is a minority everywhere.

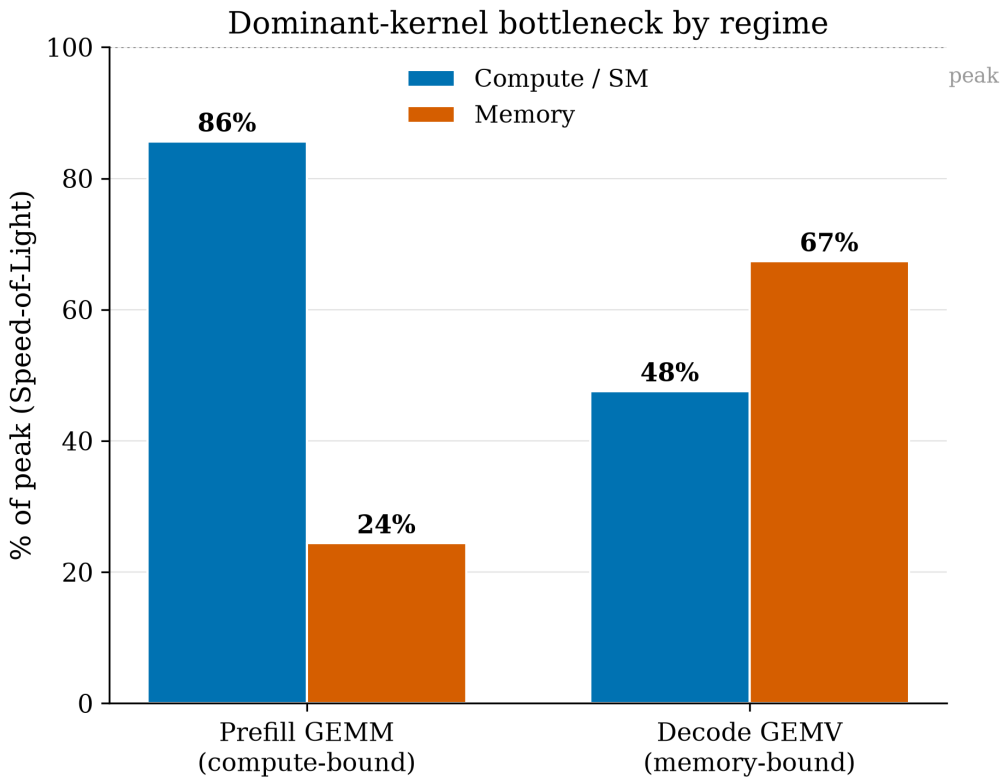


Figure 4.19: The dominant matrix kernel's bottleneck flip: compute-bound as the matrix-matrix product of prefill, memory-bound as the matrix-vector product of decode.

This phase asymmetry is the measured ground for the serving designs that separate prefill from decode [23], [24]: the two phases want different resources, and agent traffic, being prefill-shaped, leans on the compute side of that split.

5

Discussion

5.1 During Inference Versus Outside Inference

The measurements separate two CPU roles:

- **During inference:** the CPU can be busy, high-IPC, and retiring-bound, but mostly waiting on GPU completion.
- **Outside inference:** the CPU is lower-IPC and more irregular, but it performs useful work: retrieval, cache lookup, code execution, tests, browser actions, parsing, and I/O.

This is the main thesis result. During inference, IPC 3.59 and 57% retiring look like healthy CPU execution, but 85% of host samples are in CUDA synchronization. Outside inference, lower IPC is not a failure by itself, because the CPU is doing exact, irregular service and tool work. This distinction changes what should be optimized. The busy inference CPU should not be tuned like useful application code. It should be made to wait differently. The outside-inference CPU should be optimized by path: cache, RAG, storage, parsing, tool execution, and tests.

The result is over-determined on this one machine. The during-inference signature is measured under the full service with full TMA: 57% retiring at IPC 3.59, tier-invariant across all twelve cells, about 1.9 CPUs under load against 0.02 idle, validated by 366 cross-checks. Two lenses agree, since the hardware counters read healthy retiring while the host samples show that the retiring is a CUDA-synchronization busy-wait.

The contribution is the characterization across a deployable service, agent workloads, CPU counters, and GPU profiling, not a new serving mechanism [1], [2].

5.2 Service Implication

For the service, the two-axis separation is the main result. GPU generation dominates elapsed time, at 95 to 99% across the tiers, and scales linearly with the output tier, while the CPU-side cost is set by the input bucket and is dominated by the query embedding. Output length changes scale, not shape, and the pods' microarchitectural signatures are tier-invariant. Fixed output-token tiers keep generation comparable across cells, and the bucket axis isolates the CPU-side stages, so each benchmark axis reads out one engine. The practical takeaway is that the CPU is nearly irrelevant to

service inference cost. The model's dense arithmetic cannot be lifted onto the host, so the improvements that matter come from cheaper models and better accelerators, and the host's own lever is limited to waiting more cheaply.

5.3 Agent Implication

For agents, GPU inference dominates the time split, but CPU tool execution controls the next step. The CPU work is not dense compute. It is exact, irregular, and stateful.

The two agent benchmarks make that claim broader than one task. Repository repair across four repositories in three languages stresses files, builds, tests, and project state. General browser and computer-use tasks stress I/O, document and image handling, and UI-like control. The common structure is serial alternation between model calls and exact tool actions.

The measured shape of that alternation has a provisioning consequence, developed against the utilization literature in Chapter 6. An agent episode averages between 2% and a third of a busy core, under two percent of the measured partition, because the machine spends most of the wall-clock waiting on the model, yet its heavy tool calls fan out to thirteen, sixteen, even the full twenty cores of the partition for fractions of a second. The agent machine is therefore not sized by its average, which is nearly zero, but by its bursts. It wants many cores available rarely and briefly, the opposite of the steady two-core draw the serving engine shows. And where that CPU lands is not a property of the agent but of the task it is given. The same loop is ninety percent tool payload when the project is a C++ build and three-quarters harness when the episode stretches to hundreds of turns of Python, so there is no single agent CPU budget to provision, only a range set by what the tools do and how long the conversation runs.

5.4 The Harness as an Acceleration Target

Separating the agent harness from the tool payload with the lineage fence makes the harness a measurable object rather than an implementation detail, and its measured character makes it the natural next target for accelerating agentic systems. The harness is the orchestration substrate of the loop. It assembles each prompt, parses each model output, dispatches each tool call, and folds the result back into the running history. It fires once between every model turn, hundreds of times per episode, and it sits on the serial critical path, model then harness then tool then harness then model.

Its microarchitecture says what kind of work this is. The Python harness is retiring-bound at IPC 2.8, served mostly from the micro-op cache, and pinned near a single core by the global interpreter lock. The Node.js harness retires under 30 percent of its slots at a lower IPC, a JIT runtime rather than an interpreter loop. Either way it is scalar, branchy, single-thread-bound orchestration, not dense compute, and it does

almost none of the task's real work. In absolute core-seconds it is small today, but it is the loop's serial term. As served models get faster and tool calls get cheaper, a fixed per-turn orchestration cost becomes a larger share of the whole, and on the longest episode its active time already reached about 15 percent of wall-clock across hundreds of turns.

The harness cost also compounds. Across fourteen live episodes on four repositories, spanning codebases from 68 thousand to 531 thousand lines, the harness's CPU work is set almost entirely by conversation length. It grows as roughly the 2.7th power of the number of turns, with the fit explaining 99.8 percent of the variance (Figure 5.1), and codebase size contributes almost nothing, two repositories of the same size differing by 21 times the harness work at 3.1 times the turns. The mechanism is the loop's own bookkeeping. Every turn the harness re-processes a history that every previous turn has grown, so the per-turn cost rises as the episode lengthens and the total compounds superlinearly. This is why the longest episode is harness-dominated. The payload cost is bounded by the task, but the orchestration cost is bounded only by how long the agent keeps talking. For long-horizon agents this term, small in any single turn, becomes the machine's main occupation.

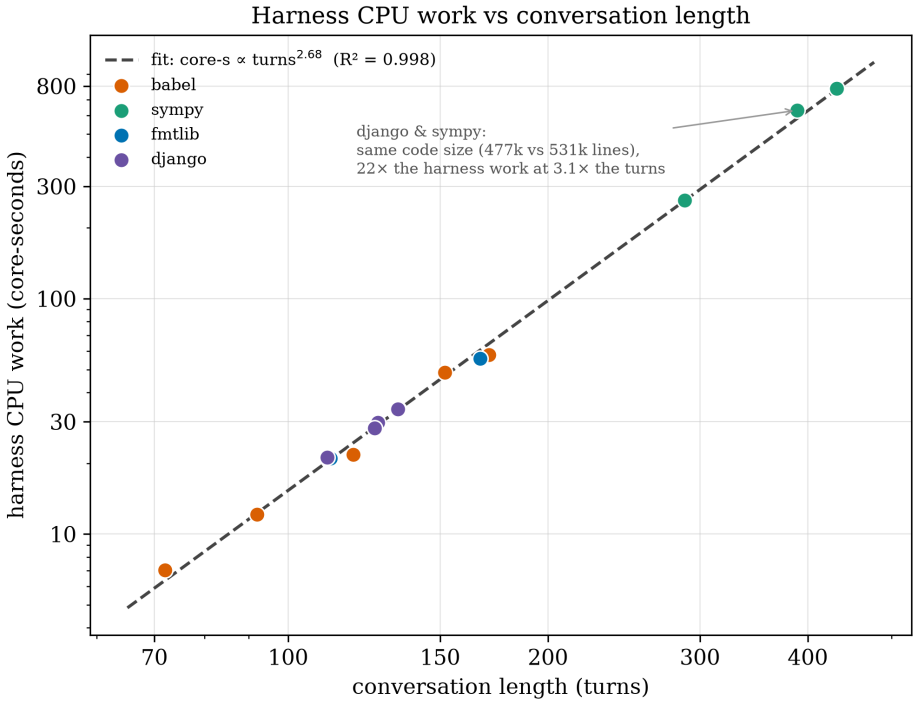


Figure 5.1: Harness CPU work against conversation length across fourteen live episodes on four repositories (log-log). The power-law fit has exponent 2.7 and explains 99.8 percent of the variance. Repository size, spanning 68 to 531 thousand lines, does not set the cost.

This frames two acceleration directions, and both require the measurement first. Algorithmically, the harness serializes the model and the tool, so the levers are to cut turn count, to assemble and parse incrementally rather than rebuild whole prompts,

to cache harness and tool state across turns, and to overlap the harness bookkeeping that does not depend on the current tool result, such as history compaction and template rendering, into the decode window when the host CPU is otherwise idle. For hardware, an agentic server is not a peak-matmul problem on the host. The harness wants strong single-thread performance, good branch prediction, a large micro-op cache, and a low-latency handoff to the accelerator, and because it is interpreter-lock-bound, scaling many concurrent agents needs many independent scalar contexts rather than a bigger GPU. Co-designing for the agentic loop therefore means optimizing the orchestration substrate and the synchronization and handoff path, not adding floating-point throughput.

The engine busy-wait and the harness are the same kind of cost seen at two fences: CPU-side orchestration on the critical path around a GPU that does the dense work. Characterizing them, which is what the lineage-fenced top-down method does, is the precondition for accelerating them, in the algorithm or in the hardware.

5.5 The Orchestration Layer Across Agents

The harness is not one fixed thing. This work measures two, and they already differ in the two ways that matter for the CPU: how the loop is *built* and how it is *structured*. Both shape the orchestration workload, and that workload, not the model arithmetic that runs on the accelerator, is what a CPU-side accelerator for agentic systems would have to target.

How the loop advances. An agent step is not a fixed computation. It is whatever the model samples next. With greedy decoding (temperature zero) the loop becomes deterministic, and it can then lock into a degenerate cycle: once the most-likely next action is to repeat the previous one, the agent repeats it without end. This is a failure mode the runs here had to guard against, with a nonzero sampling temperature and a loop check in the harness. The consequence for the CPU is that the orchestration workload is emergent and data-dependent, not fixed by the task. The number of turns, whether the loop stalls, and when it stops all fall out of the sampling, so the harness cost varies from run to run rather than being a constant that can be provisioned in advance.

Built differently. The two harnesses here are written in different languages, and the microarchitecture reflects it directly. SWE-agent is Python. Its loop is held to a single core by the global interpreter lock and served mostly from the micro-op cache, the signature of a serial interpreter. OpenClaw is Node.js, a just-in-time-compiled runtime with an event loop and worker threads, retiring-poor and bursting past one core. These are measured, not assumed. The implication carries to harnesses this work did not measure. A TypeScript agent compiles down to the same Node/V8 runtime as OpenClaw and would inherit its JIT, event-loop character, while a Go or Rust harness would look different again. The orchestration substrate therefore has no single CPU profile. It is whatever runtime the harness is written in, and that choice sets the single-thread-versus-concurrent and interpreter-versus-JIT character of the load.

Written at a different level tomorrow. The pattern in these two harnesses also matches where agent frameworks sit in a familiar technology cycle: new systems are first written in the languages that are fastest to build in, and today's harnesses are Python and JavaScript. If agents follow the path that performance-critical infrastructure has taken before, from prototype-friendly languages toward compiled, lower-level implementations once the designs stabilize, the orchestration workload will change character again. A compiled harness would lift the single-core interpreter ceiling measured here, shrink the fixed per-turn cost, and move the profile toward the front-end-bound server character of the tool payloads. What it would not change by itself is the superlinear growth with conversation length, because that cost is algorithmic, the re-processing of an ever-growing history, not a property of the implementation language. The measured contrast between the interpreter loop and the JIT runtime is the evidence that this migration matters: the harness's microarchitectural character, and therefore the right CPU for it, follows the implementation choice.

The layer applies to the payload too. A delegated task does not run on the hardware directly. It runs through the language it is written in, and that layer is itself a computation, which is why the tool workloads here span languages. The C++ fence executes native compiler code that saturates the partition, the Python fences spend their cycles inside the interpreter and never leave scalar code, and the JavaScript fence runs a runtime that compiles itself while it works. The cost of the layer is documented: interpreted execution of scalar code commonly costs one to two orders of magnitude in time and energy against compiled implementations [40], and recovering that loss is argued to be a main source of performance after Moore's law [41]. The layer can also be bypassed from within a high-level language, as in the document and image tasks, where Python only orchestrates compiled vector kernels. An agent choosing tools implicitly chooses this layer with every call.

Structured differently. Both agents here are single-agent serial loops: one model call, one action, one observation, repeated. Other designs are not. A multi-agent or hierarchical harness spawns sub-agents, often a planner on a strong model delegating to workers on lighter models, and runs several of them at once. That changes the CPU picture in kind, not degree. Instead of one serial orchestration thread waiting on a single model round-trip, there are many concurrent orchestration contexts, a scheduler coordinating them, and possibly several models of different sizes in flight together. The serial term identified above becomes a parallel coordination problem, and the host-side work grows from between-turn bookkeeping into genuine scheduling.

This is why the orchestration layer, rather than the model kernel, is the workload that decides CPU acceleration in agentic systems. The model runs on the accelerator and is largely fixed. The orchestration around it is where the host CPU actually works, and its character is set by choices the harness makes, the language it is written in and whether it is one agent or many. There is thus no single agentic CPU workload to optimize, but a family of orchestration runtimes, single-thread interpreter loops, JIT event loops, and multi-agent schedulers, each with a different profile and each needing a different treatment. Characterizing that layer, which this thesis does for two points in the family, is the precondition for accelerating any of it.

5.6 Batchability and Exactness

The measured machine is heterogeneous, because CPU and GPU have different cost operators. The GPU is an amortization engine for dense, regular, batchable work. The CPU is a latency/control engine for serial, irregular, branchy, exact work.

The useful axis is *Batchability* $\times$ *Exactness*. Dense approximate reasoning and token generation are batchable enough to amortize GPU execution. Retrieval, cache validation, file edits, tests, browser actions, and process control are exact and dependency-heavy, so they fit the CPU better. Agent workloads often sit in the worst cell: low-batch exact control is routed through the autoregressive GPU path, then the CPU waits for the next tool action.

5.7 GPU Top-Down Interpretation

The GPU breakdown is two-level because a single aggregate TMA would hide the real bottleneck. Kernel-time attribution answers where time goes. Warp-state and Speed-of-Light counters answer why the dominant kernels are slow. The warp-state view is duration-weighted by kernel time, so short kernels do not dominate the aggregate.

This is the GPU analogue of a CPU top-down drill-down, not a direct copy of Intel TMA. Issued warp slots are retiring-like, stall reasons are front-end-like or back-end-like, and bad speculation is not applicable. The Intel-style re-binning is useful because it lets the CPU and GPU be read in one language. Occupancy and lane-utilization counters help separate back-end and issue pressure from divergence. The conclusion is that GPU time is mostly dominated by matrix kernels, while the limiting resource changes by phase: tensor and math pressure in prefill, memory and back-end pressure in decode.

5.8 Limits

The main threat to validity is workload scale. Most measurements are batch-1 or single-stream, so production batching may change both GPU efficiency and host waiting behavior. The agent sample is modest and intentionally diverse rather than statistically exhaustive, one long live episode per task, and the live browser tasks are the least repeatable.

Measurement scope is also a limit. Everything comes from one workstation with a small GPU, a 7B served model, and frontier-API-driven agents, which is what makes the single-lens consistency possible.

6

Conclusion

This thesis asked where the time goes in a deployed LLM system, and the answer splits at the inference boundary. In the service, GPU generation holds 95 to 99 percent of request time, and the largest CPU consumer, the engine's roughly two cores during inference, is a synchronization busy-wait that a blocking wait policy reclaims (1.00 core down to 0.24) at unchanged throughput. In the agents the roles invert: the model round-trip dominates the clock, but the CPU does the task's real work outside inference, its character set by the delegated payload. On the GPU, quantized matrix kernels dominate every regime, compute-bound in prefill and memory-bound in decode.

Against earlier research the judgments are clear: the delegated tool work matches the front-end-bound, low-IPC profile of the warehouse-scale studies, so the agentic CPU is a familiar server workload that existing instruction-supply research addresses [1], [2]. The serving literature's GPU-first priority is confirmed, but the host's two busy-waiting cores are a cost it overlooks, recoverable in the runtime [14], [23], [24]. Sharpest is the utilization result: datacenter studies place production servers between roughly 10 and 50 percent of maximum utilization, with cluster aggregates reported below 20 percent [3], [4], [5]. The agent machine averages 0.3 to 1.5 percent of its partition while briefly demanding all of it. Provisioning by peak leaves the machine idle and by average starves critical-path bursts, so the energy-proportionality consolidation argument applies with extra force: agent workloads should share machines [3].

Genuinely new against that literature is the harness term, that can be separated from the tool execution in agentic workloads. Its cost grows as roughly the 2.7th power of conversation length, indifferent to codebase size, so in long episodes the orchestration loop, not the tool payload, becomes the machine's main occupation. It is improvable today through incremental history handling, prompt reuse, and overlap into the idle decode window, and it defines the agent-side CPU, strong single-thread performance and a fast accelerator handoff, not peak matrix throughput.

Future work should repeat the study on larger GPUs and production batching, widen the agent sample, measure multi-agent harnesses, track harnesses as they migrate from rapid-development toward compiled languages, and turn these captures into traces that drive a cycle-accurate simulator, which is what would let the design questions this characterization raises be tested on hardware that does not yet exist.

The broader lesson matches the studies that inspired this work, the important bottleneck is not always inside the most obvious kernel [1], [2] measuring it at the right fences makes those costs actionable.

Bibliography

[1] M. Ferdman et al., "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, ACM, 2012, pp. 37–48. DOI: 10.1145/2150976.2150982

[2] S. Kanev et al., "Profiling a Warehouse-Scale Computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, Also available via IEEE Xplore document 7284063, ACM, 2015, pp. 158–169. DOI: 10.1145/2749469.2750392

[3] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[4] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC)*, ACM, 2012.

[5] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, 2014.

[6] M. Tirmazi et al., "Borg: The next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, ACM, 2020.

[7] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software*, 2014, pp. 35–44. DOI: 10.1109/ISPASS.2014.6844459

[8] NVIDIA, *NVIDIA Nsight Compute Profiling Guide*, Accessed 2026-06-29, 2026. [Online]. Available: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html

[9] NVIDIA, *NVIDIA Nsight Systems User Guide*, 2026. Accessed: Jun. 29, 2026. [Online]. Available: https://docs.nvidia.com/nsight-systems/UserGuide/index.html

[10] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459–9474.

[11] R. Friel, M. Belyi, and A. Sanyal, *RAGBench: Explainable Benchmark for Retrieval-Augmented Generation Systems*, 2025. DOI: 10.48550/arXiv.2407.11005

[12] W. Gill, M. Elidrisi, P. Kalapatapu, A. Ahmed, A. Anwar, and M. A. Gulzar, "MeanCache: User-Centric Semantic Caching for LLM Web Services," in *2025 IEEE International Parallel and Distributed Processing Symposium*, 2025, pp. 1298–1310. [Online]. Available: https://arxiv.org/abs/2403.02694

[13] J. Kim et al., *Rethinking Caching for LLM Serving Systems: Beyond Traditional Heuristics*, 2025. DOI: 10.48550/arXiv.2508.18736

[14] W. Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2023. DOI: 10.1145/3600006.3613165

[15] S. Ramírez, *FastAPI*, https://fastapi.tiangolo.com, Accessed 2026, 2018.

[16] S. Xiao, Z. Liu, P. Zhang, and N. Muennighoff, *C-pack: Packed resources for general chinese embeddings*, 2023. arXiv: 2309.07597 [cs.CL].

[17] J. Wang et al., "Milvus: A purpose-built vector data management system," in *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, ACM, 2021, pp. 2614–2627.

[18] C. L. Chen, *SeaweedFS: A distributed object store and file system*, https://github.com/seaweedfs/seaweedfs, Accessed 2026, 2015.

[19] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX Association, 2010, pp. 47–60.

[20] MongoDB, Inc., *MongoDB*, https://www.mongodb.com, Accessed 2026, 2009.

[21] llm-d Project, *Llm-d: Kubernetes-native distributed inference serving*, https://llm-d.ai, Accessed 2026, 2025.

[22] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, pp. 70–93, 2016.

[23] Y. Zhong et al., *DistServe: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving*, 2024. DOI: 10.48550/arXiv.2401.09670

[24] A. Agrawal et al., *Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve*, 2024. DOI: 10.48550/arXiv.2403.02310

[25] M. Arif, A. Maurya, S. Vazhkudai, and B. Nicolae, *Understanding Inference Scaling for LLMs: Bottlenecks, Trade-offs, and Performance Principles*, 2026. DOI: 10.48550/arXiv.2605.19775

[26] J. Lin et al., *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*, 2023. DOI: 10.48550/arXiv.2306.00978

[27] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*, 2022. DOI: 10.48550/arXiv.2208.07339

[28] Intel Corporation. "Cpu metrics reference – intel vtune profiler." Accessed: 2026. [Online]. Available: https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/current/cpu-metrics-reference.html

[29] Intel Corporation, *Intel 64 and ia-32 architectures optimization reference manual*, Accessed: 2026, Intel Corporation. [Online]. Available: https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[30] S. Yao et al., *ReAct: Synergizing Reasoning and Acting in Language Models*, 2023. DOI: 10.48550/arXiv.2210.03629

[31] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, *Reflexion: Language Agents with Verbal Reinforcement Learning*, 2023. DOI: 10.48550/arXiv.2303.11366

[32] J. Yang et al., *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*, 2024. DOI: 10.48550/arXiv.2405.15793

[33] Linux man-pages project, *perf-stat(1): Linux Manual Page*, 2026. Accessed: Jun. 29, 2026. [Online]. Available: https://man7.org/linux/man-pages/man1/perf-stat.1.html

[34] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann, 2017.

[35] A. Kleen, *pmu-tools: Intel PMU Profiling Tools*, 2026. Accessed: Jun. 29, 2026. [Online]. Available: https://github.com/andikleen/pmu-tools

[36] C. E. Jimenez et al., *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* 2024. DOI: 10.48550/arXiv.2310.06770

[37] X. Liu et al., *AgentBench: Evaluating LLMs as Agents*, 2023. DOI: 10.48550/arXiv.2308.03688

[38] NVIDIA, *CUDA Programming Guide*, 2026. Accessed: Jun. 29, 2026. [Online]. Available: https://docs.nvidia.com/cuda/cuda-programming-guide/index.html

[39] NVIDIA, *CUDA Runtime API: Event Management*, 2026. Accessed: Jun. 29, 2026. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html

[40] R. Pereira et al., "Energy efficiency across programming languages: How do energy, time, and memory relate?" In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, ACM, 2017.

[41] C. E. Leiserson et al., "There's plenty of room at the top: What will drive computer performance after Moore's law?" *Science*, vol. 368, no. 6495, eaam9744, 2020.

A

Supplementary Service Figures

These figures supplement the service results in section 4.1, all from the same isolated capture.

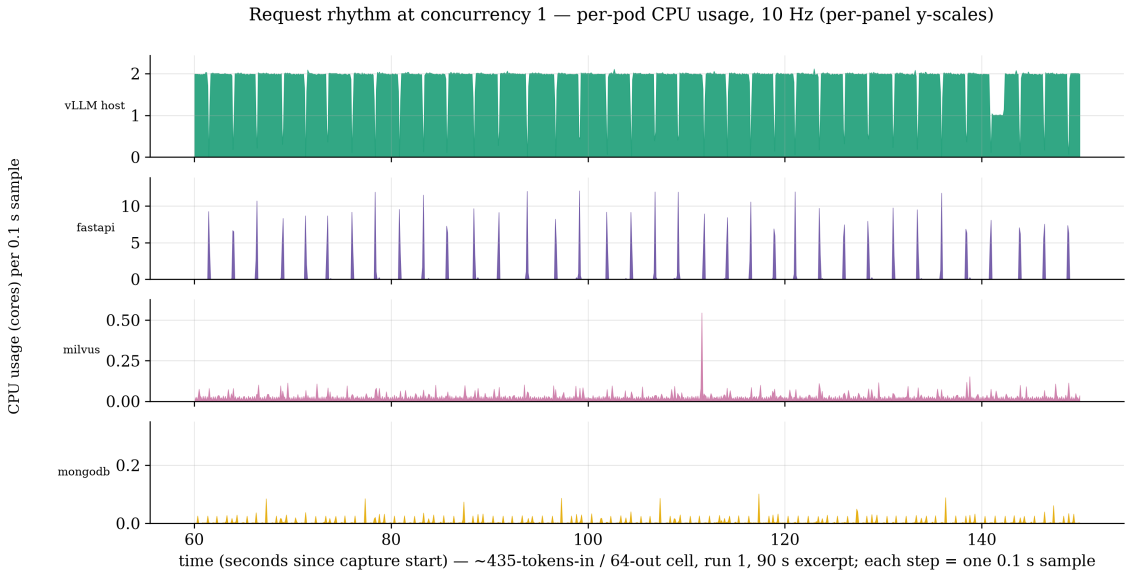


Figure A.1: Per-pod CPU usage over the capture window. The engine holds its two-core spin continuously while the CPU-side RAG stages fire per request.

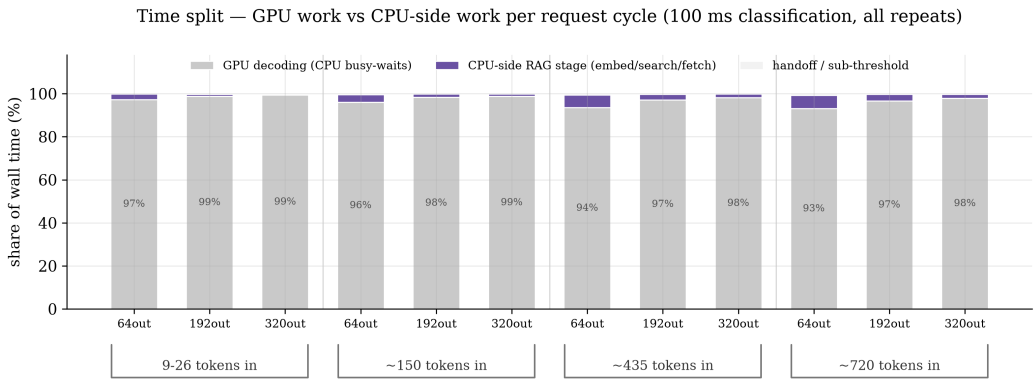


Figure A.2: Wall-clock split between GPU generation and CPU-side work across the twelve cells (isolated capture). This is the tier-donut data in per-cell bar form.

B

The Browser Agent: OpenClaw

OpenClaw is the browser and computer-use agent (WildClawBench), driven by GLM-5.2 and captured in the same windows as the main-text SWE-agent episodes. Its harness is a Node.js runtime rather than a Python loop. It also sits differently on the machine. Where SWE-agent is a host process that delegates each tool action to a separate sandbox container, giving a natural container boundary to fence on, OpenClaw runs as one container holding both the agent gateway and every tool it spawns. The harness/tool split therefore cannot follow a container boundary at all. It is drawn purely by process lineage, each descendant tagged by which process spawned it, which makes this the most delicate fencing in the study. Four tasks, all resolved: assembling an image jigsaw, digesting a long PDF, digesting a set of web pages, and crawling a wiki. All four payloads execute Python inside the tool fence.

Figure B.1 and Figure B.2 show the same two accountings as the main text. The episodes are even more inference-dominated than the code agents. The model round-trip is 91 to 95 percent of wall-clock, and the CPU totals are tiny, 19 to 53 core-seconds per episode, an average of 0.02 to 0.04 busy cores. Where that work lands depends on what the task actually computes. The jigsaw and PDF tasks do real local processing, so their tool payloads dominate (76 and 62 percent of core-seconds). The two web-reading tasks compute almost nothing locally, their tool side only fetches pages, so the largest CPU consumer becomes the local API proxy that streams the model's long responses (57 and 61 percent). When the payload is trivial, the loop's own plumbing is what is left. Figure B.4 and Figure B.5 trace both fences over episode time, instantaneous and accumulated, read exactly as the main-text views.

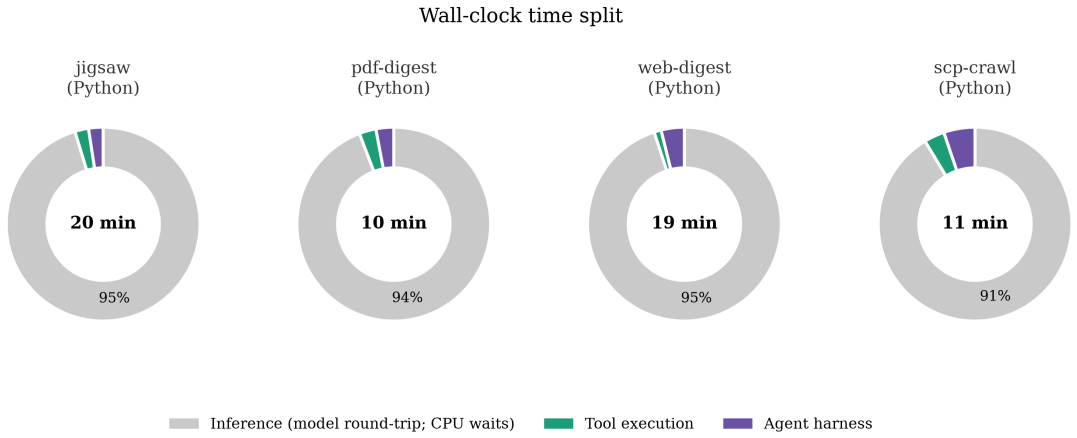
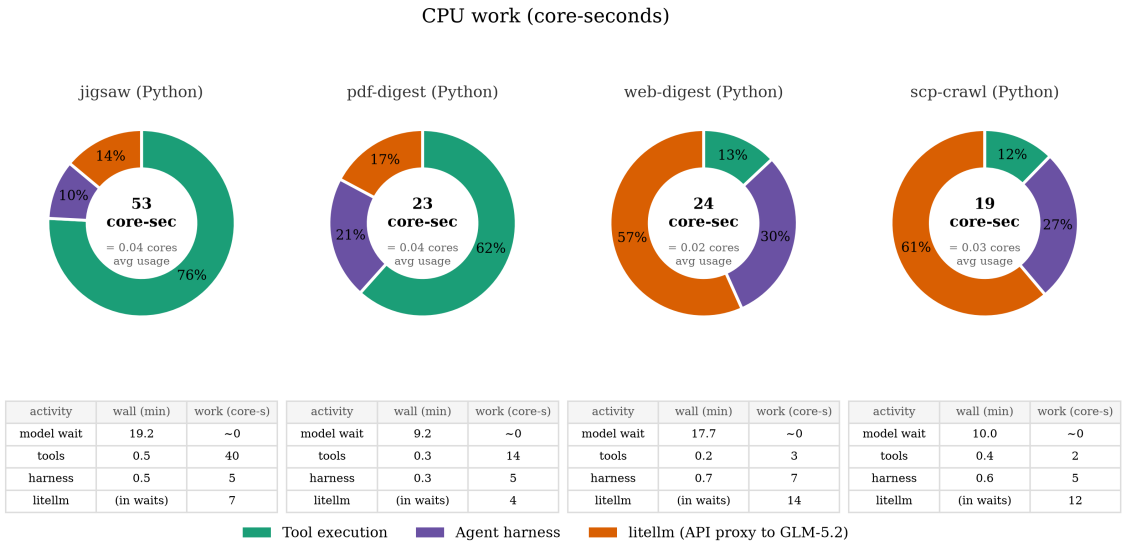


Figure B.1: Where each OpenClaw episode's wall-clock time went: model round-trip, tool execution, and the agent harness.



activity	wall (min)	work (core-s)
model wait	19.2	~0
tools	0.5	40
harness	0.5	5
litellm	(in waits)	7

activity	wall (min)	work (core-s)
model wait	9.2	~0
tools	0.3	14
harness	0.3	5
litellm	(in waits)	4

activity	wall (min)	work (core-s)
model wait	17.7	~0
tools	0.2	3
harness	0.7	7
litellm	(in waits)	14

activity	wall (min)	work (core-s)
model wait	10.0	~0
tools	0.4	2
harness	0.6	5
litellm	(in waits)	12

Figure B.2: Who consumed each OpenClaw episode's CPU core-seconds. The compute-carrying tasks (jigsaw, PDF) are tool-dominated. On the fetch-only web tasks the API proxy is the largest consumer.

Figure B.3 averages the same capture over each full episode: 0.02 to 0.04 cores, which is 0.1 to 0.2 percent of the 20-core measured partition, an even more extreme version of the utilization picture in the main text. On the two web-reading tasks the proxy stack dominates the average, matching the core-second split above.

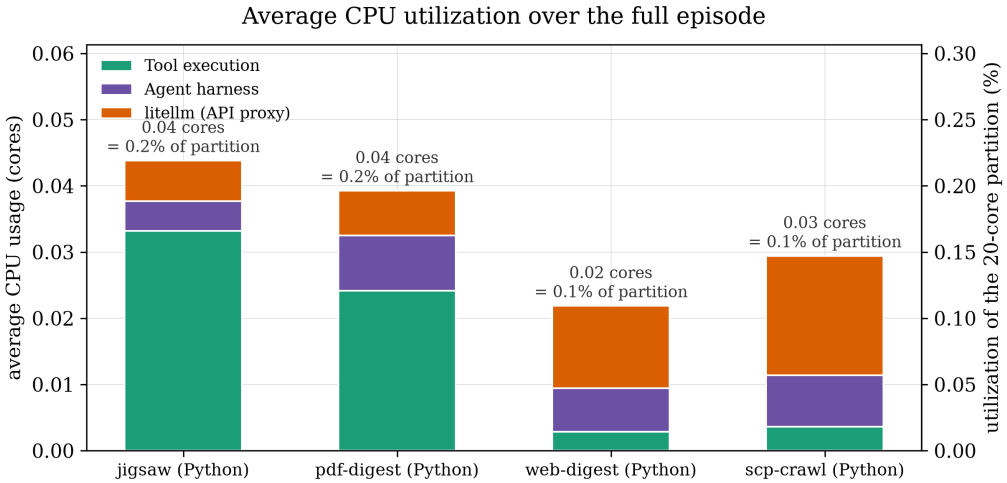


Figure B.3: Average CPU utilization over the full OpenClaw episode, stacked by fence, in cores and as a share of the 20-core measured partition. Every episode averages at or below about 0.2 percent of the partition.

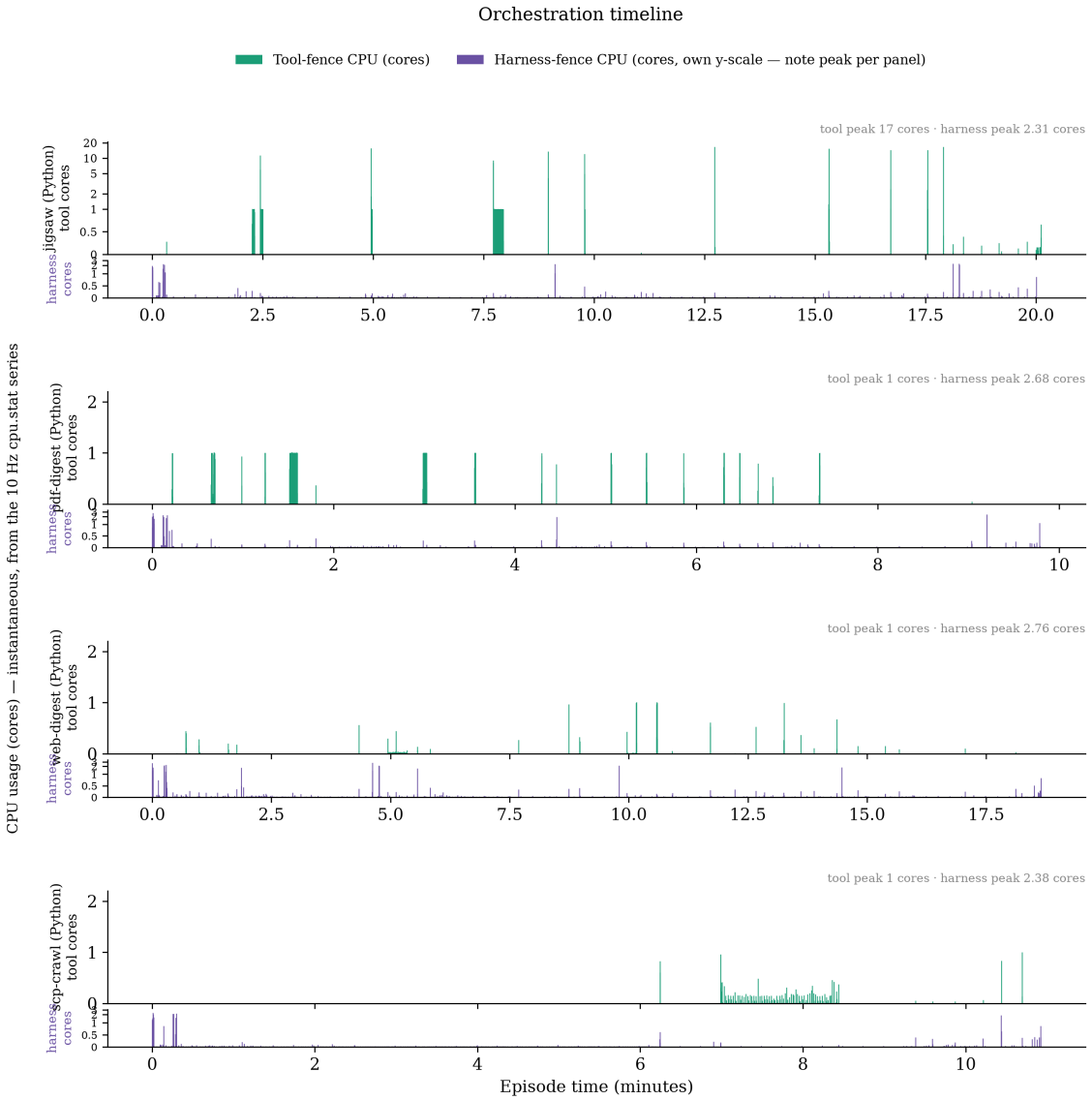


Figure B.4: Orchestration timeline per OpenClaw episode: instantaneous CPU cores in the tool and harness fences. As in the main text, white gaps are model round-trips.

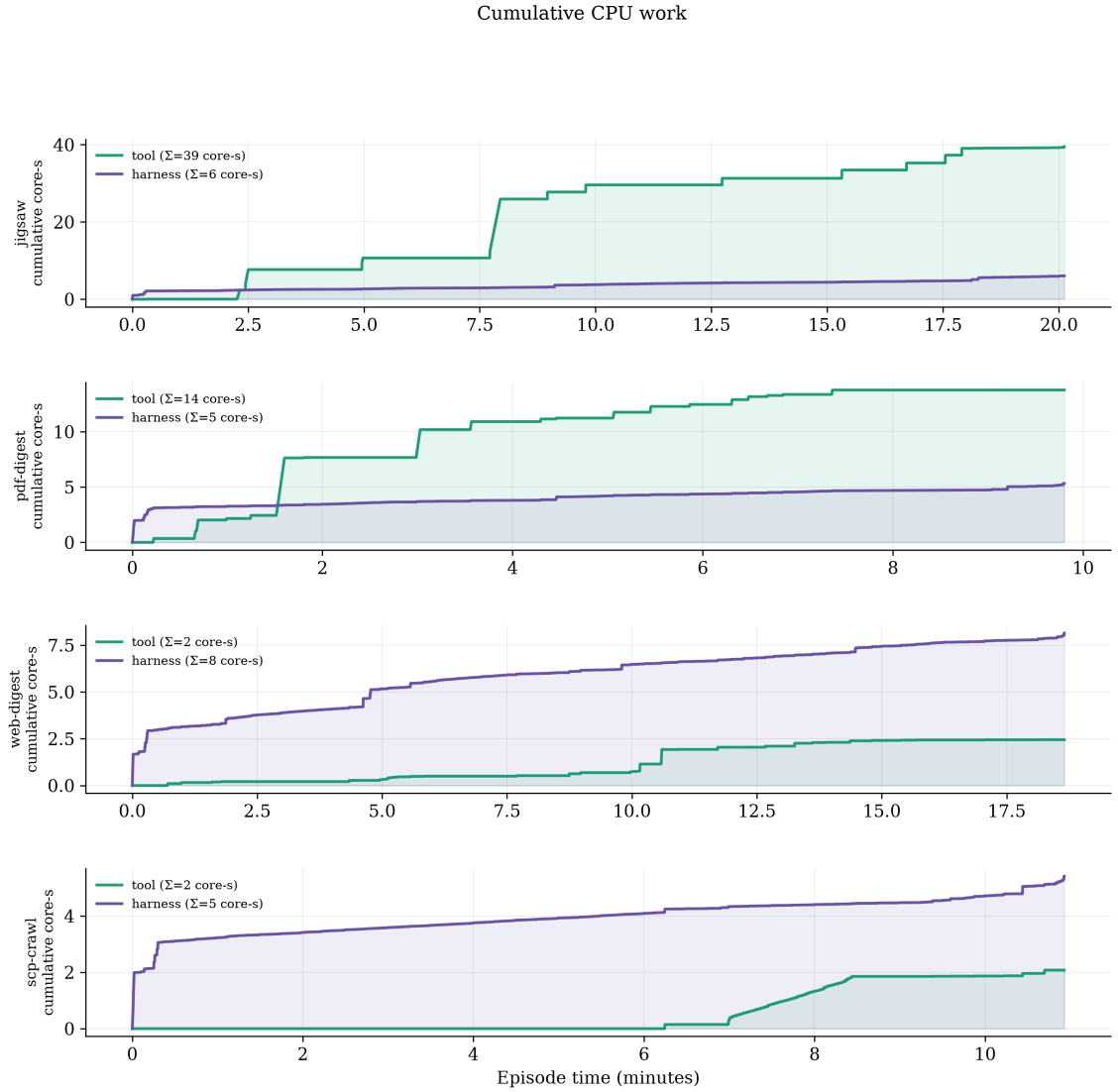


Figure B.5: Cumulative CPU work per fence for the OpenClaw episodes, read as in the main text: flat stretches are model round-trips, and the slope of a rising curve is that fence's instantaneous usage in cores.

The episodes issue far fewer actions than the code agents, 17 to 49 tool calls each (Figure B.6). Only the jigsaw task fans out, peaking at 17 cores in a vectorized image-processing burst visible in its timeline, while the other payloads stay near one core.

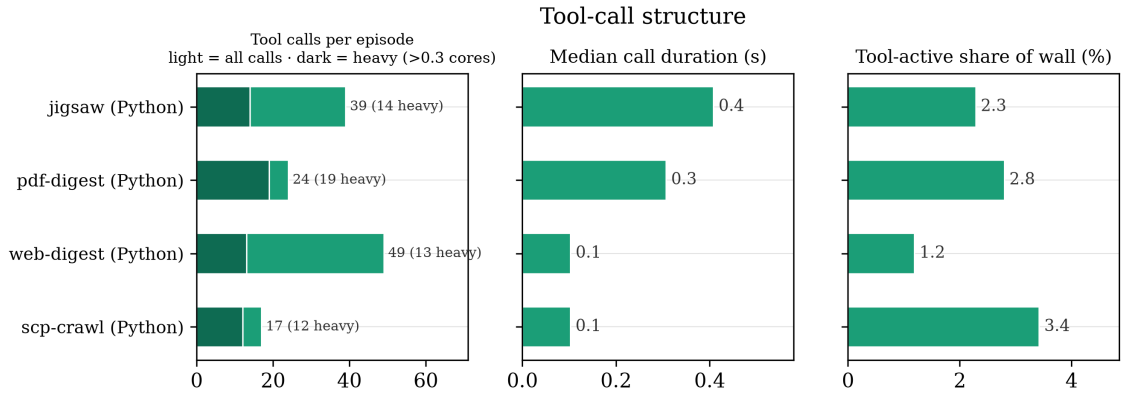


Figure B.6: Tool-call structure per OpenClaw episode: call counts, median duration, and tool-active share of wall-clock.

The counter view (Figure B.7) is where OpenClaw earns its place in the study. The floating-point counters make this the one corner where a tool payload reaches dense numeric compute. The PDF task's tool side is about half packed floating point and the jigsaw's about 15 percent, image decoding and array processing, while the two web tasks carry none, matching the SWE payloads. The Node.js harness also breaks the pattern of its Python counterpart. Its instructions per cycle sit at 1.4 to 1.6 rather than 2.8, and its instruction-cache miss rate runs an order of magnitude higher, the footprint of a large just-in-time-compiled runtime rather than a small interpreter loop.

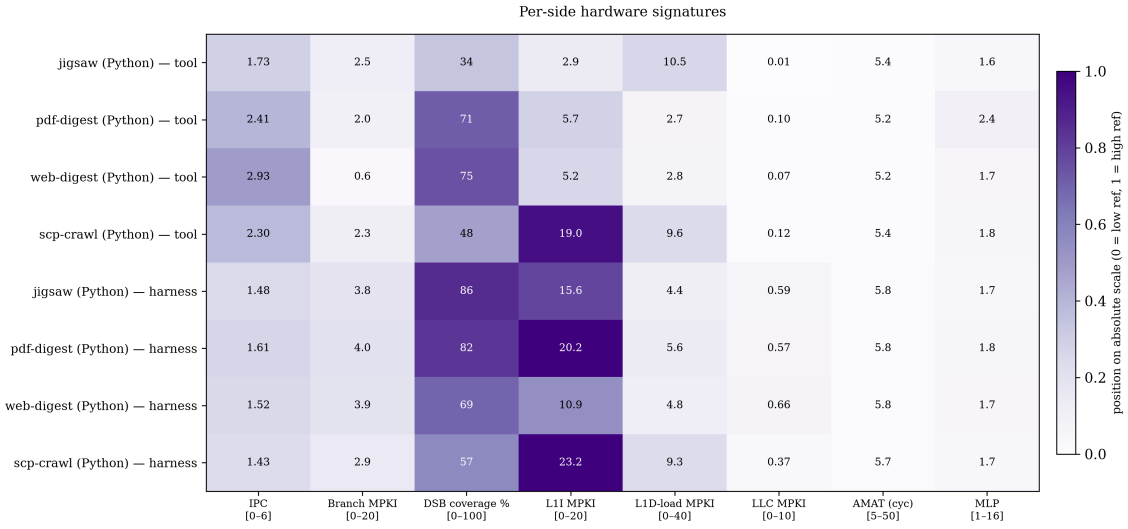


Figure B.7: Per-side hardware signatures for OpenClaw, read as in the main text: the tool payloads against the Node.js harness on absolute scales.

The top-down view (Figure B.8) confirms the inversion. The Node.js harness retires only 27 to 29 percent of its pipeline slots, losing the rest across the front end, back end, and real bad speculation, unlike the retiring-dominated Python harness of the

main text. The tool sides, lighter and partly vectorized, retire 38 to 46 percent. Figure B.9 drills the same accounting to the measured causes, read exactly as the main-text tree.

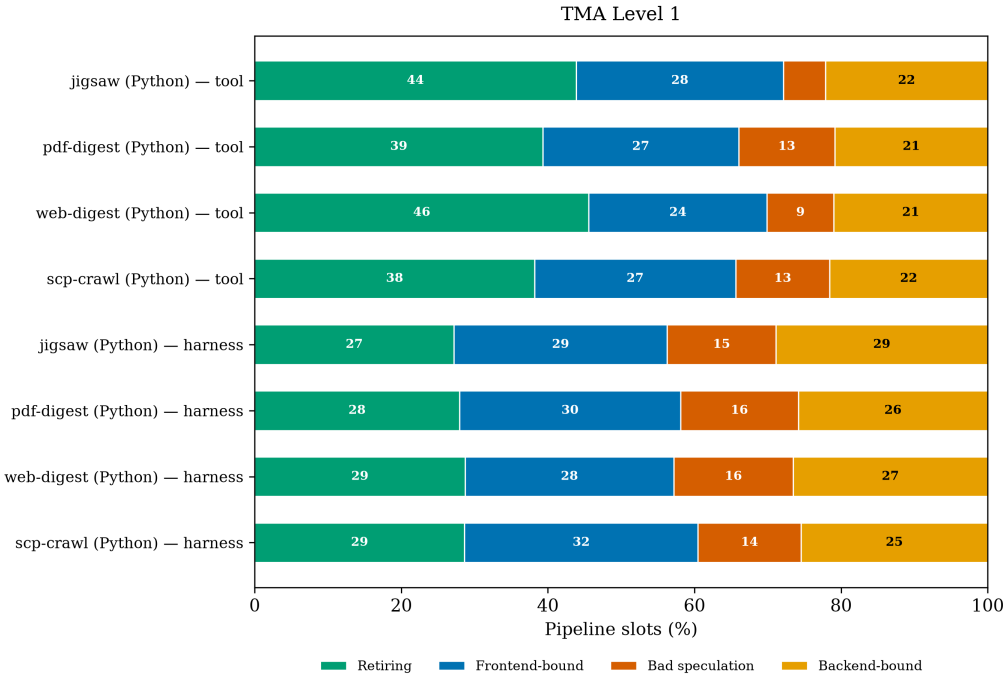


Figure B.8: Top-down Level-1 pipeline slots per side for OpenClaw. The Node.js harness is not retiring-dominated, unlike the Python harness of the code agents.

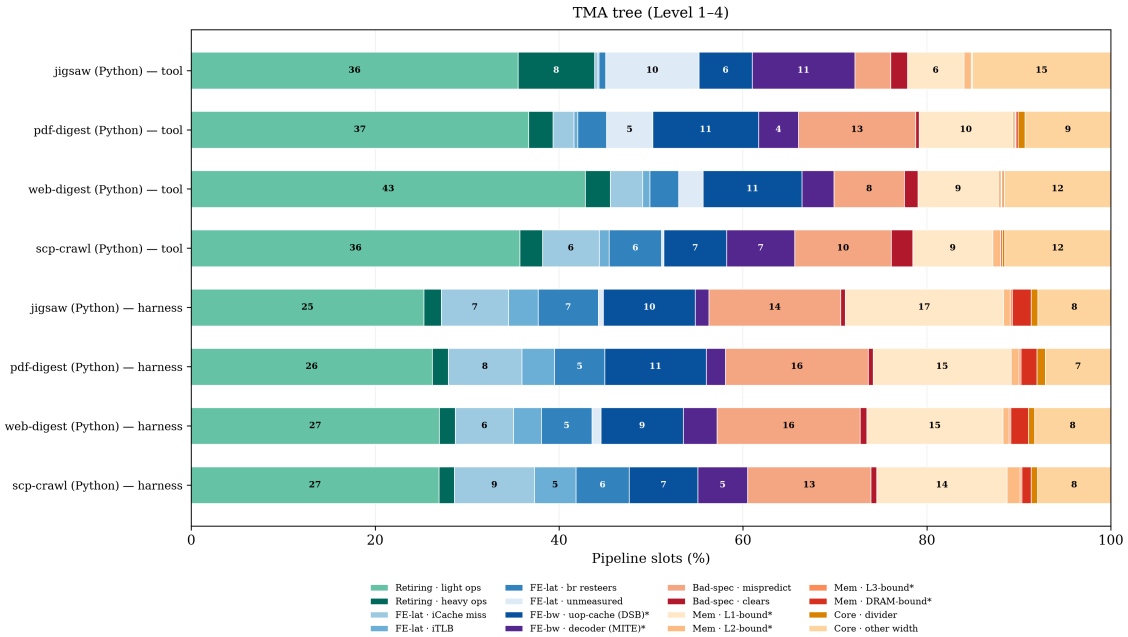


Figure B.9: The full top-down tree per fence for OpenClaw, as shares of pipeline slots. Starred segments are labeled estimates, and the pale segment is unattributed fetch-latency cost.