Hydra: Phase-Aware Workload Characterization of LLM Inference across Edge SoC Generations, Backends, and Quantization Levels

Amir Taherin*, Sana Taghipour Anvari*, Charles Amante*, Yixiao Chen*, Ruben Noroian*, Zlatan Feric*,
Nicolas Bohm Agostini*, Pu Zhao*, José Cano†, Bin Ren‡, Yanzhi Wang*, David Kaeli*
*Northeastern University, Boston, MA, USA †University of Glasgow, UK ‡College of William & Mary, VA, USA
{taherin.a, taghipouranvari.s, amante.c, chen.yixia, noroian.r, feric.z,
bohmagostini.n, p.zhao, yanzhiwang, d.kaeli}@northeastern.edu
Jose.CanoReyes@glasgow.ac.uk, bren@cs.wm.edu

***Abstract*—Edge LLM deployment is shaped by more than model size and precision: inference backend, hardware platform, memory traffic, and power management all affect latency and efficiency. We present *Hydra*, a common-schema, phase-aware workload characterization framework for LLM inference on edge SoCs. Hydra instruments HuggingFace Transformers and `llama.cpp` with a shared per-prompt timing schema and fuses those records with hardware telemetry, enabling a multi-dimensional characterization of performance, system-resource utilization, and efficiency across prefill and decode phases. Using Hydra, we evaluate three consecutive edge System-on-Chip (SoC) generations (AGX Xavier, AGX Orin, and AGX Thor), 13 instruction-tuned LLMs from seven families, five execution formats, and consider input/output-length sensitivity. The resulting artifact contains roughly 107K per-prompt records and is publicly released with Hydra. Our analysis shows that aggregate latency alone hides key deployment effects: backend structure changes where latency is introduced, quantization reduces memory traffic and energy but does not predict power monotonically, and SoC generation changes how utilization and efficiency should be interpreted. By connecting phase-level timing with system-resource utilization and efficiency metrics, Hydra enables reproducible, phase-aware characterization of edge LLM inference. Hydra's source code and the collected per-prompt trace corpus are available open-source at: https://github.com/amirtaherin/hydra.**

***Index Terms*—Edge Computing, LLM Inference, Quantization.**

I. Introduction

Large language models (LLMs) have transitioned from research artifacts to production-critical infrastructure, powering code generation [1], machine translation [2], conversational agents [3]–[5], and a growing set of multi-modal applications [6] served at scale by modern inference stacks [7]–[10]. The dominant deployment standard remains datacenter serving: model weights, a key–value (KV) cache, and per-token computation reside on discrete server-class accelerators. These systems utilize high-bandwidth memories (HBM) and abundant power and thermal headroom. Yet a growing class of latency-sensitive and privacy-sensitive applications such as robotics [11]–[13], autonomous driving [14], [15], healthcare monitoring [16], smart homes and on-device agents [17] cannot always tolerate the round-trip latency, privacy exposure, and connectivity dependence of cloud LLM serving [18], [19]. The natural alternative is to deploy smaller LLMs onto edge System-on-Chip (SoC) platforms that integrate CPU and GPU with shared unified memory.

Edge SoCs are not miniaturized datacenters. We study three consecutive edge-SoC generations in the NVIDIA Jetson lineage: AGX Xavier [20], AGX Orin [21], and AGX Thor [22]. Together, these platforms span Volta, Ampere, and Blackwell classes of edge GPUs, while preserving the broader Jetson software ecosystem [20]–[22]. Unlike datacenter accelerators, these SoCs share memory between CPU and GPU, operate under tight power and thermal budgets [23], and rely on platform-specific dynamic voltage and frequency scaling (DVFS) [24], [25] and telemetry interfaces [20]–[22]. These constraints introduce edge-specific bottlenecks that LLM-serving systems designed for datacenters [7]–[9], [26] do not encounter, so do not address. These constraints and bottlenecks also change how quantization affects edge LLM behavior. A format that reduces model footprint can still introduce backend-specific compute, memory-traffic, power, and thermal effects that single-precision characterization studies do not expose [27]–[36].

A growing body of work characterizes LLM inference on edge devices, including single-platform deep dives [37]–[40], mobile-platform benchmarks [41]–[43], analytical performance models [44], cost–latency–privacy tradeoff analyses [45], and on-device memory-tier studies [46]–[48]. Generic edge-AI benchmarks [49]–[53] have matured in parallel, as has server-side workload characterization [10], [54]–[58]. Yet, across this body of prior work, three methodological gaps persist. *First,* cross-generational comparisons within the same edge SoC lineage remain limited. Such comparisons are needed to expose how SoC microarchitecture evolution, including GPU generation, CPU organization, and memory bandwidth, changes LLM inference behavior on edge platforms. *Second,*

Accepted at IEEE IISWC 2026.

studies typically adopt either a Python-framework path (e.g., HuggingFace Transformers [59]) or a C++/GGML path (e.g., `llama.cpp` [60], [61]), but rarely both under the same per-prompt, phase-aware timing schema. Without such a common schema, backend differences across tokenization, prefill, per-token generation, de-tokenization, inter-token latency (ITL), and end-to-end latency remain difficult to compare directly. *Third,* timing and hardware telemetry are rarely fused at the same phase boundaries across backends, precisions, and platforms. As a result, studies often report aggregate end-to-end latency, throughput, or run-averaged power, but cannot directly compare; for example, decode-phase power, memory traffic, or energy per token between HuggingFace and `llama.cpp`. Prefill and decode can stress the SoC differently, but aggregate reporting collapses them into a single per-run number and obscures their distinct contributions, even though the prefill/decode boundary is central to server-side LLM serving studies [8], [9], [54].

We address these gaps with *Hydra*[1], a common-schema, phase-aware workload characterization framework for LLM inference on edge SoCs. Hydra instruments two structurally different inference backends (HuggingFace Transformers [59] and `llama.cpp` [60] on the GGML tensor library [61]) to emit per-prompt timing records through a canonical schema, and fuses that timing with high-resolution hardware telemetry from `tegrastats` [62] and the NVIDIA Management Library (NVML) [63], allowing CPU, GPU, memory, power, energy, and thermal behavior to be attributed to the prefill and decode windows of each prompt. Rather than tying the methodology to a single backend, precision, or platform instance, Hydra separates the measurement interface from the execution configuration: new models, precision formats, prompt lengths, and SoC generations can be added while preserving the same phase-aware analysis pipeline. In this paper, we instantiate Hydra across two inference backends, three Jetson generations, and five representative execution formats (`bf16`, `F16`, `Q8_0`, `Q6_K`, `Q4_K_M`), yielding a unified view that prior single-axis characterizations cannot provide.

Using Hydra, we conduct a multi-dimensional workload characterization of edge LLM inference. First, we analyze performance through end-to-end latency, decode throughput, phase-level timing, and input/output-length sensitivity. Second, we use phase-aligned telemetry to explain those performance trends through CPU-side runtime behavior, GPU effective utilization, and DRAM traffic. Third, we quantify the deployment cost through power, energy per token, total prompt energy, and thermal behavior.

Our analysis leads to three main findings:

Performance is jointly dependent on the choice of backend, model architecture, precision, and sequence length. End-to-end latency captures broad platform trends, but Hydra's phase timing shows that backend overhead, quantized decode throughput, and input/output-length scaling affect different parts of the inference pipeline.

Resource utilization explains why those performance trends occur. Backend choice changes CPU-side orchestration and GPU effective utilization; quantization reduces DRAM traffic without necessarily reducing GPU occupancy; and SoC-generation differences make raw utilization counters misleading to compare without first applying normalization.

Efficiency does not follow latency or bit-width alone. Lower-bit formats often reduce energy per token, but power and thermal behavior depend on the quantization implementation, runtime, and platform. Faster configurations may draw more power, yet still spend fewer joules per generated token.

This paper makes the following contributions:

Hydra, a cross-backend phase-aware characterization methodology. Hydra instruments HuggingFace Transformers and `llama.cpp` through a shared per-prompt schema, enabling direct comparison of tokenization, prefill, generation, de-tokenization, Time to First Token (TTFT), ITL, and end-to-end timing across structurally different backends.

Phase-aligned full-stack telemetry fusion. Hydra aligns runtime timing with CPU, GPU, memory-controller, power, energy, and thermal telemetry, producing canonical per-prompt records that attribute system behavior to prefill and decode windows.

A broad cross-generation edge-LLM characterization. We use Hydra to evaluate 13 models across seven families, three SoC generations, two inference backends, five representative execution formats, and input/output-length sweeps.

A multi-dimensional evaluation of edge LLM deployment behavior. We connect performance, utilization, and efficiency to show how backend orchestration, quantization format, SoC-generation behavior, and sequence length jointly determine latency, throughput, energy, and thermal costs.

An open phase-aware edge-LLM trace corpus. We release Hydra's unified per-prompt traces and analysis artifacts, including timing, system-resource, and efficiency aggregates, to support reproducible comparison against our cross-platform measurements (see the Artifact Appendix for details).

Hydra's source code and the collected per-prompt trace corpus are available open-source at:

- Public repository: https://github.com/amirtaherin/hydra.
- Archival copy: Zenodo, DOI 10.5281/zenodo.21844843.

II. Background and Motivation

Background. Modern generative LLMs are decoder-only Transformers [64] served through a prefill phase, which processes the input prompt and populates the KV cache, followed by a decode phase, which generates tokens autoregressively. These phases can stress edge SoCs differently because edge platforms couple CPU cores, GPU cores, memory controllers, and power-management logic within a shared memory and thermal envelope [20]–[22]. Deployment behavior also depends on the inference backend and execution format. High-level

[1] *Hydra* is named after the largest constellation, reflecting the breadth of this characterization: three SoC generations, two backends, 13 models, and five execution formats.

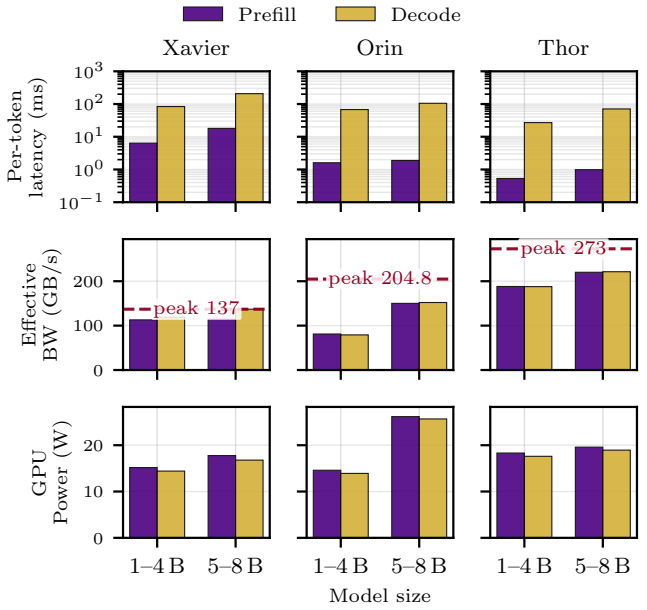

Fig. 1: *Phase-aware HuggingFace* `bf16` *inference across three consecutive SoC generations. Models are grouped into 1–4 B and 5–8 B bins. Rows show per-token latency, effective memory bandwidth, and GPU power for prefill and decode; dashed lines mark peak memory bandwidth.*

Python frameworks such as HuggingFace Transformers [59] provide broad model coverage, while lightweight backends such as `llama.cpp` [60] and GGML [61] reduce software overhead and enable efficient low-bit execution. We use *execution format* to refer to the backend-specific numeric representation used for inference, including 16-bit floating-point formats and lower-bit weight-only quantized formats. Lower-bit formats can reduce model footprint and traffic [27], [34], but the actual benefits depend on the backend implementation, dequantization overhead, and memory-system behavior. Finally, edge telemetry itself is platform-specific: `tegrastats` [62] and NVML [63] expose CPU, GPU, memory-controller, power, and thermal signals with different availability and semantics across SoC generations.

Motivation. Recent on-device LLM systems span phones, embedded SoCs, memory-tiered execution, and edge benchmarks [19], [46]–[48], but aggregate reporting still hides how performance, resource utilization, and efficiency interact across inference phases. Fig. 1 illustrates this problem. Along the *performance* dimension, prefill and decode show different per-token latency behavior across model sizes and SoC generations. Along the *resource-utilization* dimension, the same phase maps differently onto each memory subsystem relative to platform peak bandwidth. Along the *efficiency* dimension, GPU power does not track latency or bandwidth uniformly, so the fastest configuration is not necessarily the most energy-efficient one. These dimensions are coupled: edge LLM behavior depends on SoC generation, backend, execution format, and prompt/output length, but standard measurements usually expose only one dimension at a time, such as mean latency, throughput, utilization, or power. Hydra addresses this measurement problem by treating observability as a stack: token-level timing, phase-level prefill/decode timing, SoC-level resource signals, and efficiency-level power, energy, and thermal behavior. Hydra aligns these signals through a common phase-aware per-prompt schema, enabling the same analysis pipeline across HuggingFace Transformers and `llama.cpp`. The rest of the paper develops this methodology (§III & §IV) and applies it to evaluate performance, utilization, and efficiency (§V-A–§V-C).

III. Hydra: Phase-Aware Workload Characterization

Hydra is the measurement methodology for edge-LLM workload characterization. As shown in Fig. 2, Hydra converts heterogeneous backend execution and platform-specific telemetry into a common per-prompt record that can be analyzed by phase, backend, precision, and SoC generation. This section describes the three components of the methodology: *(i)* the common timing schema used to instrument HuggingFace Transformers and `llama.cpp`; *(ii)* the telemetry fusion process that aligns runtime events with SoC counters; and *(iii)* the canonical schema used to normalize platform-specific signals for downstream performance, utilization, and efficiency analysis. Section IV then describes how Hydra is instantiated for this study through the selected SoC generations, model families, workloads, execution formats, and inference configurations.

Measurement Model. Hydra uses the standard prefill/decode structure of LLM serving systems [8], [9], [54] as a measurement coordinate system. We distinguish between timing stages and telemetry windows. At the timing level, Hydra records tokenization, the prefill stage, per-token generation, de-tokenization, ITL, TTFT, and end-to-end latency. At the telemetry level, Hydra groups these events into two phase windows: a *prefill phase*, which covers tokenization and the prefill stage, and a *decode phase*, which covers autoregressive token generation and de-tokenization until the prompt completes. As shown in Fig. 2, for a prompt with start time t_{start}, tokenization time τ_{tok}, prefill-stage time τ_{prefill}, and end time t_{end}, Hydra defines the prefill-phase window as $[t_{\text{start}},\ t_{\text{start}} + \tau_{\text{tok}} + \tau_{\text{prefill}}]$, and the decode-phase window as $(t_{\text{start}} + \tau_{\text{tok}} + \tau_{\text{prefill}},\ t_{\text{end}}]$. All sampled hardware signals are then aggregated over the prompt, prefill, and decode windows using the same boundaries for both backends. These shared phase windows are the join point between backend timing records and SoC telemetry, enabling phase-attributed comparison across backends, execution formats, and edge-SoC generations.

Instrumentation and Telemetry Fusion. As shown in Fig. 2, Hydra instruments two structurally different inference backends: HuggingFace Transformers [59] in Python and `llama.cpp` [60] in C++ on the GGML tensor library [61]. Both backends emit one per-prompt timing record using a common schema for tokenization, prefill, per-token generation, de-tokenization, ITL, TTFT, end-to-end latency, and start/end timestamps. To avoid measuring

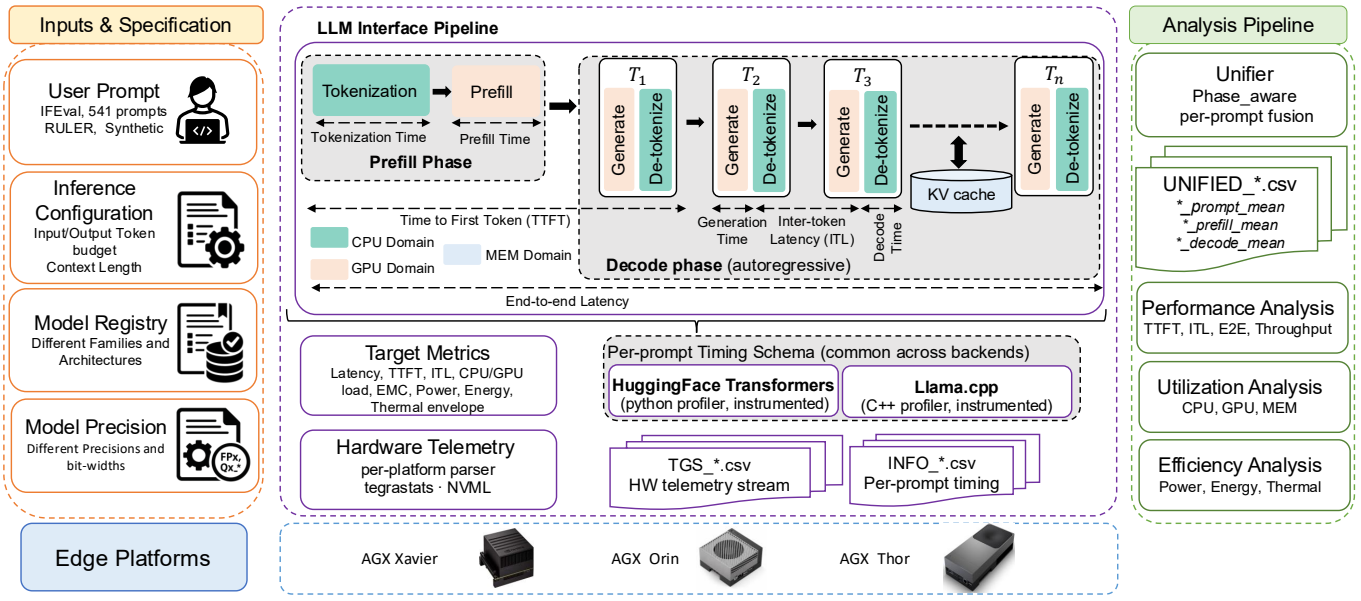

Fig. 2: *Hydra phase-aware workload characterization workflow. Hydra instruments HuggingFace Transformers and* `llama.cpp` *with a common per-prompt timing schema, collects SoC telemetry in parallel, aligns both streams per prompt, and normalizes the fused records into a canonical analysis format.*

asynchronous kernel launch time instead of completed GPU work, Hydra synchronizes CUDA-dependent regions with `torch.cuda.synchronize()` in the Python backend and `llama_synchronize()` in the C++ backend. The `llama.cpp` backend also records its internal performance counters for validation.

In parallel, Hydra samples SoC telemetry through `tegrastats` [62] and NVML [63]. The telemetry stream captures the platform-exposed CPU, GPU, memory-controller, RAM, power, and thermal signals. A platform-specific parser converts these raw samples into timestamped records on the same time base as the runtime profiler. Hydra then fuses backend timing with telemetry on a per-prompt basis. For each prompt, telemetry samples are aggregated over the full prompt window and over the prefill and decode phase windows as defined above. The resulting canonical record attaches prompt-level, prefill-level, and decode-level resource and efficiency statistics to each timing record. Because `tegrastats` samples at a fixed wall-clock interval, short prefill windows often contain only a few samples, while longer decode windows are sampled much more densely. Hydra emits per-prompt prefill aggregates as measured; the wider prefill variance in §V reflects this sampling asymmetry.

Canonical Cross-Platform Schema. Hydra normalizes platform-specific telemetry into a canonical schema so that the same analysis code can operate across Jetson generations. Raw telemetry differs by platform: CPU core counts, GPU organization, thermal-zone names, power-rail labels, and software-stack support are not identical across Xavier, Orin, and Thor devices. Hydra maps these disparate values into shared fields for CPU load, GPU activity, memory-controller behavior, RAM usage, power, and temperature, while preserving the source. This normalization is intentionally conservative. Hydra does not hide measurement asymmetries (i.e., some GPU-utilization and power channels come from different platform interfaces or represent different power domains). Cross-platform comparisons use the canonical schema for consistent data access, but the evaluation interprets platform-specific telemetry differences explicitly when they affect a result.

Validation and Artifact. Because HuggingFace Transformers and `llama.cpp` use different execution stacks, Hydra validates the timing definitions rather than assuming runtime equivalence. The common schema is enforced by construction: shared columns use identical formulas across both profilers. For `llama.cpp`, Hydra additionally logs the runtime's built-in performance counters alongside manual timings for every prompt. Prefill times agree within 0.1 ms, and per-token generation times differ by only 0.06–0.08 ms. This residual gap matches the cost of the logit read and greedy `argmax` included by Hydra, but excluded from the internal counter, making the difference bounded and explainable. Hydra is released as an open artifact with the profilers, telemetry parsers, unifier, canonical schema, analysis pipeline, and per-prompt record corpus; the Artifact Appendix details the repository and archival DOI.

IV. Hydra: Workload Characterization Design

Section III described how Hydra converts backend timing and SoC telemetry into a common phase-aware record. This section describes how we instantiate that workflow, shown in Fig. 2, to characterize edge LLM inference across the design space. Our experimental matrix is chosen to disambiguate four effects that are often conflated in edge LLM studies: SoC generation, model architecture, execution backend/format, and prompt/output length. We carry out our evaluation on

TABLE I: Evaluated edge SoC platforms.

Feature	Xavier	Orin	Thor
GPU arch.	Volta	Ampere	Blackwell
GPU SMs/CUDA cores	8 / 512	16 / 2048	20 / 2560
CPU	8-core Carmel	12-core A78AE	14-core Neoverse V3AE
Memory	32 GB LPDDR4X	32 GB LPDDR5	128 GB LPDDR5X
Peak BW	137 GB/s	204.8 GB/s	273 GB/s
L2/system cache	512 KB / –	4 MB / 4 MB	32 MB / 16 MB
Power budget	30 W	60 W	130 W
System Software	JP 5.1	JP 6.2	JP 7.1
CUDA ver.	11.4	12.6	13.2
PyTorch/Transformers	2.2.0 / 4.46.3	2.3.0 / 4.51.3	2.11.0 / 5.5.4

three edge-SoC generations, a set of instruction-tuned model families, multiple execution formats across HuggingFace Transformers and `llama.cpp`, and both fixed-workload and length-sensitivity prompt sets.

We evaluate three NVIDIA Jetson AGX edge-SoC generations: Xavier (Volta) [20], Orin (Ampere) [21], and Thor (Blackwell) [22]. These platforms define one axis of our workload characterization. Table I summarizes the compute resources, memory subsystem, power budget, and software stack of each platform. All three platforms use a unified-memory design: CPU and GPU activity share the same DRAM pool, eliminating explicit host-to-device transfers in the inference path. Across the three generations, memory capacity scales by roughly $4\times$ (32 GB $\rightarrow$ 128 GB), peak memory bandwidth by $\sim 2\times$ (137 $\rightarrow$ 273 GB/s), while the max power raises more than $4\times$ (30 $\rightarrow$ 130 W). Across these generations, Hydra studies how the evolution of edge-SoC microarchitecture can influence LLM inference behavior.

A. Models and Execution Formats

The model axis of our characterization covers 13 instruction-tuned, decoder-only LLMs from seven families: LLaMA, Qwen, Granite, Gemma, Phi, Mistral, and Moxin. The selected models span 1.24–8.54 B parameters (Table II), covering both compact sub-2 B models and the 7–8 B range that represents the practical upper end for 32 GB edge SoCs under 16-bit execution. This range lets Hydra separate effects caused by model scale from those caused by architecture, such as depth, hidden size, FFN expansion, attention layout, vocabulary size, and context length. The *Code* column in Table II defines the short identifiers used in the x-axis labels of figures throughout the paper. *Mem* is the `bf16` weight footprint. Q/KV denotes the query and key-value head counts; unequal values indicate grouped-query attention (GQA). Moxin-7B and Gemma-7B exceed the nominal 7B class, but we keep the developer-published names for consistency with public model registries.

The execution-format axis captures how backend and precision choices change system behavior. We evaluate HuggingFace (HF) Transformers at `bf16` and `llama.cpp` at `F16`, `Q8_0`, `Q6_K`, and `Q4_K_M`. The GGML quantized formats are weight-only post-training quantization formats at roughly 8, 6.6, and 4.5 bits per weight, respectively. Because our quantization study is conducted in `llama.cpp`, we use `F16` as the 16-bit quality reference and report the quality cost of the three GGUF quantized formats relative to that baseline. Note that

TABLE II: Evaluated instruction-tuned LLMs.

Code	Model	Family	Params	Mem	Architecture (L, H, FFN, Q/KV, Attn., Act., Vocab, Ctx.)
LL-1B	LLaMA-3.2-1B	LLaMA	1.24 B	2.36 GB	16, 2048, 8192, 32/8, GQA, SwiGLU, 128K, 131K
QW-1.5B	Qwen2.5-1.5B	Qwen	1.54 B	2.95 GB	28, 1536, 8960, 12/2, GQA, SwiGLU, 152K, 32K
GE-2B	Gemma-2B	Gemma	2.51 B	4.78 GB	18, 2048, 16384, 8/1, MQA, GELU, 256K, 8K
GR-2B	Granite-3.3-2B	Granite	2.53 B	4.83 GB	40, 2048, 8192, 32/8, GQA, SwiGLU, 49K, 131K
QW-3B	Qwen2.5-3B	Qwen	3.09 B	5.99 GB	36, 2048, 11008, 16/2, GQA, SwiGLU, 152K, 32K
LL-3B	LLaMA-3.2-3B	LLaMA	3.21 B	6.13 GB	28, 3072, 8192, 24/8, GQA, SwiGLU, 128K, 131K
PH-4B	Phi-3.5-mini	Phi	3.82 B	7.29 GB	32, 3072, 8192, 32/32, MHA, SiLU, 32K, 131K
QW-7B	Qwen2.5-7B	Qwen	7.62 B	14.57 GB	28, 3584, 18944, 28/4, GQA, SwiGLU, 152K, 32K
MI-8B	Ministral-8B	Mistral	8.02 B	15.30 GB	36, 4096, 12288, 32/8, GQA, SwiGLU, 131K, 32K
LL-8B	LLaMA-3.1-8B	LLaMA	8.03 B	15.32 GB	32, 4096, 14336, 32/8, GQA, SwiGLU, 128K, 131K
MO-7B	Moxin-7B	Moxin	8.11 B	15.48 GB	36, 4096, 14336, 32/8, GQA, SwiGLU, 32K, 32K
GR-8B	Granite-3.3-8B	Granite	8.17 B	15.58 GB	40, 4096, 12800, 32/8, GQA, SwiGLU, 49K, 131K
GE-7B	Gemma-7B	Gemma	8.54 B	16.28 GB	28, 3072, 24576, 16/16, MHA, GELU, 256K, 8K

TABLE III: Average quality change of each `llama.cpp` quantized format relative to `F16`. Positive ΔPPL is worse; negative Δaccuracy is worse.

Format	Mean ΔPPL	Max ΔPPL	Mean ΔWino.	Max drop	Mean ΔHella.	Max drop
`Q8_0`	−0.03	0.08	+0.20	0.47	+0.02	0.21
`Q6_K`	+0.01	0.23	−0.17	0.94	−0.01	0.30
`Q4_K_M`	+0.36	0.91	−0.78	2.37	−0.43	0.79

Xavier's Volta GPU lacks the native BF16 tensor-core path of Ampere and Blackwell, so Xavier `bf16` results reflect backend/software behavior; our native 16-bit cross-generation comparisons therefore rely on `llama.cpp F16`.

Table III summarizes quality changes on WikiText-2 perplexity (PPL) and zero-shot Winogrande [65] and HellaSwag [66] accuracy, evaluated with `lm-eval-harness` [67]. The quality cost is small: `Q8_0` and `Q6_K` remain close to `F16`, while `Q4_K_M` stays within 1 PPL point and about 2.4 accuracy points in the worst case. Thus, the system-level differences analyzed in §V primarily reflect runtime, precision, and platform effects rather than large quality regressions.

B. Evaluated Prompts and Sequence Lengths

The workload axis is designed to separate standard instruction-following behavior from controlled input- and output-length effects. The main characterization corpus uses IFEval [68], which contains 541 prompts spanning 25 verifiable instruction types. Prompt lengths range from 13 to 345 tokens (mean 47, std 23). For system measurements, we disable early stopping and use a fixed 500-token decode budget, ensuring that all models execute a uniform decode workload rather than stopping at model-dependent end-of-sequence points.

The main sweep combines three SoC generations, 13 models, and 5 execution formats: one HuggingFace format and four `llama.cpp` formats. This yields $3\times13\times(1_{\text{HF}}+4_{\text{llama.cpp}}) = 195$ (platform, model, execution-format) cells. Five Xavier `F16` configurations with 7–8 B models fail to load due to the JetPack 5 cuBLAS/NVMAP interaction discussed in §V-A; the remaining 190 cells contribute roughly 103,000 per-prompt records.

To test whether the IFEval trends hold beyond short prompts and fixed 500-token responses, we add two targeted sensitivity corpora:

- ***S1**: input-length sweep.* 90 prompts from RULER [69], balanced across three needle-in-a-haystack (NIAH)/tracking task families at exact input-length tiers of 1k, 3k, and 5k

tokens (10 prompts × 3 tasks × 3 tiers), with fixed 500-token decode budget.

- ***S2**: output-length sweep.* 30 IFEval prompts filtered to a 40–60 token input band and swept across output budgets of 1k, 3k, and 5k tokens, isolating decode-length scaling while holding prefill cost nearly fixed.

The sensitivity corpora run on Orin and Thor using `llama.cpp Q4_K_M` and HuggingFace `bf16` for six LLaMA/Qwen models (LL-1B/3B/8B and QW-1.5B/3B/7B). Xavier is excluded from these long-context sweeps because 5k-token prompts with 7–8 B models exceed its practical memory envelope. Together, the sensitivity corpora add roughly 4,300 records, bringing the full study to about 107,000 per-prompt records.

Execution Policy. The execution policy is chosen to match single-prompt interactive edge deployment. All runs use batch size 1, greedy decoding, token-by-token generation, and KV-cache reuse. Both HuggingFace Transformers and `llama.cpp` execute on the integrated GPU through CUDA. For `llama.cpp`, we request full GPU offload using `n_gpu_layers=99`; configurations that cannot allocate under this policy are marked as failed rather than partially offloaded, preserving a consistent backend comparison.

V. Evaluation

Hydra enables us to analyze edge LLM inference along three connected dimensions: performance, system-resource utilization, and efficiency. These dimensions map directly to Hydra's design: the common per-prompt timing schema exposes latency and throughput behavior; the phase-aligned telemetry fusion explains how CPU, GPU, and memory activity produce those trends, and the power, energy, and thermal metrics quantify the deployment cost of each backend, precision, model, and SoC-generation choice. We organize the analysis around three questions. **Q1**: How do platform generation, backend, model architecture, and precision affect latency? **Q2**: What CPU, GPU, and memory behaviors explain those trends? **Q3**: What are the associated power, energy, and thermal costs? This structure connects Hydra's phase-aware measurements to deployment decisions rather than reporting aggregate performance alone.

A. Performance Analysis

To answer **Q1**, we use Hydra's common timing schema to analyze performance from four views: end-to-end latency across SoC generations and backends (Fig. 3), quantized decode throughput (Fig. 4), phase-level latency attribution (Fig. 5), and input/output-length sensitivity (Fig. 6). Together, these views separate overall performance trends from the backend, execution format, model-architecture, and sequence-length effects that produce them.

End-to-end latency reflects platform, backend, and model-family effects. Fig. 3 shows the overall latency trends at similar 16-bit precision. Newer SoC generations reduce latency substantially, but the gain is not uniform: running HuggingFace `bf16`, Thor is about 2.8–2.9× faster than Xavier, while the Thor/Orin gap shrinks from roughly 2.9× on LL-1B to about 1.4× on GE-7B. Our backend choice also impacts performance: at similar 16-bit execution, `llama.cpp` consistently improves end-to-end latency over HuggingFace, with the largest gains on smaller models where runtime overhead is a larger fraction of decode time. Finally, model family matters within the same, similarly sized models: 7–8 B models show noticeably different latencies, and several Xavier `F16 llama.cpp` runs fail to allocate memory despite 32 GB of unified memory. Thus, end-to-end latency already shows that edge LLM performance is not determined by parameter count or hardware generation alone; it depends on the interaction between platform, backend, and model architecture.

Quantization changes the decode-throughput behavior. Fig. 4 shows that `Q4_K_M` substantially increases decode throughput across all three generations by reducing the weight traffic per generated token. The effect is large enough to make smaller models practical (even on Xavier): sub-3 B models reach roughly 25–60 tok/s, despite Xavier's older Volta architecture and lower memory bandwidth. For 7–8 B models, however, throughput compresses into a much narrower range: Thor reaches 35–44 tok/s, Orin 22–28 tok/s, and Xavier 13–16 tok/s. Thus, quantization shifts the deployment boundary, but it does not eliminate the effects of model scale or SoC generation.

Phase attribution explains backend latency differences. The previous figures show that our choice of backend impacts end-to-end latency; Fig. 5 shows why. Hydra reports the same timing stages for HuggingFace and `llama.cpp`, separating model execution from runtime overheads. Running HuggingFace, ITL takes substantially more time than raw generation because each token step also pays CPU-side orchestration and de-tokenization costs. For QW-7B on Thor, generation takes 33.0 ms, but ITL reaches 63.5 ms; the additional cost is largely exposed as de-tokenization and per-step runtime overhead. Under `llama.cpp`, de-tokenization is sub-millisecond and ITL nearly matches generation time (58.6 ms vs. 58.5 ms), indicating that most per-token latency is spent in the measured generation stage.

This attribution reveals a non-obvious backend tradeoff: `llama.cpp` can achieve lower end-to-end latency, even when its raw per-token generation latency is higher than HuggingFace. The advantage comes from lower runtime overhead, not simply faster model kernels. The tighter TTFT and prefill variation under `llama.cpp` further suggest more stable prompt-side execution, which we revisit in the system-utilization analysis (§V-B). Thus, Hydra turns an aggregate backend comparison into a stage-level explanation of where latency is introduced and where optimization effort should be directed.

Input and output length stress different performance components. Fig. 6 separates input-length scaling from output-length scaling. In the input sweep, TTFT increases almost proportionally with prompt length: increasing the input tokens from 1k to 5k increases the TTFT by about 5.0–5.7× across

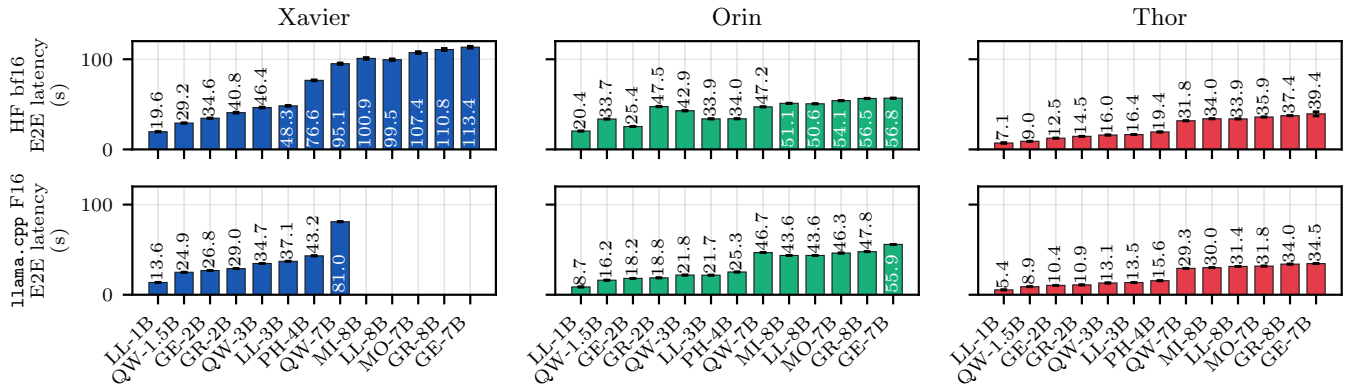

Fig. 3: *End-to-end latency across SoC generations and backends. Missing bars indicate failed full-GPU-offload configurations.*

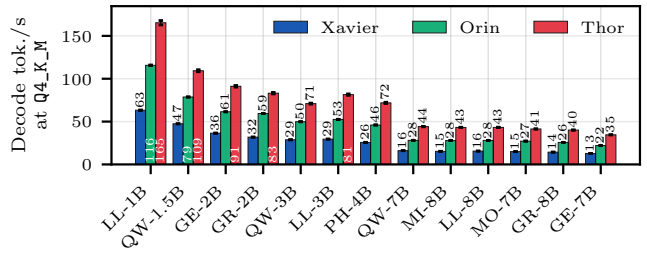

Fig. 4: `llama.cpp` *decode throughput at* `Q4_K_M` *across three SoC generations. Quantization improves decode throughput and compresses the cross-platform gap relative to 16-bit execution.*

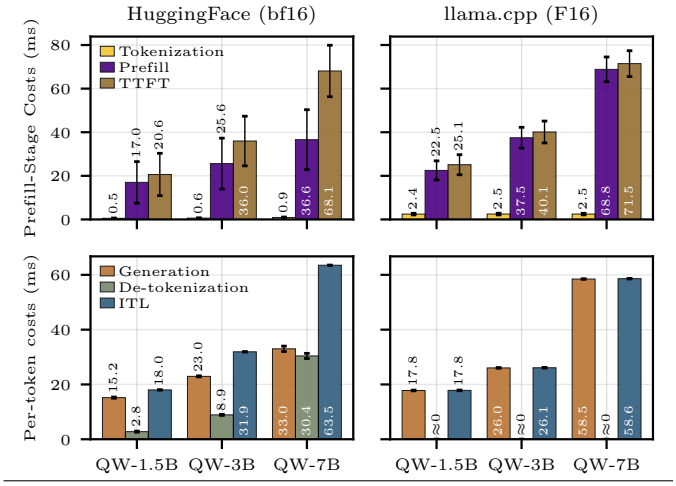

Fig. 5: *Phase-level latency decomposition on Thor for the Qwen2.5 family under HuggingFace* `bf16` *and* `llama.cpp F16`*. Hydra exposes prefill-stage timing, per-token generation, de-tokenization, TTFT, and ITL under the same schema.*

the tested models and platforms. For larger models, this pushes first-token latency into the multi-second range, making prefill responsiveness the limiting factor for long-context interactive use. The output sweep shows the opposite behavior. With short inputs fixed, decode throughput remains nearly flat as the output budget increases from 1k to 5k tokens, changing by only a few percent in most cases. Thus, within this range, longer outputs do not substantially degrade steady-state token generation, while longer inputs directly increase first-token delay. This separation is exactly why Hydra treats prefill and decode as distinct performance components rather than collapsing them into a single latency or throughput number.

B. System Resource Utilization

To answer **Q2**, Hydra shows that the performance trends in §V-A arise from three system mechanisms: CPU-side runtime orchestration, GPU effective utilization, and DRAM traffic. Our choice of backend changes how much the CPU performs between token steps; quantization changes the amount of DRAM traffic needed per generated token; and SoC generation changes how GPU frequency, memory bandwidth, and utilization interact. These effects are only visible because Hydra aligns timing and telemetry at the same per-prompt phase boundaries.

CPU-side behavior explains backend overhead. Fig. 7 explains the backend effect observed in §V-A. Running HuggingFace `bf16`, CPU activity is visible on every generation: Xavier keeps all cores at peak frequency, Orin shows scheduler migration across its three clusters, and Thor concentrates work on a small number of cores, while the rest remain mostly idle. These patterns show that HuggingFace inference is not only a GPU workload; each token step also includes CPU-side orchestration, synchronization, and runtime management overhead.

Running `llama.cpp F16`, the CPU footprint largely disappears. The per-core load drops below 15% across all platforms, with most cores near idle frequency. This is the system-level counterpart of the phase result in Fig. 5: when the backend removes most per-token CPU orchestration, ITL aligns closely with generation time and the prompt-stage variance is reduced. Thus, backend speedups depend on factors beyond GPU kernels; they also need to account for CPU-side work between token steps.

GPU and DRAM utilization explain backend and quantization trends. Table IV connects the performance trends from §V-A to GPU and memory behavior on Orin. We report effective GPU utilization $U_{\text{eff}} = \text{load} \times \text{freq}/\text{freq}_{\text{peak}}$ and effective DRAM bandwidth BW_{eff}, using Orin's 204.8 GB/s peak bandwidth. The key comparison is the decode phase, which dominates end-to-end latency for the 500-token main

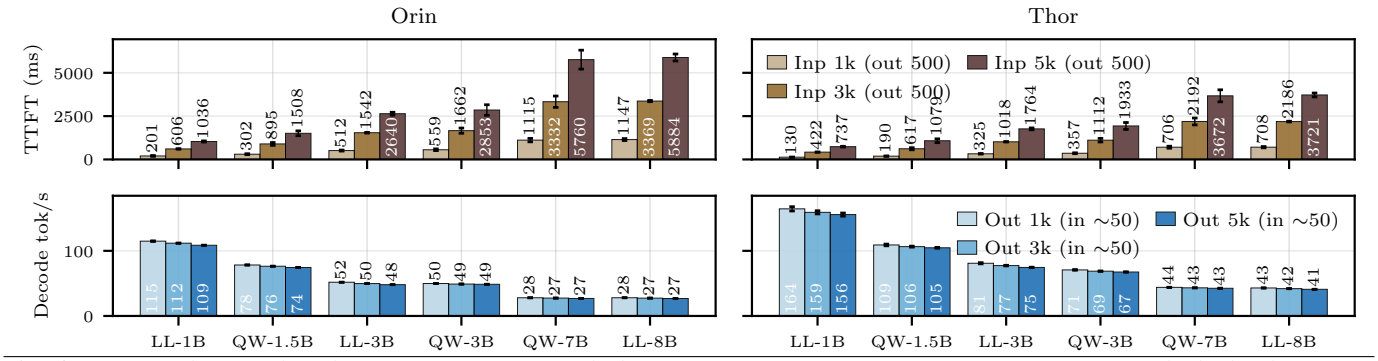

Fig. 6: *Input- and output-length sensitivity on Orin and Thor using* `llama.cpp Q4_K_M`*. Top: TTFT for 1k/3k/5k-token inputs with 500 output tokens. Bottom: decode throughput for 1k/3k/5k output budgets with short inputs.*

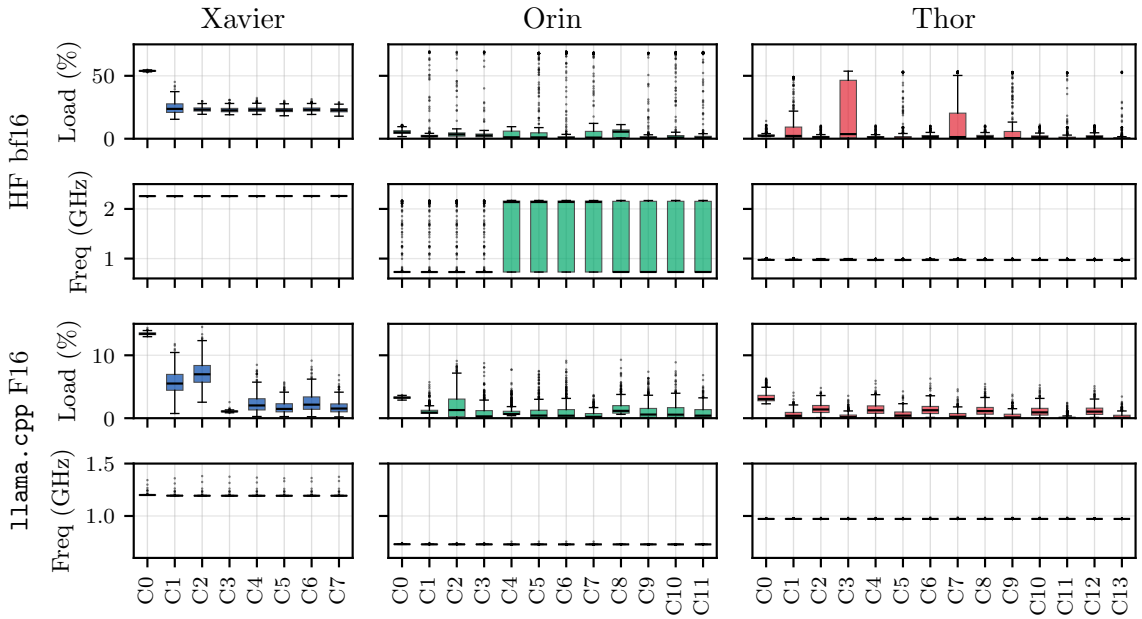

Fig. 7: *Per-core CPU load and frequency for Qwen2.5-7B across SoC generations and backends. HuggingFace exposes CPU-side orchestration and DVFS behavior, while* `llama.cpp` *largely removes per-step CPU activity.*

workload; prefill is included for completeness, but has higher variance because its execution windows are shorter.

The first lesson is that backend choice changes how well the GPU is fed. For small Qwen models, HuggingFace leaves both GPU utilization and DRAM bandwidth well below peak, matching the CPU-side orchestration gaps observed in Fig. 7. Switching to `llama.cpp` at the same 16-bit precision, decode utilization increases and is close to saturation. The effective bandwidth also increases without changing the model. This explains why `llama.cpp` improves end-to-end latency in Fig. 3: the native runtime reduces CPU-side gaps and keeps the accelerator busier.

The second lesson is that quantization reduces memory pressure without necessarily reducing GPU occupancy. Across `llama.cpp` formats, decode U_{eff} remains high, while BW_{eff} falls as weight precision decreases. This explains the throughput behavior in Fig. 4: quantization improves tokens/s by reducing DRAM traffic per token, but the GPU can remain highly occupied because the runtime continues streaming token work efficiently.

The third lesson is that model scale amortizes backend overhead. For QW-7B, HuggingFace and `llama.cpp` both result in high GPU utilization and similar DRAM bandwidth utilization at 16-bit precision. At this size, the amount of per-token model work is large enough that the Python-side overhead becomes a smaller portion of the decode step. Thus, the backend performance gap that is large for smaller models narrows as model size increases, matching the end-to-end trend in Fig. 3.

Length sweeps explain prefill/decode utilization behavior. Table V connects the length-sensitivity results from Fig. 6 to GPU utilization. The primary pattern is that input length changes prefill utilization, while output length does not substantially change decode utilization. On Orin, increasing the input from 1k to 5k tokens raises prefill U_{eff} for smaller Qwen models, indicating that longer prompts provide enough work to keep the GPU busy during the prefill phase. In contrast,

TABLE IV: Phase-split GPU effective utilization and DRAM effective bandwidth on Orin for the Qwen2.5 family across backends and execution formats. Values are mean(std) over prompts.

		HF bf16		F16		Q8_0		Q6_K		Q4_K_M	
Model	Phase	U_{eff}	BW_{eff}	U_{eff}	BW_{eff}	U_{eff}	BW_{eff}	U_{eff}	BW_{eff}	U_{eff}	BW_{eff}
QW-1.5B	Pre	28.7(1.6)	47.1(1.8)	86.2(14.6)	96.0(3.8)	89.1(12.2)	85.9(3.2)	87.4(11.6)	81.7(2.7)	88.8(11.6)	81.8(2.3)
	Dec	27.3(0.2)	47.1(0.2)	97.1(0.1)	96.2(0.3)	95.9(0.1)	86.0(0.4)	95.3(0.2)	81.6(0.5)	94.1(0.3)	81.7(0.5)
QW-3B	Pre	42.2(6.6)	75.5(2.9)	85.4(11.1)	140.9(5.8)	85.4(10.2)	106.3(4.1)	84.8(10.8)	100.1(3.7)	85.2(12.1)	100.4(3.5)
	Dec	38.5(0.4)	73.9(0.6)	97.4(0.1)	141.2(0.3)	96.8(0.1)	106.4(0.3)	96.3(0.1)	100.3(0.4)	95.5(0.2)	100.4(0.5)
QW-7B	Pre	90.9(5.7)	151.2(6.3)	91.1(6.1)	150.4(6.2)	91.2(6.7)	128.6(5.1)	91.7(6.6)	123.4(4.9)	90.1(8.0)	127.2(5.1)
	Dec	95.6(0.1)	152.8(0.3)	98.2(0.1)	151.5(0.2)	97.7(0.1)	130.2(0.4)	97.5(0.1)	124.7(0.3)	97.1(0.1)	128.6(0.4)

TABLE V: HF `bf16` GPU effective utilization on Orin and Thor across input- and output-length sweeps for the Qwen2.5 family. Values are mean(std) over prompts.

Model	Platf.	Phase	Input sweep, Out=500			Output sweep, In=50		
			1k	3k	5k	1k	3k	5k
QW-1.5B	Orin	Pre	53.7(10.4)	74.9(4.2)	80.8(3.7)	27.8(4.7)	28.3(4.9)	26.2(5.0)
	Orin	Dec	28.8(0.1)	30.6(0.3)	32.8(0.5)	27.9(0.1)	28.6(0.1)	27.8(0.2)
	Thor	Pre	63.2(20.8)	55.5(17.2)	59.8(27.8)	68.2(25.2)	73.6(21.6)	70.9(16.2)
	Thor	Dec	85.4(0.5)	85.7(0.3)	86.3(0.3)	85.5(0.3)	85.7(0.1)	85.9(0.1)
QW-3B	Orin	Pre	79.9(10.0)	92.0(1.8)	94.1(1.1)	39.8(6.7)	39.7(7.2)	38.2(7.0)
	Orin	Dec	42.7(0.6)	46.9(2.1)	52.0(3.0)	40.2(0.1)	41.2(0.2)	40.8(0.2)
	Thor	Pre	78.5(18.9)	87.7(7.9)	86.7(6.2)	74.3(22.1)	79.4(19.5)	73.1(20.3)
	Thor	Dec	88.9(0.3)	88.9(0.2)	89.2(0.2)	88.9(0.2)	88.8(0.1)	88.9(0.1)
QW-7B	Orin	Pre	95.5(11.2)	97.7(0.7)	97.9(0.4)	90.8(17.3)	86.0(17.0)	79.9(21.7)
	Orin	Dec	95.4(0.2)	95.6(0.2)	95.8(0.1)	95.4(0.1)	95.3(0.1)	95.4(0.1)
	Thor	Pre	74.2(26.4)	72.4(13.7)	73.3(8.5)	77.7(19.3)	76.8(18.4)	72.9(20.6)
	Thor	Dec	92.9(0.3)	93.0(0.2)	93.2(0.2)	92.8(0.2)	92.9(0.1)	93.0(0.1)

TABLE VI: Decode-phase efficiency on Orin and Thor for the Qwen2.5 family. Each metric cell reports Orin/Thor. Values are mean (std) over prompts; temperatures in °C

Model	Pr.	Power (W)	mJ/tok	T_{GPU}	T_{CPU}
QW-1.5B	bf16	16.2(0.1)/32.8(0.1)	1092(4)/591(1)	50.9(0.3)/55.9(0.1)	54.2(0.4)/55.8(0.5)
	F16	35.4(0.1)/47.2(0.3)	1142(3)/843(2)	63.8(0.4)/65.2(0.4)	63.9(0.5)/63.6(0.4)
	Q8	34.9(0.1)/37.3(0.2)	672(2)/561(2)	63.1(0.3)/58.3(1.0)	63.2(0.3)/56.7(1.0)
	Q6	42.8(0.1)/55.7(0.5)	676(2)/533(2)	68.2(0.5)/69.6(0.5)	67.2(0.5)/67.0(0.4)
	Q4	39.9(0.1)/47.0(0.4)	509(1)/431(2)	63.8(0.5)/62.0(0.6)	62.9(0.5)/60.7(0.5)
QW-3B	bf16	20.9(0.2)/34.1(0.1)	1788(10)/1088(3)	54.0(0.5)/56.7(0.4)	56.8(0.5)/55.7(0.5)
	F16	41.5(0.1)/53.6(0.2)	1812(3)/1399(4)	65.4(0.4)/69.1(0.2)	65.1(0.4)/67.0(0.2)
	Q8	39.0(0.1)/42.7(0.2)	1209(2)/981(3)	66.1(0.2)/62.1(0.4)	66.1(0.3)/60.4(0.4)
	Q6	48.5(0.1)/50.4(0.2)	1227(3)/1012(2)	69.7(0.8)/66.5(0.3)	68.3(0.9)/64.5(0.3)
	Q4	45.0(0.1)/53.5(0.4)	903(2)/755(2)	67.9(0.3)/67.2(0.5)	66.8(0.3)/65.0(0.5)
QW-7B	bf16	39.6(0.1)/37.8(0.1)	3731(11)/2399(7)	64.9(0.5)/59.5(0.3)	66.0(0.6)/58.1(0.4)
	F16	43.5(0.0)/51.2(0.0)	4068(4)/3000(5)	67.7(0.3)/67.5(0.2)	67.5(0.4)/65.5(0.2)
	Q8	42.5(0.1)/51.5(0.2)	2462(7)/1864(7)	65.4(1.0)/66.7(1.0)	65.2(1.1)/64.7(1.0)
	Q6	54.8(0.1)/68.4(0.3)	2559(4)/1983(7)	70.4(0.3)/74.3(0.8)	68.3(0.3)/71.8(0.8)
	Q4	52.3(0.1)/65.8(0.3)	1867(4)/1493(4)	71.6(0.3)/74.6(0.3)	70.0(0.3)/72.5(0.4)

increasing the output budget from 1k to 5k tokens leaves decode U_{eff} nearly flat, because each generated token still executes the same per-token loop.

The Orin/Thor contrast also shows why utilization must be interpreted with platform behavior in mind. Thor maintains high decode U_{eff} across the sweep, while Orin achieves lower utilization on smaller models and approaches saturation only for QW-7B. This explains the **Q1** sensitivity result in Fig. 6: longer inputs increase first-token cost by expanding prefill work, whereas longer outputs mostly extend steady-state decode without changing the per-token utilization behavior. Thus, sequence length affects performance through different system paths, depending on whether it expands the prompt or the generated output.

C. Efficiency

To answer **Q3**, we use Hydra's power, energy, and thermal metrics to quantify the cost of the performance and utilization trends identified in §V-A and §V-B. The key question is not only which configuration is faster, but which configuration produces more tokens per joule and whether this power and thermal behavior is sustainable. We analyze decode-phase power, energy per generated token, thermal response, and total prompt energy across backends, execution formats, model sizes, and SoC generations.

Quantization reduces energy per token, but power is not monotonic. Table VI shows that lower-bit formats reduce energy per generated token across both Orin and Thor. This follows directly from **Q2**: quantization reduces DRAM traffic while maintaining high GPU occupancy, so each token completes faster and burns less energy. The effect is most noticeable at `Q4_K_M`, which consistently delivers the lowest mJ/token among the `llama.cpp` formats.

Power, however, does not follow bit-width monotonically. `Q6_K` often draws more power and runs hotter than `Q8_0` and `Q4_K_M` , despite using fewer bits than `Q8_0`. This reinforces a key deployment lesson: quantization format is not only a memory-footprint choice. The format also changes unpacking, scaling, and dequantization work, so power and thermal behavior must be measured rather than inferred from bit-width alone.

Thor spends more watts but fewer joules per token. Across matched model/format cells, Thor usually draws more instantaneous power than Orin, but its higher throughput yields lower energy per token. This connects **Q1** and **Q2**: Thor's newer GPU and memory subsystem increase the token rate enough to offset the higher power draw. The result is that platform choice changes the energy operating point, not just the latency operating point.

Thermals identify the active subsystem, but do not bind single-stream inference. The CPU/GPU temperature rows mirror the utilization trends from §V-B. Running HuggingFace on Orin, CPU temperatures can exceed GPU temperatures, consistent with CPU-side runtime orchestration. Under `llama.cpp`, the GPU becomes the warmer component, matching the shift toward sustained GPU execution. Across all measured single-stream runs, temperatures remain below throttling-relevant

TABLE VII: Total energy per prompt across input- and output-length sweeps. Values are mean(std) over prompts.

Model	Plat.	Format	Input sweep, Out=500			Output sweep, In=50		
			1k	3k	5k	1k	3k	5k
QW-1.5B	Orin	HF bf16	530(2)	562(4)	597(8)	1034(2)	3126(10)	5401(21)
	Orin	lcpp Q4	273(3)	314(6)	365(13)	516(2)	1587(2)	2723(3)
	Thor	HF bf16	316(2)	347(5)	375(7)	604(2)	1893(3)	3245(4)
	Thor	lcpp Q4	232(3)	264(5)	296(8)	438(2)	1348(5)	2291(7)
QW-3B	Orin	HF bf16	884(5)	956(17)	1045(32)	1687(6)	5119(13)	8663(27)
	Orin	lcpp Q4	482(4)	542(8)	604(15)	913(2)	2767(2)	4660(4)
	Thor	HF bf16	583(4)	634(7)	677(9)	1111(3)	3486(3)	5961(5)
	Thor	lcpp Q4	412(3)	472(8)	534(15)	769(3)	2369(6)	4042(8)
QW-7B	Orin	HF bf16	1875(12)	2035(25)	2167(30)	3637(7)	11038(17)	18615(23)
	Orin	lcpp Q4	994(9)	1135(20)	1279(33)	1883(7)	5712(8)	9735(28)
	Thor	HF bf16	1272(14)	1347(13)	1416(13)	2464(4)	7543(10)	12805(20)
	Thor	lcpp Q4	794(9)	914(16)	1029(26)	1506(6)	4600(18)	7810(27)

levels, so energy/token is a more useful constraint than peak temperature.

Total prompt energy is dominated by generated length. Table VII shows that total energy grows primarily with output length. Increasing generated tokens from 1k to 5k scales energy almost proportionally (5.1 to 5.4×) because decode dominates the run time. Increasing input length also raises energy, but more modestly (11 to 34%), because prefill is paid once while decode is paid once per generated token. This matches the **Q1** sensitivity result in Fig. 6: long inputs mainly increase first-token delay, while long outputs accumulate steady-state decode cost.

Backend, precision, and platform choices compound. The total-energy sweep shows that no single axis explains deployment cost. Moving from HuggingFace `bf16` to `llama.cpp Q4_K_M` reduces both runtime overhead and weight traffic; moving from Orin to Thor improves the token rate at higher instantaneous power. Combined, these choices can more than double the number of prompts served under the same energy budget. Thus, the practical efficiency decision is joint: backend, precision, model size, sequence length, and SoC generation must be chosen together.

Efficiency takeaway. Q3 completes the chain from **Q1** and **Q2**. Performance gains become deployment gains only when they reduce energy per useful token. Hydra shows that quantization, native runtime structure, and newer SoC generations can all improve that metric, but the power and thermal consequences are format- and platform-dependent. Aggregate latency alone would miss these tradeoffs.

VI. Discussion, Limitations, and Future Directions

Practical deployment implications. Hydra's characterization translates into concrete deployment guidance:

- Latency-bound small-model deployments should utilize backends with low orchestration overhead;
- Energy-bound deployments should focus on mJ/token, rather than nominal bit-width, since `Q4_K_M` is often best, while `Q6_K` can regress;
- Power/thermal-capped systems do not always benefit, in terms of power, from using lower precision; long-input interactive workloads should optimize prefill/TTFT, while long-output workloads should optimize decode mJ/token; and
- Platform selection must consider both native precision support and allocator/runtime constraints.

We found these choices can more than double the number of prompts served under a fixed energy budget.

Measurement overhead and validity. Hydra's timing overhead is bounded by the GPU synchronization needed for correct phase attribution (< 0.1 ms per phase). The per-token 0.06–0.08 ms logit-read/argmax cost is part of decoding, and our measurements agree with `llama.cpp` internal counters (27.04 vs. 27.00 ms prefill). Telemetry is collected out-of-process, and each per-prompt record is $\sim$0.4 KB. The main threats to validity are the telemetry sampling asymmetry discussed in §III, the batch-size-one scope, and the platform-specific telemetry gaps covered below.

Limitations. Hydra is designed as a portable characterization methodology, but this study instantiates it on three NVIDIA Jetson SoCs. The measurement schema is backend- and platform-extensible, yet the specific performance, utilization, and efficiency trends reported here reflect the Jetson software stack, telemetry interfaces, CUDA path, and power-management policies. Second, our quantization study focuses on weight-only GGUF formats (`Q8_0`, `Q6_K`, and `Q4_K_M`); activation quantization, KV-cache quantization, FP8 execution, and fully integer pipelines may shift the balance between memory traffic, dequantization work, and energy. Third, we evaluate single-stream, batch-size-one interactive inference. Continuous batching, concurrent requests, speculative decoding, and serving-style scheduling can change both throughput and resource contention. Fourth, our model set is limited to dense instruction-tuned LLMs in the 1–8 B range; mixture-of-experts, multimodal, and larger dense models may expose different memory and runtime behavior. Finally, power telemetry is limited by the rails exposed on each platform. Hydra reports the available GPU, CPU/SoC, and memory/IO rails consistently, but none of the evaluated boards exposes a clean DRAM-only power rail comparable to a discrete memory power sensor.

Future directions. Hydra's phase-aware schema enables several follow-up studies. First, adaptive runtime control can use per-phase GPU utilization, memory bandwidth, energy, and thermal headroom to change precision, offload policy, or scheduling decisions during prefill and decode rather than once per request; this is not directly supported by current server-oriented LLM serving systems [7]–[9]. Second, KV-cache management is a natural extension: our output-length sweeps show that total energy grows with generated length, motivating KV-cache compression, eviction, or precision adaptation for long generations. Third, Hydra can be extended beyond Jetson-class SoCs by adding platform-specific telemetry parsers while preserving the canonical per-prompt schema. This would enable direct comparison across AMD Ryzen AI, Qualcomm Snapdragon, Apple M-series, and emerging RISC-V edge accelerators. Finally, extending the current single-stream sweeps to longer contexts and concurrent serving would result in KV-

cache fragmentation, memory-capacity pressure, and DRAM-contention, which is future work.

VII. Related Work

We position Hydra against prior LLM characterization and benchmarking work along the dimensions that matter for edge workload characterization: hardware coverage, backend support, model and precision scope, phase-aware timing, SoC telemetry, efficiency metrics, and released artifacts. Table VIII summarizes the comparison, where: *Common schema* means that multiple inference backends are instrumented using the same per-prompt, phase-aware timing and telemetry schema, *Phase* denotes whether metrics are attributed separately to prefill and decode windows.

Edge LLM characterization. Recent edge LLM studies span mobile devices, Raspberry Pi-class systems, Jetson platforms, and analytical models [37]–[40], [44], [45]. The closest edge-side comparisons are MELTing Point [42] and PalmBench [41]: both evaluate compressed or mobile LLM execution across edge-class hardware and include runtime, energy, or thermal measurements. Other studies motivate on-device LLM deployment through memory-tiering, mobile model design, speculative decoding, and foundation-model firmware paths [43], [46]–[48]. These efforts established the importance of edge LLM measurement, but did not combine HuggingFace and `llama.cpp` under one per-prompt phase-aware schema with CPU/GPU/memory, power, energy, and thermal telemetry across multiple edge-SoC generations.

Phase-aware serving and benchmarking. Server-side characterization has shown the value of separating prefill and decode. TokenPowerBench [54] measures phase-level power on H100s, while Splitwise [8], DistServe [9], `vLLM` [7], and ELLIE [70] motivate phase-aware serving and KV-cache management. Trace generation, simulation, and server benchmarks consider request dynamics, scheduling, and accelerator behavior in datacenters [10], [26], [55]–[58], [71]. Hydra offers a richer picture: measured single-device edge SoCs, where shared memory bandwidth, CPU–GPU coordination, DVFS, and thermal headroom shape execution.

Broader benchmarks, surveys, and quantization. Generic edge and ML benchmarking efforts (e.g., the MLPerf suites) standardize top-line latency, throughput, power, and model-size reporting across devices and workloads [19], [49]–[53], [72]. In parallel, post-training and hardware-aware quantization methods reduce LLM footprint and improve deployability [27]–[36], [73]–[75]. Hydra is complementary to these efforts: it does not propose a new quantizer or benchmark score, but characterizes how deployed execution formats affect timing, resource utilization, energy, and thermals on edge SoCs.

Hydra's position. Hydra connects pieces that prior work typically studies separately: cross-generation edge SoC behavior, dual-backend instrumentation (HuggingFace Transformers and `llama.cpp` [60]), 16-bit and GGUF quantized execution formats, phase-attributed telemetry, and an open per-prompt trace corpus. This combination is the key distinction: the same canonical record links token-level timing to system-resource behavior and efficiency, allowing backend, precision, model-family, and SoC-generation effects to be compared within one workload-characterization framework.

TABLE VIII: Hydra relative to other LLM characterization/benchmarking tools. Columns use no/part/full to denote whether none, some, or all submetrics are reported.

	Hardware		Backend			Models			Coverage				Reporting		Artifact		
Work	Edge	Cross-gen.	llama.cpp	HF	Comm. sch.	# models	# families	Cross-arch	Precision	Timing	HW perf.	Efficiency	Phase	Per-prompt	Tool	Dataset	# metrics
Dhar [37]	yes	no	yes	no	no	1	1	no	part	part	part	no	no	no	no	no	5
Nezami [38]	yes	no	yes	no	no	8	7	yes	part	part	part	no	part	no	yes	no	4
PalmBench [41]	yes	yes	yes	no	no	10	8	yes	full	part	full	part	part	no	yes	no	7
LLMPi [39]	yes	no	yes	no	no	7	4	no	full	part	no	part	no	no	no	no	6
Husom [40]	yes	no	yes	no	no	4	3	no	part	part	no	part	no	part	yes	no	4
Jang & Mor. [45]	yes	no	–	–	–	17	4	no	–	part	no	part	no	no	no	no	4
EdgeProf. [44]	no	no	no	no	no	4	4	no	part	part	no	part	no	no	yes	no	4
MELT [42]	yes	no	yes	no	no	7	5	yes	part	part	full	full	full	yes	yes	no	7
ServeGen [10]	no	no	–	–	no	10	3	no	–	part	no	no	no	yes	yes	yes	4
TokenPow. [54]	no	no	no	yes	no	15	4	yes	part	part	no	part	full	part	yes	no	5
TokenSim [57]	no	no	no	no	no	2	2	no	part	part	no	no	part	no	yes	no	2
LLMSrvSim [58]	no	no	no	no	no	6	2	no	–	no	part	no	part	no	yes	no	2
LLM-IB [56]	no	no	yes	no	no	8	3	yes	part	part	no	part	part	no	yes	yes	5
Hydra (ours)	**yes**	**yes**	**yes**	**yes**	**yes**	**13**	**7**	**yes**	**full**	**full**	**full**	**full**	**full**	**yes**	**yes**	**yes**	**16**

VIII. Conclusion

This paper presented *Hydra*, a common-schema, phase-aware workload characterization methodology for LLM inference on edge SoCs. Hydra aligns backend timing from HuggingFace Transformers and `llama.cpp` with system telemetry, enabling performance, utilization, and efficiency to be attributed to prefill and decode windows. Across three Jetson generations, thirteen LLMs, five execution formats, and input/output-length sweeps, Hydra shows that edge LLM behavior cannot be explained by model size, precision, backend, or platform generation alone. Backend structure changes where latency is introduced, quantization reduces memory traffic and energy, but does not predict power monotonically, and SoC generation changes how utilization and efficiency should be interpreted. These results show that practical edge LLM deployment requires phase-aware, backend-aware, and platform-aware observability rather than aggregate latency/throughput alone. We release Hydra and its per-prompt trace corpus to support reproducible characterization and future edge-LLM research.

Acknowledgment

We would like to thank Matin Raayai Ardakani for his assistance during artifact evaluation and anonymous reviewers for their constructive feedback. This work was partially supported by the U.S. National Science Foundation (NSF) SaTC program under Grant No. 2414652, the EU Project dAIEDGE (GA Nr 101120726), and the Innovate UK Horizon Europe Guarantee (GA Nr 10090788). In preparing this paper, the authors used generative AI tools (OpenAI ChatGPT and Anthropic Claude) for language and presentation refinement. All technical content, results, and claims were produced and verified by the authors, who take full responsibility for the content of this paper.

REFERENCES

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] NLLB Team, M. R. Costa-Jussà, J. Cross, O. Çelebi *et al.*, "Scaling neural machine translation to 200 languages," *Nature*, vol. 630, no. 8018, pp. 841–846, 2024, arXiv:2207.04672.

[3] Y. Zhang, S. Sun, M. Galley, Y.-C. Chen, C. Brockett, X. Gao, J. Gao, J. Liu, and B. Dolan, "DialoGPT: Large-scale generative pre-training for conversational response generation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations (ACL)*. Association for Computational Linguistics, 2020, pp. 270–278, arXiv:1911.00536.

[4] OpenAI, "ChatGPT: Language model," https://openai.com/chatgpt, 2023.

[5] Anthropic, "Claude," https://www.anthropic.com, 2023.

[6] T. Rupprecht, P. Zhao, A. Taherin, A. Akbari, A. Akbari, Y. He, T. Imtiaz, S. Duffy, J. Lin, Y. Chen, R. Chowdhury, E. Nan, Y. Shen, Y. Cao, H. Zeng, W. Chen, G. Yuan, J. Dy, S. Ostadabbas, X. Zhang, D. Kaeli, E. Yeh, and Y. Wang, "Human cognition in machines: A unified perspective of world models," 2026. [Online]. Available: https://arxiv.org/abs/2604.16592

[7] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023, arXiv:2309.06180.

[8] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative LLM inference using phase splitting," in *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, arXiv:2311.18677.

[9] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024, arXiv:2401.09670.

[10] Y. Xiang, X. Li, K. Qian, W. Yu, E. Zhai, and X. Jin, "ServeGen: Workload characterization and generation of large language model serving in production," in *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2026, arXiv:2505.09999.

[11] F. Zeng, W. Gan, Y. Wang, N. Liu, and P. S. Yu, "Large language models for robotics: A survey," *arXiv preprint arXiv:2311.07226*, 2023.

[12] A. Taherin, J. Lin, A. Akbari, A. Akbari, P. Zhao, W. Chen, D. Kaeli, and Y. Wang, *Cross-Platform Scaling of Vision-Language-Action Models from Edge to Cloud GPUs*. New York, NY, USA: Association for Computing Machinery, 2026, p. 234–239. [Online]. Available: https://doi.org/10.1145/3787109.3816400

[13] J. Lin, A. Taherin, A. Akbari, A. Akbari, L. Lu, G. Chen, T. Padir, X. Yang, W. Chen, Y. Li, X. Lin, D. Kaeli, P. Zhao, and Y. Wang, "Vote: Vision-language-action optimization with trajectory ensemble voting," 2026. [Online]. Available: https://arxiv.org/abs/2507.05116

[14] C. Cui, Y. Ma, X. Cao, W. Ye, Y. Zhou, K. Liang, J. Chen *et al.*, "A survey on multimodal large language models for autonomous driving," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2024, pp. 958–979.

[15] H. Wang, W. Shao, C. Sun, K. Yang, D. Cao, and J. Li, "A survey on an emerging safety challenge for autonomous vehicles: Safety of the intended functionality," *Engineering*, vol. 33, pp. 17–34, 2024.

[16] B. G. Mohammed and D. S. Hasan, "Smart healthcare monitoring system using IoT," *International Journal of Interactive Mobile Technologies*, vol. 17, no. 1, pp. 141–152, 2023.

[17] G. Vardakis, G. Hatzivasilis, E. Koutsaki, and N. Papadakis, "Review of smart-home security using the Internet of Things," *Electronics*, vol. 13, no. 16, p. 3343, 2024.

[18] Z. Zhou, X. Ning, K. Hong, T. Fu, J. Xu, S. Li, Y. Lou, L. Wang, Z. Yuan, X. Li *et al.*, "A survey on efficient inference for large language models," *arXiv preprint arXiv:2404.14294*, 2024.

[19] S. Liu, K. Han, A. Fernandez-Lopez, A. K. Jaiswal, Z. Atashgahi, B. Wu, E. Ponti, C. Hao, R. Burkholz, O. Saukh, L. Yin, A. Zinonos, T. Huang, J. Tanner, and Y. Wang, "Edge-LLMs: Edge-device large language model competition," in *NeurIPS 2024 Competition Track*, 2024.

[20] NVIDIA, "Jetson agx xavier — developer kit and module spec," https://developer.nvidia.com/embedded/jetson-agx-xavier, 2024.

[21] ——, "Jetson agx orin — developer kit and module spec," https://developer.nvidia.com/embedded/jetson-agx-orin, 2024.

[22] ——, "Jetson agx thor — technical reference manual," https://developer.nvidia.com/embedded/jetson-thor, 2025.

[23] Q. Sun, A. Taherin, Y. Siatitse, and Y. Zhu, "Energy-efficient 360-degree video rendering on fpga via algorithm-architecture co-design," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 97–103. [Online]. Available: https://doi.org/10.1145/3373087.3375317

[24] A. Taherin, M. Salehi, and A. Ejlali, "Reliability-aware energy management in mixed-criticality systems," *IEEE Transactions on Sustainable Computing*, vol. 3, no. 3, pp. 195–208, 2018.

[25] ——, "Stretch: exploiting service level degradation for energy management in mixed-criticality systems," in *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, 2015, pp. 1–8.

[26] J. Stojkovic, C. Zhang, Í. Goiri, J. Torrellas, and E. Choukse, "DynamoLLM: Designing LLM inference clusters for performance and energy efficiency," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2025, best Paper Award; arXiv:2408.00741.

[27] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training quantization for generative pre-trained transformers," in *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023.

[28] J. Lin, J. Tang *et al.*, "Awq: Activation-aware weight quantization for on-device llm compression and acceleration," *MLSys*, 2024.

[29] G. Xiao, J. Lin *et al.*, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *ICML*, 2023.

[30] C. Lee and Others, "Owq: Lessons learned from activation outliers for weight quantization in large language models," *CoRR*, 2023.

[31] J. Chee *et al.*, "Quip: 2-bit quantization of large language models with guarantees," *NeurIPS*, 2023.

[32] A. Tseng, J. Chee, Q. Sun, V. Kuleshov, and C. De Sa, "QuIP#: Even better LLM quantization with hadamard incoherence and lattice codebooks," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

[33] Z. Yao, R. Yazdani Aminabadi *et al.*, "Zeroquant: Efficient and affordable post-training quantization for large-scale transformers," *NeurIPS*, vol. 35, pp. 27 168–27 183, 2022.

[34] F. Tan, R. Lee, l. Dudziak, S. X. Hu, S. Bhattacharya, T. Hospedales, G. Tzimiropoulos, and B. Martinez, "Mobilequant: Mobile friendly quantization for on device language models," in *Findings of the Association for Computational Linguistics: EMNLP 2024*. Association for Computational Linguistics, 2024. [Online]. Available: https://aclanthology.org/2024.findings-emnlp.570/

[35] X. Shen, Z. Kong, C. Yang, Z. Han, L. Lu, P. Dong, C. Lyu, C.-h. Li, X. Guo, Z. Shu *et al.*, "Edgeqat: Entropy and distribution guided quantization-aware training for the acceleration of lightweight llms on the edge," *arXiv preprint arXiv:2402.10787*, 2024.

[36] C. Guo, J. Tang, W. Hu, J. Leng, C. Zhang, F. Yang, Y. Liu, M. Guo, and Y. Zhu, "OliVe: Accelerating large language models via hardware-friendly outlier-victim pair quantization," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[37] N. Dhar, B. Deng, D. Lo, X. Wu, L. Zhao, and K. Suo, "An empirical analysis and resource footprint study of deploying large language models on edge devices," in *Proceedings of the 2024 ACM Southeast Conference (ACMSE)*. ACM, 2024, pp. 69–76.

[38] Z. Nezami, M. Hafeez, K. Djemame, and S. A. R. Zaidi, "Generative AI on the edge: Architecture and performance evaluation," in *Proceedings of the IEEE International Conference on Communications (ICC)*. IEEE, 2025, arXiv:2411.17712.

[39] M. Ardakani, J. Malekar, and R. Zand, "LLMPi: Optimizing LLMs for high-throughput on raspberry pi," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2025, pp. 6379–6388.

[40] E. J. Husom, A. Goknil, M. Astekin, L. K. Shar, A. Kåsen, S. Sen, B. A. Mithassel, and A. Soylu, "Sustainable LLM inference for edge AI: Evaluating quantized LLMs for energy efficiency, output accuracy, and inference latency," *ACM Transactions on Internet of Things*, 2025.

[41] Y. Li, J. Liu, H. Zhang, M. B. Narayanan, U. Sharma, S. Zhang, Y. Zeng, J. Raghuram, and S. Banerjee, "PalmBench: A comprehensive benchmark of compressed large language models on mobile platforms," in *Proceedings of the 13th International Conference on Learning Representations (ICLR)*, 2025, arXiv:2410.05315. [Online]. Available: https://openreview.net/forum?id=xzSUdw6s76

[42] S. Laskaridis, K. Katevas, L. Minto, and H. Haddadi, "MELTing Point: Mobile evaluation of language transformers," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom)*, 2024, arXiv:2403.12844.

[43] D. Xu, W. Yin, H. Zhang, X. Jin, Y. Zhang, S. Wei, M. Xu, and X. Liu, "EdgeLLM: Fast on-device LLM inference with speculative decoding," *IEEE Transactions on Mobile Computing*, vol. 24, no. 4, pp. 3256–3273, 2025.

[44] A. Pinnock, S. Jayakody, K. A. Roxy, and M. R. Ahmed, "EdgeProfiler: A fast profiling framework for lightweight LLMs on edge using analytical model," in *Proceedings of the IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2025.

[45] S. Jang and R. Morabito, "Edge-first language model inference: Models, metrics, and tradeoffs," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2025.

[46] K. Alizadeh, I. Mirzadeh, D. Belenko, S. K. Khatamifard, M. Cho, C. C. Del Mundo, M. Rastegari, and M. Farajtabar, "LLM in a flash: Efficient large language model inference with limited memory," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024, arXiv:2312.11514.

[47] Z. Liu, C. Zhao, F. Iandola, C. Lai, Y. Tian, I. Fedorov, Y. Xiong, E. Chang, Y. Shi, R. Krishnamoorthi, L. Lai, and V. Chandra, "MobileLLM: Optimizing sub-billion parameter language models for on-device use cases," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024, arXiv:2402.14905.

[48] J. Yuan, C. Yang, D. Cai, S. Wang, X. Yuan, Z. Zhang, X. Li, D. Zhang, H. Mei, X. Jia, S. Wang, and M. Xu, "Mobile foundation model as firmware," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom)*, 2024, arXiv:2308.14363.

[49] S. P. Baller, A. Jindal, M. Chadha, and M. Gerndt, "DeepEdgeBench: Benchmarking deep neural networks on edge devices," in *IEEE International Conference on Cloud Engineering (IC2E)*, 2021, pp. 20–30.

[50] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau, U. Thakker, A. Torrini, P. Warden, J. Cordaro, G. Di Guglielmo, J. Duarte, S. Gibellini, V. Parekh, H. Tran, N. Tran, N. Wenxu, and X. Xuesong, "MLPerf tiny benchmark," *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[51] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "MLPerf inference benchmark," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[52] A. Tschand, A. T. R. Rajan, S. Idgunji, A. Ghosh, J. Holleman, C. Kiraly, P. Ambalkar, R. Borkar, R. Chukka, T. Cockrell, O. Curtis, G. Fursin, M. Hodak, H. Kassa, A. Lokhmotov, D. Miskovic, Y. Pan, M. P. Manmathan, L. Raymond, T. St.J̃ohn, A. Suresh, R. Taubitz, S. Zhan, S. Wasson, D. Kanter, and V. J. Reddi, "MLPerf power: Benchmarking the energy efficiency of machine learning systems from microwatts to megawatts for sustainable AI," *arXiv preprint arXiv:2410.12032*, 2025.

[53] D. Minott, S. Siddiqui, and R. J. Haddad, "Benchmarking edge AI platforms: Performance analysis of NVIDIA Jetson and Raspberry Pi 5 with Coral TPU," in *SoutheastCon 2025*. IEEE, 2025, pp. 1384–1389.

[54] C. Niu, W. Zhang, J. Li, Y. Zhao, T. Wang, X. Wang, and Y. Chen, "TokenPowerBench: Benchmarking the power consumption of LLM inference," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2026.

[55] S. Samsi, D. Zhao, J. McDonald, B. Li, A. Michaleas, M. Jones, W. Bergeron, J. Kepner, D. Tiwari, and V. Gadepally, "From words to watts: Benchmarking the energy costs of large language model inference," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, 2023, arXiv:2310.03003.

[56] K. T. Chitty-Venkata, S. Raskar, B. Kale, F. Ferdaus, A. Tanikanti, K. Raffenetti, V. Taylor, M. Emani, and V. Vishwanath, "LLM-Inference-Bench: Inference benchmarking of large language models on AI accelerators," in *2024 SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (PMBS)*, 2024, arXiv:2411.00136.

[57] F. Wu, Z. Bian, G. Duan, T. Xu, J. Wu, T. Ma, Y. Yao, R. Gong, and Y. Zhuo, "TokenSim: Enabling hardware and software exploration for large language model inference systems," in *Advanced Parallel Processing Technologies: 16th International Symposium (APPT)*. Springer, 2025.

[58] J. Cho, M. Kim, H. Choi, G. Heo, and J. Park, "LLMServingSim: A HW/SW co-simulation infrastructure for LLM inference serving at scale," in *2024 IEEE International Symposium on Workload Characterization (IISWC)*, 2024, arXiv:2408.05499.

[59] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020, pp. 38–45.

[60] G. Gerganov *et al.*, "llama.cpp: Llm inference in c/c++," https://github.com/ggml-org/llama.cpp, 2023.

[61] ggml-org, "GGML: Tensor library for machine learning," https://github.com/ggml-org/ggml, 2024.

[62] NVIDIA, "tegrastats utility — NVIDIA Jetson Linux developer guide," https://docs.nvidia.com/jetson/, 2024.

[63] ——, "NVIDIA Management Library (NVML)," https://developer.nvidia.com/management-library-nvml, 2024, accessed: 2026-05-18.

[64] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[65] K. Sakaguchi, R. Le Bras, C. Bhagavatula, and Y. Choi, "WinoGrande: An adversarial winograd schema challenge at scale," *Communications of the ACM*, vol. 64, no. 9, pp. 99–106, 2021.

[66] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi, "HellaSwag: Can a machine really finish your sentence?" in *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.

[67] L. Gao, J. Tow, B. Abbasi, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, A. Le Noac'h, H. Li, K. McDonell, N. Muennighoff, C. Ociepa, J. Phang, L. Reynolds, H. Schoelkopf, A. Skowron, L. Sutawika, E. Tang, A. Thite, B. Wang, K. Wang, and A. Zou, "A framework for few-shot language model evaluation," https://github.com/EleutherAI/lm-evaluation-harness, 2024.

[68] J. Zhou, T. Lu, S. Mishra, S. Brahma, S. Basu, Y. Luan, D. Zhou, and L. Hou, "Instruction-following evaluation for large language models," *arXiv preprint arXiv:2311.07911*, 2023.

[69] C.-P. Hsieh, S. Sun, S. Kriman, S. Acharya, D. Rekesh, F. Jia, Y. Zhang, and B. Ginsburg, "RULER: What's the real context size of your long-context language models?" in *Conference on Language Modeling (COLM)*, 2024.

[70] H. Fan, Y.-C. Lin, and V. Prasanna, "ELLIE: Energy-efficient LLM inference at the edge via prefill-decode splitting," in *2025 IEEE 36th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2025, pp. 139–146.

[71] A. K. Kakolyris, D. Masouros, P. Vavaroutsos, S. Xydis, and D. Soudris, "throttLL'eM: Predictive GPU throttling for energy efficient LLM inference serving," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 1363–1378.

[72] Y. Zheng, Y. Chen, B. Qian, X. Shi, Y. Shu, and J. Chen, "A review on edge large language models: Design, execution, and applications," *ACM Computing Surveys*, 2024, arXiv:2410.11845.

[73] V. Egiazarian, A. Panferov, D. Kuznedelev, E. Frantar, A. Babenko, and D. Alistarh, "Extreme compression of large language models via additive quantization," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, ser. PMLR, vol. 235, 2024, pp. 12 284–12 303, arXiv:2401.06118.

[74] A. Ramachandran, S. Kundu, and T. Krishna, "MicroScopiQ: Accelerating foundational models through outlier-aware microscaling quantization," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, 2025.

[75] X. Shen, P. Dong, L. Lu, Z. Kong, Z. Li, M. Lin, C. Wu, and Y. Wang, "Agile-quant: Activation-guided quantization for faster inference of llms on the edge," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, 2024, pp. 18 944–18 951.

ARTIFACT APPENDIX

A.1 Abstract

This artifact provides ***two*** research objects:

1) **Hydra (code)**: the common-schema, cross-backend profilers (HuggingFace Transformers and `llama.cpp`), the `tegrastats`/NVML fusion pipeline, the canonical cross-platform schema, and the analysis pipeline.
2) **The Hydra corpus (dataset)**: the complete released measurement corpus of 107,110 per-prompt records (286 unified CSVs: 190 main-corpus configurations plus the S1/S2 length-sensitivity sweeps), shipped inside the repository as a split `xz` tarball.

The artifact is assessed through ***two*** evaluations with distinct goals:

1) **Evaluation I** reproduces every computational result of the paper (Figs. 1, 3–7 and Tables 4–7) from the released corpus on any Linux or macOS machine—no GPU or Jetson hardware required.
2) **Evaluation II** validates the measurement pipeline itself: it re-measures the paper's flagship model on the three Jetson testbeds (SSH access provided) and compares fresh measurements against the corpus.

A.2 Artifact Meta-Information Checklist

- **Program:** Hydra profilers (Python + C++), unifier, analysis pipeline (Python).
- **Dataset:** 107,110 per-prompt records; 25 timing fields + phase-attributed telemetry aggregates (349–419 columns, depending on platform/backend).
- **Hardware (Eval. I):** any Linux or macOS machine (x86-64 or ARM); no GPU. Can also run on the provided boards (never concurrently with an Evaluation II measurement), though a separate machine is preferred.
- **Hardware (Eval. II):** NVIDIA Jetson AGX Xavier / Orin / Thor; SSH access to all three provided during evaluation.
- **Software (Eval. I):** Python $\geq$3.10 with `pandas`, `numpy`, `matplotlib`, `seaborn`.
- **Software (Eval. II):** pre-provisioned on the boards; per-platform Python/PyTorch stacks are documented in `docs/ENVIRONMENT.md`.
- **Metrics:** latency (TTFT, ITL, E2E), throughput, GPU/CPU utilization, DRAM bandwidth, power, energy/token, temperature.
- **Output:** paper Figs. 1, 3–7 (PDF) and Tables 4–7 (`md`/`csv`); spot-check comparison report.
- **Disk space:** $\sim$1 GB (Eval. I).
- **Time:** Eval. I $\sim$5 min; Eval. II $\sim$0.5–1 h per board (longer on Xavier).
- **Publicly available:** yes (GitHub + archival DOI).
- **Badges applied for:**
  - **Datasets**: Available, Reviewed, Reproducible (all via Eval. I)
  - **Code** Available (Eval. I), Reviewed (Eval. I: analysis pipeline; Eval. II: measurement pipeline), Reproducible (Eval. II).

A.3 Access

Public artifact repository: https://github.com/amirtaherin/hydra. Archival copy: Zenodo, DOI 10.5281/zenodo.21844843. The corpus ships *inside* the repository under `data/unified/`; the expected outputs ship with the repository under `expected_results/`; per-platform environment recipes and the archived PyTorch wheels are documented in `docs/ENVIRONMENT.md`.

A.4 Evaluation I: Reproducing the Paper's Results (no Jetson GPU needed)

Goal: regenerate every computational result of the paper from the released corpus, demonstrating that the **dataset** is complete and that the **analysis pipeline** (corpus loader, canonical cross-platform schema, and the figure and table generators) runs and reproduces the published figures and tables. This evaluation supports the *Available* and *Reviewed* badges for both research objects and the *Reproducible* badge for the dataset.

All steps from scratch, on any Linux or macOS machine:

```
$ git clone https://github.com/amirtaherin/hydra.
      git
$ cd hydra
$ python3 -m venv .venv && . .venv/bin/activate
$ pip install pandas numpy matplotlib seaborn
$ bash scripts/ae_reproduce.sh
```

The driver first **verifies the corpus**: a sha256 check of the release tarball, then an integrity manifest—107,110 per-prompt records in 286 unified CSVs (190 main-corpus configurations plus the S1/S2 sweeps)—that hard-stops on any mismatch. It then renders Figs. 1 and 3–7 and regenerates Tables 4–7, emitted both as `.md` (for visual comparison against the published tables) and `.csv` (machine-readable). All regenerated figures and tables are written to `ae_output/` (`figures/` and `tables/` subdirectories).

Expected outcome: *(i)* the manifest prints `CORPUS VERIFICATION PASSED` (automated); *(ii)* six figure PDFs render (observed); *(iii)* the driver's final step reports that every value in the regenerated tables is identical to the reference tables shipped in `expected_results/` (automated). The same check can be run by hand:

```
$ diff -r ae_output/tables \
          expected_results/tables
```

`diff` *prints nothing when the tables are identical*—empty output is the expected success result. The whole evaluation takes about five minutes on a laptop. *On the provided boards*, skip the clone/venv/pip steps above—the pre-installed environment activates on login and `~/hydra` is already checked out; simply run the driver. Do NOT run Evaluation I on a board while an Evaluation II measurement is in progress there: the analysis load perturbs the telemetry being recorded.

A.5 Evaluation II: Validating the Measurement Pipeline (Jetson testbeds, SSH)

Goal: certify that the **measurement pipeline**—the HuggingFace and `llama.cpp` profilers, the telemetry collection, and the timing+telemetry unifier—works end-to-end on real hardware, by re-measuring the paper's flagship model and comparing fresh measurements against the released corpus across all three SoC generations. This evaluation supports the *Reviewed* and *Reproducible* badges for the code.

(a) Spot-check on the provided boards. Everything is already set up on the boards: the Python environment activates at login, the profiler is pre-compiled, and the models are pre-downloaded; the login banner summarizes these steps:

```
$ ssh <board>        # credentials via HotCRP
$ cd ~/hydra
$ tmux new -s ae     # keeps the run alive if SSH
      drops
$ bash scripts/ae_quick.sh
```

(`tmux` basics: detach with `Ctrl-b` then `d`; reattach later with `tmux attach -t ae`; the run continues while detached.) The spot-check measures Qwen2.5-7B under HF `bf16` and `llama.cpp` `Q8_0`/`Q6_K`/`Q4_K_M` (∼20 IFEval prompts, 500-token decode), fuses timing with telemetry, and prints the fresh decode power, ITL, energy/token, and throughput next to the corresponding values from the released corpus.

Expected outcome: the run produces a fresh mini-corpus (20 prompts × 4 configurations, unified exactly like the released corpus) and a comparison table that ends in `Result: INVARIANTS PASS`. Success is judged by the two reproducibility checks described in § A.6. A run takes about 30–60 minutes per board; Xavier, the oldest platform, takes somewhat longer.

(b) Rebuilding the profiler from source (to verify the build, or on one's own Jetson). This is the only step that requires the pinned `llama.cpp` submodule:

```
$ git submodule update --init --recursive
$ bash scripts/build_llamacpp_profiler.sh
$ bash scripts/ae_quick.sh   # re-run with the
      rebuilt binary
```

The build script detects the platform and selects the CUDA architecture (Xavier `sm_72`, Orin `sm_87`, Thor `sm_110`); it requires `cmake`, the CUDA toolkit (`nvcc` on `PATH`), and a C++17 compiler. The repository README documents overrides and troubleshooting; `docs/ENVIRONMENT.md` gives the full per-board environment recipes. Full-scale collection (`scripts/run_{hf,llamacpp}_experiments.sh`) takes days of device time and is not expected of reviewers.

A.6 Interpreting Results

For **Evaluation I**, the driver performs two automated checks. It first verifies the released corpus itself (the checksum and record manifest of § A.4), and in its final step it verifies that the regenerated tables are identical to the reference tables shipped in `expected_results/`—this must hold exactly, since the analysis pipeline is deterministic. Two further comparisons are manual, made by the reviewer against the paper: the regenerated `.md` tables can be compared with the published Tables 4–7—they match at the printed precision, though in a handful of cells the last printed digit differs by one (e.g., 86.3 vs. 86.2) due to rounding during manuscript preparation—and the generated figures should look identical to the published Figs. 1 and 3–7 (same scripts, same data; reference copies in `expected_results/figures/`; compared visually, not with `diff`, since PDF files are never byte-identical across systems).

In **Evaluation II**, the comparison applies two reproducibility checks to the freshly measured mini-corpus. *(1) Quantitative agreement:* every fresh value is printed next to its corpus reference with the percentage deviation; values typically fall within $\pm 15\%$ of the corpus means (thermal state and background load shift absolute numbers, so this band is guidance, not a hard gate). *(2) Qualitative findings*—the binding pass/fail criterion, printed as `Qualitative invariants`: three findings of the paper must reproduce on every board:

- `Q6_K` draws more decode power than `Q8_0` (bit-width non-monotonicity, §V-C);
- `Q4_K_M` has the lowest energy per token among the `llama.cpp` formats (energy efficiency, §V-C);
- `llama.cpp` ITL < HF ITL (runtime-overhead gap, §V-A).

Together, the two evaluations exercise the artifact's full pipeline: Evaluation II covers the collection and fusion stages of Fig. 2, and Evaluation I the analysis stage. The remaining figure and tables are not covered because they are not produced by Hydra: Fig. 2 is a drawn diagram, Tables 1, 2, and 8 are hand-written summaries, and Table 3 comes from the public `lm-eval-harness` tool, independent of the telemetry corpus.

A.7 Customization

The repository follows the paper's architecture: `inputs/` (prompts, model registry), `inference/` (HF profiler, `llama.cpp` profiler, telemetry collection), and `analysis/` (unifier, figure and table generators). New models, precisions, or prompt sets require only edits under `inputs/`; new platforms require a `tegrastats` parser (`inference/telemetry/`). Beyond the paper's figures, `analysis/main.py` exposes further plot families (distributions, scaling, thermal, memory pressure) over the same corpus, e.g.:

```
$ python3 -m analysis.main --all \
    --results-root <corpus> --out figs/
```