Performance analysis of localised large language models in resource-constrained edge for Python and Rust APIs

Partha Pratim Ray[1], Mohan Pratap Pradhan[1]

[1]*Department of Computer Applications, Sikkim University, 6th Mile, Samdur, Gangtok, 737102, India*

Abstract. Edge deployments of large language models (LLMs) often suffer from significant latency due to the overhead of high-level client runtimes on resource-constrained hardware. To address this challenge, we conducted a side-by-side performance analysis of four quantised LLMs – Llama 3.2:1b, Gemma 3:1b, Granite 3.1-MoE:1b, and Qwen 2.5:0.5b – on a Raspberry Pi 4 Model B (8 GB LPDDR4, quad-core ARM Cortex-A72) using both Python and Rust API clients. Each model was served via a local Ollama inference server, and a fixed suite of twenty prompts – covering factual retrieval, arithmetic reasoning, translation, code synthesis, and creative generation – was executed sequentially with a two-second inter-request delay, yielding 160 measurements per client. Rust markedly reduces cold-start delays: mean model load times fall from 1 648.7 ms (Python) to 52.8 ms (Rust) for Llama 3.2:1b, and from 607.0 ms to 171.3 ms for Qwen 2.5:0.5b. Corresponding end-to-end latencies decrease by 1.4-2.0 s across models. In warm-start conditions, both clients deliver nearly identical decoding throughput – ≈2.7 tokens/s for Llama 3.2:1b, 4.4 tokens/s for Gemma 3:1b, 7.4 tokens/s for Granite 3.1-MoE, and 8.6 tokens/s for Qwen 2.5:0.5b – indicating that runtime overhead is negligible once models are loaded. Rigorous statistical testing, including paired t-tests, Mann-Whitney U tests, and bootstrap confidence intervals, confirms that Rust's cold-start advantages are highly significant ($p < 0.01$). At the same time, throughput differences in steady-state inference are not statistically meaningful. We discuss limitations in platform specificity, quantisation approaches, and prompt diversity, and outline future work on heterogeneous accelerators, adaptive scheduling, and on-device fine-tuning. Finally, we highlight practical applications in smart agriculture, healthcare monitoring, industrial IoT, autonomous robotics, and offline educational tools. This benchmark furnishes actionable guidelines for selecting client languages and quantised models in edge AI scenarios.

Keywords: edge computing, large language models, Python client, Rust client, latency, performance analysis

1. Introduction

The rapid proliferation of LLMs has revolutionised natural language processing across a broad spectrum of applications, from conversational agents to code generation [21]. However, deploying these models on resource-constrained edge devices poses formidable challenges. Chief among these is the performance overhead introduced by high-level, interpreted languages such as Python [6, 13, 19]. Despite Python's extensive ecosystem and developer productivity advantages, its reliance on a global interpreter lock, dynamic typing, and memory-managed runtime leads to non-negligible latency in HTTP request handling, JSON serialisation, and file I/O operations [3, 25]. When inference demands millisecond-scale responsiveness – such as in real-time sensor

0000-0003-2306-2792 (P. P. Ray); 0009-0007-8731-764X (M. P. Pradhan)
pprau@cus.ac.in (P. P. Ray); mppradhan@cus.ac.in (M. P. Pradhan)
https://cus.ac.in/index.php/en/department-of-computer-applications/partha-pratim-ray (P. P. Ray); https://cus.ac.in/index.php/en/schools-e/physical-sciences/computer-applications-dept/dr-mohan-pratap-pradhan (M. P. Pradhan)

data interpretation or autonomous control loops – Python's runtime overhead can dominate the end-to-end latency, undermining the very promise of localised, low-latency intelligence.

In contrast, systems-level languages like Rust offer a compelling alternative for edge deployments [31, 37]. Rust compiles to native code, eliminates runtime garbage collection, and provides zero-cost abstractions that translate into minimal overhead for critical operations. Its ownership and borrowing model enforces memory safety without sacrificing performance, enabling developers to write concurrent, asynchronous code with confidence [22, 43]. In the context of LLM inference, a Rust client can significantly reduce cold-start penalties by streamlining binary loading, JSON parsing, and file writes, while preserving the deterministic throughput of the underlying model. Furthermore, Rust's 'reqwest' library [39], combined with 'serde' [41] for serialisation, achieves efficient HTTP transactions and zero-copy JSON deserialization, delivering performance that approaches bare-metal C implementations yet retains the developer ergonomics needed for complex integration tasks.

Despite these theoretical advantages, there has been no systematic, empirical evaluation comparing Python and Rust clients for quantised LLM inference on genuine edge hardware [27, 29, 52]. Existing benchmarks often focus solely on model-level performance – such as tokens per second under GPU acceleration – or contrast interpreted versus compiled environments in synthetic workloads. To our knowledge, this work represents the first comprehensive side-by-side analysis of Python and Rust API clients driving Ollama-hosted [34], quantized LLMs on a Raspberry Pi 4 Model B. By issuing a standardized set of twenty prompts spanning factual queries, arithmetic reasoning, translation, code synthesis, and creative generation, we obtain a granular view of both cold-start and warm-start performance across four distinct quantized architectures: Llama 3.2:1b (8-bit)[1], Gemma 3:1b[2] (mixed 4/8-bit), Granite 3.1-MoE:1b[3] (8-bit Mixture-of-Experts), and Qwen 2.5:0.5b[4] (mixed 4/8-bit). Our methodology isolates client-induced latency from model inference time, allowing for a clear attribution of performance differences to language runtime characteristics.

The primary contributions of this work are as follows:

- A reproducible benchmarking suite for quantised LLM inference on Raspberry Pi, capturing end-to-end latency, model load time, prompt eval duration, full evaluation duration, token counts, and decoding throughput.
- Empirical evidence that Rust clients reduce cold-start load times by up to 90% (e.g., from 1 648.7 ms in Python to 52.8 ms in Rust for Llama 3.2:1b), significantly mitigating initialisation delays.
- Confirmation that, once models are warmed, both Python and Rust APIs deliver equivalent decoding performance – approximately 2.7 tokens/s for Llama 3.2:1b – indicating negligible runtime overhead during sustained inference.
- Comprehensive statistical validation – including paired t-tests, Mann-Whitney U tests, bootstrap confidence intervals, and nonparametric clustering – that substantiates the robustness of performance differences and reveals distinct regimes across prompts and models.

2. Related work

The rapid advancement of LLMs has inspired numerous applications and tools, particularly in the Rust ecosystem; yet, few address performance trade-offs in resource-

[1] https://ollama.com/library/llama3.2:1b
[2] https://ollama.com/library/gemma3:1b
[3] https://ollama.com/library/granite3.1-moe:1b
[4] https://ollama.com/library/qwen2.5:0.5b

constrained edge deployments. Chu et al. [9] introduce PALM, a hybrid program analysis and LLM-driven approach to boost Rust unit test coverage. While PALM achieves impressive coverage improvements through path-constraint-guided prompts, it does not consider inference performance or client runtime overhead on embedded devices. Zhang et al. [55] present RustC4, an LLM-assisted tool for detecting code-comment inconsistencies via program analysis and natural language understanding. Although RustC4 highlights the synergy between static analysis and LLM reasoning, it similarly overlooks the impact of API client implementation on runtime latency in IoT or edge contexts.

Luo et al. [30] propose HALURust, which exploits LLM hallucinations to detect vulnerabilities in Rust by fine-tuning on generated hallucinated reports. HALURust demonstrates strong generalisation across unseen vulnerabilities, yet it focuses on security analysis rather than inference efficiency. Yang et al. [49] develop VERT, a formally verified transpilation framework that combines LLM-generated Rust and WebAssembly-derived oracles to ensure correctness. VERT's guarantee of semantic equivalence through model checking is orthogonal to concerns about cold-start or sustained throughput on low-power hardware.

Efforts to integrate LLMs with formal verification include Yao et al. [50], who combine GPT-4 with static analysis to synthesise proofs in Rust's Verus framework, and Deligiannis et al. [14], who leverage LLMs to fix Rust compilation errors automatically. Both demonstrate the practical power of LLMs in correctness tasks, but neither evaluates runtime trade-offs between high-level vs. system-level client runtimes under memory and CPU constraints.

Benchmarking challenges in Rust code generation are addressed by Liang et al. [28] with RustEvo^2, a dynamic API evolution evaluation suite that measures LLM adaptability to version changes. While RustEvo^2 uncovers significant degradation in behavioural changes, it does not measure latency or resource overhead on edge platforms. Gao et al. [18] propose ClozeMaster, which utilises masked code infilling to fuzz the Rust compiler. Their focus on synthesising valid test cases for compiler robustness differs from our interest in inference path performance.

Transpilation from C to Rust has been explored by Nitin et al. [32] with C2SaferRust and by Shetty et al. [42] with Syzygy, both blending rule-based translation, LLM refinement, and dynamic analysis to produce safe Rust. These approaches yield more idiomatic and verified code, but do not study how Python and Rust clients compare in calling local inference servers for quantised models on constrained hardware.

Scaling LLM-driven translation to large codebases is tackled by Zhang et al. [53], who partition real-world projects into fragments for validated translation via feature mapping and type compatibility. Their focus is on translation correctness at scale rather than runtime characteristics of API clients. Cheng et al. [7] develop RUG, an end-to-end Rust unit test generator that combines coverage-guided fuzzing with LLM context construction, yet performance metrics are reported only in terms of coverage and compilation success, not inference latency on embedded devices.

Eniser et al. [17] present FLOURINE, a differential fuzzing framework to validate LLM translations of Go to Rust at the function level, employing counterexample-driven feedback loops. FLOURINE's insights into translation accuracy complement our concerns with inference speed and cold-start overhead. Finally, Wu et al. [46] introduce PseudoEval, a pseudocode-to-code benchmark that isolates problem-solving from language-coding capabilities, revealing that LLMs struggle more with Rust syntax than Python. Although PseudoEval highlights language-specific bottlenecks, it does not examine how client implementations affect end-to-end performance in resource-limited edge contexts.

Summarising all related works leads to the following. Existing works span test

generation [9], [7], code-comment consistency [55], vulnerability detection [30], formal verification [50], compilation error fixing [14], application programming interface (API) evolution benchmarking [28], compiler fuzzing [18], C-to-Rust translation [32], [42], large-scale translation validation [53], [17], and pseudocode-based evaluation [46]. However, none directly address the nuanced performance comparison of Python versus Rust API clients driving quantised LLM inference on truly constrained edge hardware, such as a Raspberry Pi 4 B, via localised inference servers.

2.1. Research gap

While prior studies have leveraged large language models for tasks such as automated testing, code translation, and vulnerability detection in Rust [9, 30, 32, 42, 50], they have largely focused on accuracy, correctness guarantees, or code quality rather than end-to-end inference performance under tight resource constraints. Similarly, benchmarks for LLM throughput and latency typically assume server-grade graphics processing units (GPUs) or high-memory instances, and recent efforts to evaluate API evolution or pseudocode translation [28, 46] overlook the impact of client-side runtime choice on real-world, on-device deployments. In particular, no existing work systematically measures how the choice of a high-level interpreted language (i.e. Python) versus a systems-level language (i.e. Rust) affects cold-start overhead, per-prompt latency, or sustained tokens-per-second throughput when driving quantised LLMs through a local representational state transfer (REST) API on an 8 GB Raspberry Pi 4 B. This absence of empirical evidence leaves practitioners without concrete guidance for selecting client implementations in latency-sensitive edge AI scenarios. Table 1 shows a comparison of related works.

None of the surveyed works simultaneously address localised LLM deployment, edge computing scenarios, resource-limited environments, comprehensive performance analysis, or comparative evaluation of Python versus Rust client implementations. This work is the first to integrate all five dimensions in a single empirical study.

2.2. Novelty of this work

To address this gap, our study presents the first comprehensive, side-by-side evaluation of Python and Rust API clients for quantised LLM inference on a genuinely resource-constrained device. By deploying four representative one billion-parameter models (Llama 3.2:1b, Gemma 3:1b, Granite 3.1-MoE:1b, Qwen 2.5:0.5b) via Ollama's local server on a Raspberry Pi 4 B and issuing a standardised suite of twenty prompts across five task categories, we obtain 160 detailed observations that isolate client-induced overhead from core model evaluation. Our novel contributions include (i) quantifying Rust's dramatic reduction in cold-start model-load times – up to 90 % faster than Python – and (ii) demonstrating that once warmed, both clients achieve statistically indistinguishable tokens-per-second rates, underscoring minimal runtime overhead. Through rigorous statistical testing (paired t-tests, Mann-Whitney U test, bootstrap CIs, MANOVA) and clustering analyses, we furnish actionable guidelines for client-language selection, thereby enabling practitioners to optimise latency and throughput in localised LLM deployments on edge hardware.

3. Tools and methodology

3.1. Experimental setup and toolchain

All experiments were conducted on a Raspberry Pi 4 Model B equipped with 8 GB of LPDDR4 RAM and a quad-core ARM Cortex-A72 CPU running a 64-bit Linux distribution. This configuration represents a realistic edge-device environment where both memory capacity and compute throughput are tightly constrained compared to server-grade hardware. We selected the Raspberry Pi 4 B for its wide availability, low

Table 1
Comparison of related works.

Paper	Key contributions	Key limitations
Chu et al. [9]	Hybrid program analysis + LLM prompts for Rust unit tests	Relies on fixed prompts; no on-device or runtime benchmarking
Zhang et al. [55]	RustC4: LLM-driven code-comment inconsistency detection	Focus on documentation accuracy; omits performance metrics
Luo et al. [30]	HALURust: uses LLM hallucinations to detect Rust vulnerabilities	Depends on hallucination calibration; no edge deployment study
Yang et al. [49]	VERT: formally verified Rust transpilation combining WASM oracle + LLM	Emphasis on correctness; no client-side performance analysis
Yao et al. [50]	LLM + static analysis for Rust proof synthesis (Verus)	Prototype-level; no measurement of inference latency or resource usage
Deligiannis et al. [14]	RustAssistant: LLM-based compilation error fixes	Focus on fix accuracy; lacks runtime or edge evaluation
Liang et al. [28]	RustEvo: benchmark for Rust API evolution in code generation	Does not consider inference performance or localized deployment
Gao et al. [18]	ClozeMaster: LLM-infilling for Rust compiler fuzzing	Emphasizes code coverage; no client runtime profiling
Nitin et al. [32]	C2SaferRust: neuro-symbolic C→Rust translation improving safety	Ignores inference latency and memory constraints
Shetty et al. [42]	Syzygy: LLM + dynamic analysis for C→safe Rust translation	Scalable correctness; omits on-device performance measurements
Zhang et al. [53]	Modular LLM + rule-based code translation with I/O validation	Focus on translation accuracy; no edge or runtime study
Cheng et al. [7]	RUG: semantic-aware, coverage-guided LLM unit-test generation for Rust	Evaluates coverage only; no resource-constrained performance analysis
Eniser et al. [17]	FLOURINE: differential fuzzing + LLM for Go→Rust translation validation	Prioritizes translation correctness; ignores latency and memory usage
Wu et al. [46]	PseudoEval: isolates problem-solving vs. coding in LLM benchmarks	Benchmark design only; does not address edge or resource constraints
This work	Side-by-side benchmarking of four quantised LLMs on a Raspberry Pi 4 Model B via Python and Rust clients, including cold vs. warm start analysis, tokens/s throughput, and rigorous statistical validation.	Restricted to Raspberry Pi 4 hardware and Ollama server; limited to four quantised models and twenty prompts; no evaluation of output quality or alternative accelerators.

cost, and community support for containerised and native application deployment. At the core of our setup is the Ollama inference server, which hosts four quantised LLMs – Llama3.2:1b, Gemma3:1b, Granite3.1-MoE:1b, and Qwen2.5:0.5b – exposed via a local REST endpoint at *http://localhost:11434/api/generate*. By running Ollama locally, we eliminate network variability and focus exclusively on end-to-end inference performance on the Pi. The server was installed according to official Ollama documentation,

utilising Docker containers to isolate model dependencies and ensure reproducibility. For the Python-based workload, we used the following standard libraries:

- *requests* provides a synchronous HTTP client for constructing and sending JSON payloads to the Ollama endpoint [38]. Its simplicity and ubiquity in the Python ecosystem make it ideal for lightweight benchmarking scripts;
- *datetime* enables ISO-8601 timestamp generation for each invocation, ensuring precise temporal metadata in the CSV log [12];
- *csv* facilitates structured, line-oriented logging of tabular performance data, which can be seamlessly imported into analysis tools such as pandas or R [11];
- *os* and *time* handle file existence checks, directory operations [35], and wall-clock timing [44] for measuring total latency and enforcing inter-request delays.

Parallel to the Python benchmark, we developed a Rust client to evaluate performance in a system-level language with a minimal runtime footprint. Running identical Python and Rust implementations on the same hardware with a consistent prompt workload allows for a clear separation of language-specific and runtime overheads from the actual model inference costs. This comparative approach underscores the impact of client-side tooling decisions on benchmarking outcomes in resource-constrained settings. The inclusion of both a high-level language, Python, and a system-level language, Rust, further demonstrates the adaptability of the methodology and serves as a practical reference for researchers evaluating language choices for deploying LLMs at the edge. The Rust implementation relies on:

- *reqwest* (*blocking* + *json*) features provides a synchronous, type-safe HTTP client that integrates seamlessly with Rust's ownership model and avoids the overhead of an async runtime;
- *serde* and *serde_json* offer zero-cost JSON serialisation and deserialization through automatically derived *Serialise* and *Deserialise* traits, enabling direct mapping between network messages and a Rust *OllamaResponse* struct;
- *chrono* supplies robust utilities for generating UTC timestamps in ISO-8601 format, analogous to Python's *datetime*, but with compile-time type safety [8];
- *std::fs::OpenOptions* and *std::io::Write* implement efficient, buffered file appends to *rust_ollama_log.csv*, ensuring that log writes do not buffer indefinitely in memory;
- *std::thread::sleep* and *std::time::Instant* provide precise delay and timing controls to replicate the two-second inter-request idle period, preventing I/O contention and thermal throttling on the CPU.

3.2. Quantized LLMs used

In this work, we evaluate four lightweight, quantised LLMs deployed via Ollama on a Raspberry Pi 4 B. Each model is chosen for its unique architecture, parameter count, quantisation scheme, and licensing terms, reflecting a spectrum of design trade-offs suitable for resource-constrained edge inference.

Llama3.2:1b is a community-driven release by Meta, built on the proven LLaMA architecture. With approximately 1.24 billion parameters, it employs 8-bit quantisation (Q8_0) to compress the weight matrices, resulting in a 1.3 GB memory footprint. This quantisation strikes a balance between preserving model accuracy and reducing RAM usage, making it feasible to load and run on the Pi's 8 GB of system memory. The model is distributed under the LLAMA 3.2 Community License Agreement (September 25, 2024) and Meta's Acceptable Use Policy, which together govern both academic and commercial usage while enforcing ethical standards.

Developed by AI21 Labs[5], Gemma3:1b offers a similarly sized parameter count

[5] https://www.ai21.com/

(1.0 billion) but opts for a more aggressive mixed-precision quantisation scheme (Q4_K_M), yielding an 815 MB disk size. By reducing most weights to 4 bits and selectively maintaining key matrices at 8 bits, Gemma3:1b targets minimal memory usage without compromising performance unduly. Its Terms of Use (last revised February 21, 2024) grant broad research and non-commercial rights, with specific provisions for distribution and derivative works.

Granite3.1-MoE:1b, an IBM implementation of a Mixture-of-Experts (MoE) architecture, contains 1.33 billion parameters. Using 8-bit quantisation (Q8_0) across all expert modules, the model occupies 1.4 GB on disk. The MoE design dynamically activates subsets of "expert" subnets per input token, potentially increasing inference latency variability but improving parameter efficiency. Licensed under the Apache License 2.0 (January 2004), Granite3.1-MoE allows unrestricted commercial and academic use, making it an attractive option for production deployments.

Qwen2.5:0.5b is a compact, 494 million-parameter model by Qwen, also leveraging mixed-precision quantisation (Q4_K_M) to achieve a 398 MB footprint. Its architecture strikes a balance between minimal resource consumption and robust language understanding across a wide range of tasks. The model is distributed under the Apache License 2.0, allowing for flexible integration in both open-source and proprietary systems. Its smaller size yields faster load times and higher tokens-per-second throughput on the Pi, at the potential cost of reduced generative richness compared to larger counterparts.

These four quantised models represent distinct points in the design space of edge-optimised LLMs, varying in parameter scale, quantisation granularity, and licensing constraints. By benchmarking their performance on identical prompt workloads via both Python and Rust APIs, we isolate the impacts of architecture, precision reduction, and runtime environment on inference latency, memory usage, and token throughput. Table 2 compares these LLMs.

Table 2
Comparison of quantised LLMs deployed on Raspberry Pi.

Model	Architecture	Parameters	Quantisation	Size	License
Llama3.2:1b	LLaMA	1.24 B	Q8_0	1.3 GB	LLAMA 3.2 Community & AUP
Gemma3:1b	Gemma3	1.00 B	Q4_K_M	815 MB	Gemma terms of use (Feb 21, 2024)
Granite3.1-MoE:1b	Granite MoE	1.33 B	Q8_0	1.4 GB	Apache license 2.0 (Jan 2004)
Qwen2.5:0.5b	Qwen2	0.494 B	Q4_K_M	398 MB	Apache license 2.0

3.3. Problem definition

In modern edge-computing scenarios, deploying LLMs such as Ollama on resource-constrained devices (e.g., Raspberry Pi 4 B) poses significant challenges in balancing inference latency, throughput, and system stability. Let $\mathcal{D}$ denote such a device with limited CPU cores and no GPU acceleration. Our goal is to characterize the performance of a locally hosted LLM M under a variety of sampling temperatures $\tau \in [0, 1]$ and prompt inputs p. Formally, let $\mathcal{P} = \{(p_i, \tau_i)\}_{i=1}^{N}$ be a finite set of N input pairs, where each p_i is a natural-language prompt and τ_i the temperature parameter controlling the stochasticity of the model's output distribution.

For each invocation i, the model produces a response r_i along with timing metrics

(1):

$$d_i^{\text{total}}, \quad d_i^{\text{load}}, \quad c_i^{\text{prompt}}, \quad d_i^{\text{prompt}}, \quad c_i^{\text{eval}}, \quad d_i^{\text{eval}}. \tag{1}$$

here:

- d_i^{total} is the total wall-clock latency for the HTTP transaction;
- d_i^{load} is the time to load the model into memory (if not already resident);
- c_i^{prompt} and d_i^{prompt} are the token count and duration for initial prompt processing;
- c_i^{eval} and d_i^{eval} are the token count and duration for full model evaluation.

We define the tokens-per-second throughput for invocation i as (2)

$$\mu_i = \begin{cases} \dfrac{c_i^{\text{eval}}}{d_i^{\text{eval}}} \times 10^9, & d_i^{\text{eval}} > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Our objective is to design an automated measurement algorithm that issues each (p_i, τ_i) sequentially, collects $\{d_i^{\text{total}}, d_i^{\text{load}}, c_i^{\text{prompt}}, d_i^{\text{prompt}}, c_i^{\text{eval}}, d_i^{\text{eval}}, \mu_i\}$, and persists these metrics in a CSV file for downstream statistical analysis.

Crucially, the pattern of invocations must minimise interference effects such as CPU contention or file-system locking. To this end, we introduce a fixed inter-request delay $\Delta = 2$ s between consecutive calls. Let t_i denote the start time of the ith invocation; then (3)

$$t_{i+1} \;\geq\; t_i + d_i^{\text{total}} + \Delta. \tag{3}$$

This ensures that each request begins only after the previous one has fully completed and the system has had a moment to recover. From a systems perspective, the Raspberry Pi 4 B can be modelled as a server with service rate λ and processing capacity μ. Under the M/M/1 queuing approximation, the average response time $E[T]$ grows as (4).

$$E[T] \;=\; \frac{1}{\mu - \lambda}, \tag{4}$$

provided $\lambda < \mu$. Here, $\lambda = \frac{1}{\Delta + \bar{d}^{\text{total}}}$ is the arrival rate of prompts (one every $\Delta + \bar{d}^{\text{total}}$ seconds) and $\mu \approx \frac{1}{\bar{d}^{\text{total}}}$ is the service rate. While our system is not strictly M/M/1 – due to deterministic inter-request delays and variable service times – the model provides intuition on the impact of Δ and the distribution of d_i^{total}.

Formally, we pose the following measurement problem:

Given:

- A device $\mathcal{D}$ with fixed computational resources.
- An LLM M accessible via REST endpoint U.
- A sequence $\mathcal{P} = \{(p_i, \tau_i)\}_{i=1}^{N}$.
- A minimum inter-request delay Δ.

Compute: in (5)

$$\mathcal{M} = \{(d_i^{\text{total}}, d_i^{\text{load}}, c_i^{\text{prompt}}, d_i^{\text{prompt}}, c_i^{\text{eval}}, d_i^{\text{eval}}, \mu_i)\}_{i=1}^{N}, \tag{5}$$

such that each invocation respects the start-time constraint $t_{i+1} \geq t_i + d_i^{\text{total}} + \Delta$, and all metrics are recorded without loss.

The complexity of this measurement procedure comprises two parts: (i) the algorithmic overhead of constructing and parsing JSON payloads and writing to disk, which scales as $O(N)$, and (ii) the external latency of HTTP transactions and model inference, which dominates wall-clock time but remains linear in N. By formalising the measurement in this way, we establish a rigorous foundation for comparing Python and Rust implementations, quantifying both their computational footprints and their end-to-end performance on constrained edge hardware.

3.4. Prompt set and experimental justification

In this study, we employ a fixed set of twenty distinct natural-language prompts, each paired with a predefined temperature parameter, to systematically evaluate the inference performance of four quantised local LLMs – Llama3.2:1b, Gemma3:1b, Granite3.1-MoE:1b, and Qwen2.5:0.5b – running on a Raspberry Pi 4 B. Each prompt is assigned a unique identifier $i \in \{1, \ldots, 20\}$ to facilitate precise logging and subsequent analysis. By invoking this set of N=20 prompts sequentially in both Python and Rust API implementations, we generate a total of 4 models $\times$ 2 APIs $\times 20$ prompts =160 performance observations. This structured approach ensures a uniform workload across models and interfaces, enabling the direct comparison of per-prompt latency, model load time, token throughput, and overall system behaviour under constrained edge conditions.

The twenty prompts have been carefully chosen to span a broad spectrum of linguistic tasks and cognitive demands. They include simple factual queries (e.g., "Name any one river in India", Prompt 1; "What is the capital of Canada?", Prompt 7), basic arithmetic and sequence reasoning (e.g., "What is the square root of 144?", Prompt 11; "What comes next in the sequence: 2, 4, 8, 16,. . . ?", Prompt 16), translation tasks ("Translate 'peace' to French.", Prompt 3; "Translate 'thank you' to Spanish.", Prompt 17), code generation ("Write a Python function to add two numbers.", Prompt 5), and open-ended creative outputs ("Tell me a short joke.", Prompt 18). By covering factual retrieval, translation, mathematical reasoning, code synthesis, and creative generation, the prompt set exercises both the knowledge retrieval and generative capabilities of each LLM. This diversity is crucial for assessing how model architecture and quantisation affect performance across different NLP workloads. To probe the impact of stochastic sampling and output diversity, each prompt is coupled with a temperature value $\tau_i \in [0, 1]$. Lower temperatures (e.g., $\tau = 0.2$ for Prompts 1, 2, 7, 9, 11, 14, 19) favour deterministic, high-confidence responses, while higher temperatures (e.g., $\tau = 0.8$ or 0.9 for Prompts 10, 12, 18) encourage more varied outputs. This temperature variation allows us to observe how each quantised model's internal softmax sampling interacts with resource constraints. In particular, we can analyse whether increased randomness ($\tau \to 1$) leads to longer decoding times or higher token-processing overhead on the CPU-only Raspberry Pi, thereby revealing trade-offs between creativity and throughput.

Given the limited RAM (8 GB) and absence of GPU acceleration on the Raspberry Pi 4 B, prompt length and complexity must remain modest to avoid paging or memory thrashing. All selected prompts are concise – each fewer than twenty tokens – and avoid excessively long narrative inputs. This design ensures that model loading (d^{load}) and prompt evaluation (d^{prompt}) remain within the Pi's working set, preventing artificial inflation of measurements due to memory-management overhead. Moreover, the two-second inter-request delay Δ alleviates temporary CPU contention, ensuring that each invocation reflects steady-state performance rather than noisy transient spikes. The experimental methodology employs the same set of twenty prompts for each of the four quantised LLMs. Quantisation reduces model parameter precision to 4-bit or 8-bit representations, thereby reducing memory footprint and enabling deployment on edge devices. By holding the prompt set constant, we isolate the effects of quantisation level and model architecture on inference speed and resource consumption. In particular, comparisons between Llama3.2:1b (Q4_K_M, 398 MB) and Granite3.1-MoE:1b (Q8_0, 1.4 GB MoE) reveal how different memory and compute distributions translate into measurable load times and tokens-per-second.

We further replicate the entire benchmark in two API environments – Python (using the Requests library) and Rust (using Reqwest's blocking client). This dual-language

approach quantifies the overhead introduced by each runtime ecosystem. For each prompt i, we measure total latency d_i^{total}, inference sub-latencies (d_i^{load}, d_i^{prompt}, d_i^{eval}), and token throughput μ_i. The Python implementation captures these metrics via a CSV writer, whereas the Rust version uses direct file appends with synchronised writes. Differences in garbage collection, I/O buffering, and HTTP client performance between Python and Rust are thus revealed under identical model and prompt conditions.

Statistically, a sample size of twenty distinct prompts per model is sufficient to characterise central performance tendencies while limiting experimental duration. Assuming each prompt invocation takes on the order of one to five seconds, the whole 160-run suite completes within practical laboratory time constraints (< 15 minutes). This balance between sample breadth and experimental feasibility enables repeated trials for variance estimation without overburdening the edge device. Moreover, the choice of prompts ensures coverage of key NLP subtasks, facilitating generalizable insights into localised LLM behaviour in resource-constrained contexts. Thus, the twenty-prompt set represents a scientifically justified compromise between diversity and manageability. It exercises each quantised model's core inference pathways – fact retrieval, translation, reasoning, code generation, and creative text – under varying levels of stochasticity. By repeating the suite across four LLMs and two API implementations, we obtain a structured dataset of performance metrics that supports rigorous comparative analysis. This methodology establishes a replicable benchmark for future studies of localised LLMs on edge hardware and lays the groundwork for optimisations in model architecture, quantisation schemes, and API design. Table 3 shows all test prompts used in this study.

On the Raspberry Pi 4 B, both Python and Rust clients interact with a centralised Ollama Engine via its JSON/RESTful API. The engine, governed by shared configuration and policy modules, dispatches prompt requests to multiple localised small LLM instances and returns responses. Each client environment – Python virtual environment versus Rust Cargo – sends identical prompt payloads, measures end-to-end performance, and writes detailed logs (Py Log versus Rs Log). By comparing these logs, the framework isolates client-side overhead in request serialisation, HTTP handling, and deserialization, enabling precise analysis of latency, throughput, and load-time differences under resource constraints. All components run on limited CPU and memory resources. The proposed framework of comparison is shown in figure 1.

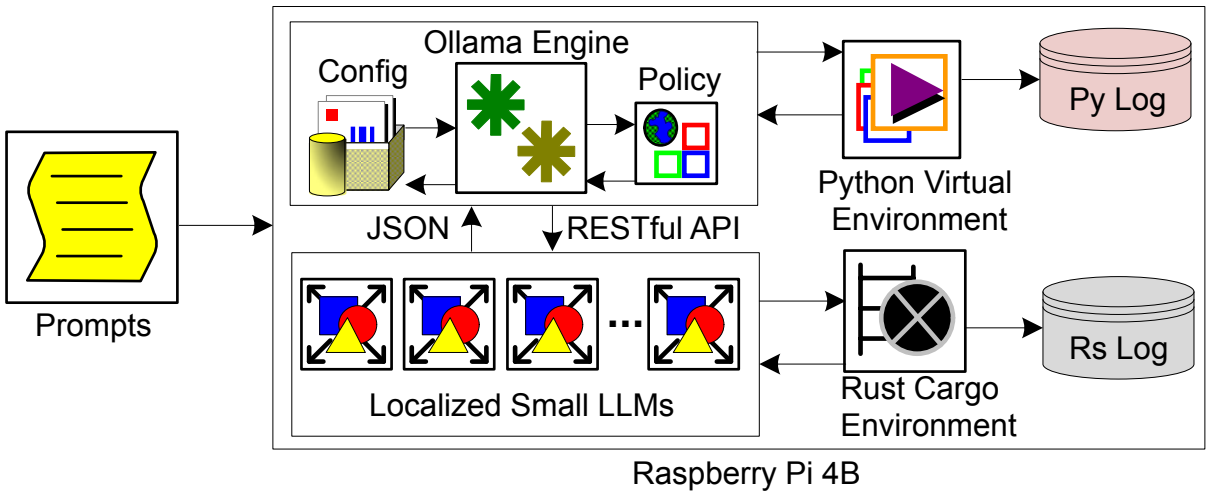


Figure 1: Proposed comparison framework.

Table 3
Prompt set with temperature and justification.

Prompt ID	Prompt	Temperature	Justification
1	Name any one river in India.	0.2	Tests factual recall and knowledge retrieval.
2	Who wrote the Indian National Anthem?	0.2	Verifies retrieval of specific historical fact.
3	Translate 'peace' to French.	0.3	Exercises translation capabilities.
4	Suggest a healthy snack for children.	0.7	Assesses creative suggestion under higher stochasticity.
5	Write a Python function to add two numbers.	0.5	Evaluates simple code synthesis.
6	Summarize the water cycle in one sentence.	0.4	Tests concise summarization ability.
7	What is the capital of Canada?	0.2	Checks retrieval of geographic information.
8	Name a fruit that is yellow.	0.3	Simple categorization and recall.
9	Who is known as the father of computers?	0.2	Validates retrieval of domain-specific attribution.
10	Give me a random English word.	0.8	Measures creative word generation.
11	What is the square root of 144?	0.2	Tests basic arithmetic reasoning.
12	Suggest a nickname for a friendly dog.	0.8	Assesses creative naming under high randomness.
13	Explain gravity to a child.	0.5	Evaluates ability to simplify complex concepts.
14	Which planet is called the Red Planet?	0.2	Checks common astronomy fact retrieval.
15	List any one prime number between 10 and 20.	0.2	Tests simple numerical reasoning.
16	What comes next in the sequence: 2, 4, 8, 16, ...?	0.3	Exercises pattern recognition and sequence prediction.
17	Translate 'thank you' to Spanish.	0.3	Further tests translation functionality.
18	Tell me a short joke.	0.9	Assesses humor generation at high temperature.
19	Who is the current UN Secretary-General?	0.2	Verifies up-to-date factual knowledge.
20	Complete: To be, or not to be, ...	0.4	Tests continuation and contextual completion.

4. Proposed algorithms

The Python API algorithm is designed to evaluate the inference performance of a locally hosted Ollama LLM on a Raspberry Pi by issuing sequential HTTP requests with varying temperature settings. For each prompt – temperature pair, it captures comprehensive metrics, such as model load duration, prompt evaluation time, and token throughput, from the JSON response. Outputs are sanitised and appended in real-time to a structured CSV file, preserving timestamps, prompt details, and all performance fields. A fixed inter-request delay and robust error handling prevent resource contention, ensuring that failures do not interrupt the benchmarking process. This automated logging mechanism provides a repeatable framework for statistical analysis of latency and throughput under different sampling conditions.

4.1. Python API algorithm

Algorithm 1 Python Ollama API performance logging.

Require: Ollama endpoint U, model name M, random seed s, prompt-temperature list $\mathcal{P} = \{(p_i, \tau_i)\}_{i=1}^{N}$, CSV file path F

Ensure: Append performance metrics for each prompt to F

Constants:

$$U = \text{OLLAMA\_URL},$$
$$M = \text{MODEL\_NAME},$$
$$s = \text{SEED},$$
$$\mathcal{P} = \{(p_i, \tau_i)\},$$
$$F = \text{CSV\_FILE}$$

Stage 1: Initialize CSV log
if not file_exists(F) **then**
    write_header(F)
end if
Stage 2: Iterate over prompts
for each $(p, \tau) \in \mathcal{P}$ **do**
    Build JSON payload

$$\pi = \{\, \text{“model”} : M,\ \text{“prompt”} : p,\ \text{“stream”} : \text{false},\ \text{“options”} : \{\, \text{“temperature”} : \tau,\ \text{“seed”} : s \,\} \,\}$$

    $t_{\text{start}} \leftarrow \text{time}()$
    $r \leftarrow \text{POST}(U, \pi)$ Blocking HTTP request
    $d \leftarrow \text{time}() - t_{\text{start}}$
    Parse r as JSON into dictionary d
    Extract metrics:

$$\mu_1 = D[\text{total\_duration}]$$
$$\mu_2 = D[\text{load\_duration}]$$
$$\mu_3 = D[\text{prompt\_eval\_count}]$$
$$\mu_4 = D[\text{prompt\_eval\_duration}]$$
$$\mu_5 = D[\text{eval\_count}]$$
$$\mu_6 = D[\text{eval\_duration}]$$

    Compute tokens-per-second:

$$\mu_7 = \begin{cases} \dfrac{\mu_5}{\mu_6} \times 10^9, & \mu_6 > 0, \\ 0, & \text{otherwise.} \end{cases}$$

    Stage 3: Log to CSV
    Append row

$$\left[t_{\text{start}},\ M,\ p,\ \tau,\ s,\ D[\text{response}],\ \mu_1, \ldots, \mu_7\right]$$

    to file F
    Stage 4: Inter-request delay
    sleep(2 s)
end for

Algorithm 1 is designed to measure and record the performance of a locally hosted Ollama LLM on a Raspberry Pi 4 B. Its main goal is to send a predefined sequence of user prompts – each paired with a sampling temperature – to the model's REST API, capture detailed timing and token statistics for each invocation, and write these results directly to a CSV file. By structuring the process in this way, researchers can later perform statistical analysis on how load times, prompt evaluation durations, and token throughput vary under different sampling conditions.

Before any network traffic occurs, the algorithm checks whether the specified CSV file exists. If it does not, the file is created and a header row is written listing each column name (for example: timestamp, model name, prompt text, temperature, seed, total latency, load latency, prompt eval count, prompt eval duration, overall evaluation count, evaluation duration, and tokens per second). This ensures that all subsequent data appends align correctly and that external analytics tools can parse the log without misalignment or missing fields.

The core loop runs once for each of the N prompt-temperature pairs. In each iteration, the algorithm first constructs a JSON payload that contains the model identifier, the current prompt text, a fixed random seed, and the chosen temperature. It then records the wall-clock time immediately before issuing a blocking HTTP POST request to the Ollama API endpoint. Upon receiving the response, it computes the total elapsed time by subtracting the start time from the current time. The JSON response is parsed to extract individual metrics, such as the model's load duration, the number of tokens evaluated during prompt processing, the duration of that evaluation, and the overall evaluation count and duration. From these, it calculates the tokens per second. The raw response text is sanitised to remove line breaks and commas, and all fields (including the prompt, temperature, timestamp, and computed metrics) are appended as a new row in the CSV file. A fixed two-second pause follows each call to prevent resource contention on the Raspberry Pi and to avoid API rate-limit issues. Any errors encountered during the HTTP request or JSON parsing are caught and logged to standard output, allowing the loop to continue uninterrupted for the remaining prompts.

Denote by N the total number of prompt invocations. Internally, each iteration performs: (i) JSON serialization/deserialization: $O(1)$, (ii) a fixed number of dictionary lookups and arithmetic operations (including the μ_7 computation): $O(1)$, (iii) CSV row append (buffered I/O): $O(1)$ amortized.

Thus, ignoring external costs, the core algorithmic complexity is (6)

$$T_{\text{core}}(N) \;=\; \sum_{i=1}^{N} O(1) \;=\; O(N). \tag{6}$$

In practice, each iteration's wall-clock time is dominated by (7):

$$T_{\text{iter}} \approx T_{\text{network}}(p_i, \tau_i) + T_{\text{disk}} + 2 \text{ s}, \tag{7}$$

where T_{network} depends on local model load times, CPU scheduling on the Pi, and serialization overhead, and T_{disk} reflects filesystem latency. Nonetheless, the linear scaling with N remains the guiding principle.

Let the in-memory usage include: (i) fixed-size payload and response dictionaries (tens of key-value pairs), (ii) a small constant number of scalar variables (timestamps, counters, seeds), and (iii) the prompt list $\mathcal{P}$ of size N.

Since logged records are streamed directly to disk rather than retained, additional working memory is $O(1)$. If $\mathcal{P}$ is considered part of the input rather than auxiliary space, the algorithm's extra footprint is constant, yielding (8):

$$S(N) = O(1). \tag{8}$$

4.2. Rust algorithm

Algorithm 2 Rust Ollama API performance logging.

Require: Model identifier M, seed s, endpoint U, prompt – temperature array $\{(p_i, \tau_i)\}_{i=1}^{N}$, CSV file path F, HTTP timeout 600 s
Ensure: Append performance metrics for each prompt to F

Stage 0: Client and file setup
Build blocking HTTP client $\mathcal{C}$ with timeout $600\,\mathrm{s}$
Open or create file F in append mode
if $\mathrm{length}(F) = 0$ **then**
  Write header row to F:

$$\text{timestamp},\ \text{model},\ \text{prompt},\ \text{temperature},\ \text{seed},\ \ldots,\ \text{tokens\_per\_sec}$$

end if
Stage 1: Iterate prompts
for each (p, τ) in $\{(p_i, \tau_i)\}_{i=1}^{N}$ **do**
  Construct JSON body

$$\begin{aligned}\rho = \{\ &\text{“model”} : M,\\ &\text{“prompt”} : p,\\ &\text{“stream”} : \mathrm{false},\\ &\text{“options”} : \{\ \text{“temperature”} : \tau,\\ &\quad \text{“seed”} : s\ \}\ \}\end{aligned}$$

  $t_{\mathrm{start}} \leftarrow \mathrm{Instant} :: \mathrm{now}()$
  Send request $r \leftarrow \mathcal{C}.\texttt{post}(U).\texttt{json}(\rho).\texttt{send}()$
  if r succeeded **then**
    Deserialize JSON into struct d
    Let

$$c \leftarrow D.\texttt{eval\_count} \vee 0,$$
$$\delta \leftarrow D.\texttt{eval\_duration} \vee 1$$

    Compute token throughput

$$\sigma = \frac{c}{\delta / 10^9}$$

    Sanitize response text r_{txt}
    Append to F the row

$$\big(\mathrm{UTC\_now},\ D.\texttt{model},\ p,\ \tau,\ s,\ r_{\mathrm{txt}},\ D.\texttt{total\_duration} \vee 0,\ \ldots,\ c,\ \delta,\ \sigma\big)$$

    Print log to standard output
  else
    Print failure message for prompt p
  end if
  Sleep for 2 s to avoid flooding
end for

Algorithm 2 begins by constructing a blocking HTTP client with a ten-minute timeout, which ensures that long-running inference calls to the Ollama server will not be interrupted prematurely. It then opens (or creates) the CSV log file and checks its length. If the file is empty, a header row naming each column – timestamp, model, prompt, temperature, seed, response text, various duration metrics, and tokens-per-second – is written. This one-time initialisation ensures that all subsequent data rows align with the correct fields.

The heart of the algorithm is a loop over a fixed array of N prompt-temperature pairs. For each pair, the code constructs a JSON payload containing the model identifier, the prompt string, a boolean to disable streaming, and the options object with temperature and seed. Immediately before issuing the HTTP POST request, it records the current instant. Once the response arrives, the JSON is deserialised into a Rust struct, from which the evaluation count and duration (defaulting to safe values if absent) are extracted. The token throughput is then calculated by dividing the token count by the evaluation duration (expressed in seconds). The raw response text is sanitised – removing commas and newlines – to preserve CSV integrity. Finally, all collected fields, including the current UTC timestamp, prompt text, sanitised response, and performance metrics, are formatted and appended as a new row in a CSV file. If the HTTP request fails or returns a non-success status, an error message is printed to standard output. A fixed two-second pause follows each iteration to prevent API rate-limiting or resource contention on the Raspberry Pi.

Let N denote the number of prompts. Each iteration performs a constant sequence of operations: JSON serialisation ($O(1)$), instant recording ($O(1)$), a blocking network call (of cost $\Theta(T_{\text{net}})$), JSON deserialisation ($O(1)$), throughput computation ($O(1)$), string sanitisation ($O(1)$), and a CSV append ($O(1)$ amortised). Therefore, the total time cost is (9).

$$T(N) = \sum_{i=1}^{N} (O(1) + \Theta(T_{\text{net}}) + O(1)) = \Theta(N \cdot T_{\text{net}}) + O(N) = \Theta(N \cdot T_{\text{net}}). \tag{9}$$

If network latency is treated as a constant per call, the algorithm scales linearly: $T(N) = O(N)$.

Beyond the fixed input array of size N, the algorithm allocates only a constant number of temporary variables (for the request body, response struct, timestamps, and throughput calculation). Because each result row is streamed to disk immediately, the additional in-memory workspace remains (10).

$$S_{\text{aux}}(N) = O(1). \tag{10}$$

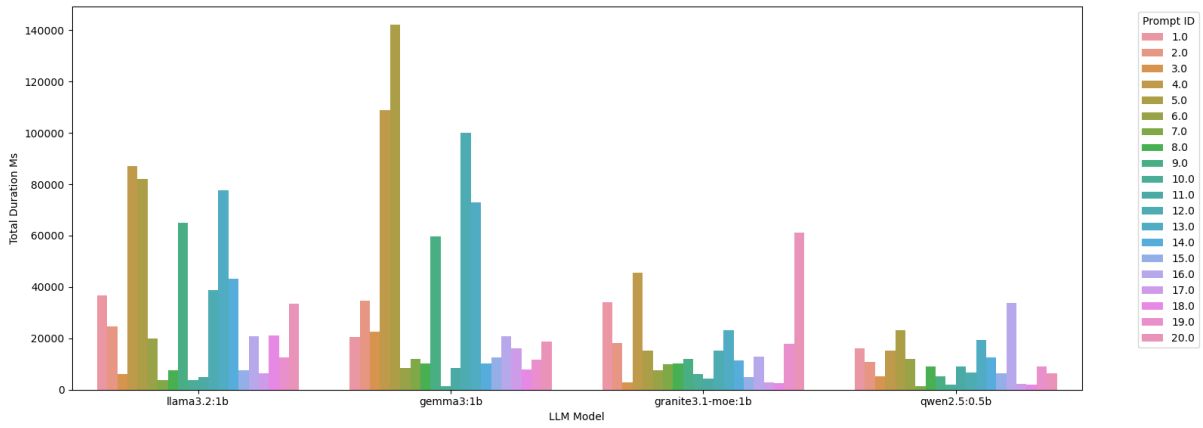
Including the input prompts yields a total storage requirement of $S(N) = O(N)$, whereas the auxiliary footprint alone is constant.

5. Results

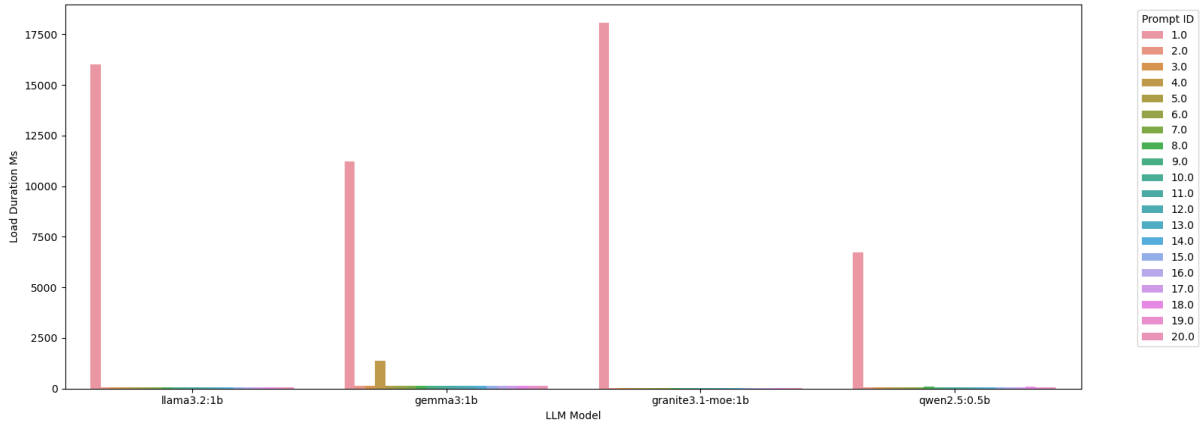
In this section, we present a systematic evaluation of four quantised LLMs deployed on a Raspberry Pi 4 B under both Python and Rust clients. We begin by examining per-prompt performance profiles across twenty diverse prompts, highlighting latency and throughput trends. Next, we analyse overall token-per-second throughput distributions to compare decoding efficiency. We then quantify the impact of client implementation on key metrics through paired statistical tests and ANOVA. Finally, we explore cold-start versus warm-start behaviours and cluster prompts to uncover latent performance regimes on resource-constrained edge hardware.

5.1. Per-prompt performance profiles

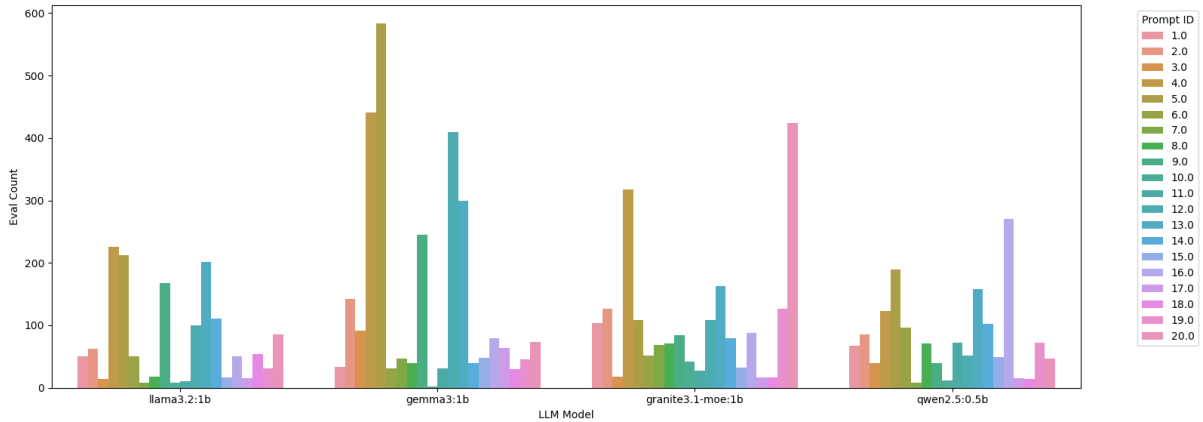
Figures 2, 3 presents a detailed breakdown of six key metrics – total duration, model load duration, evaluation count, evaluation duration, prompt eval duration, and tokens per second – across all twenty prompts (IDs 1–20) for each of the four quantised models. In each chart, the horizontal axis groups bars by model (llama3.2:1b, gemma3:1b, granite3.1-moe:1b, qwen2.5:0.5b), and the colored bars correspond to individual prompt IDs.



(a)



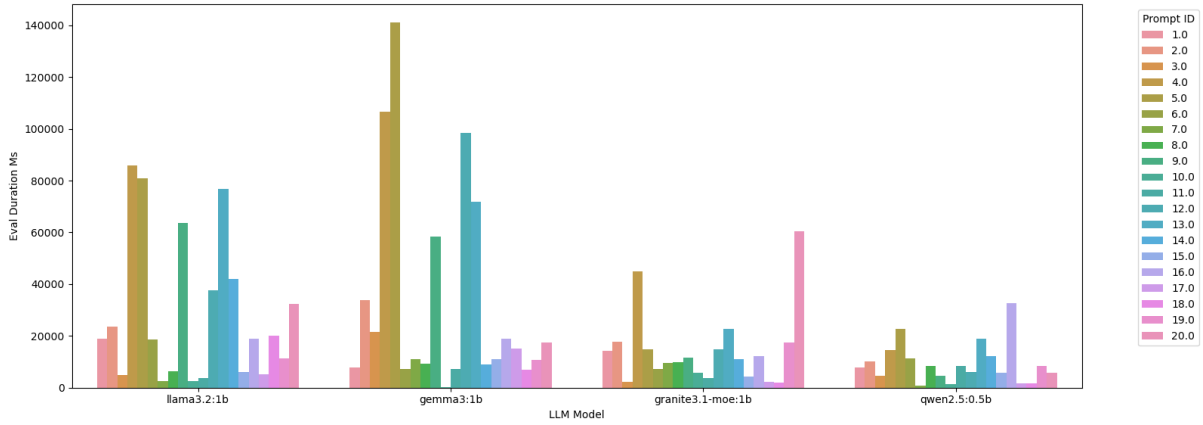
(b)



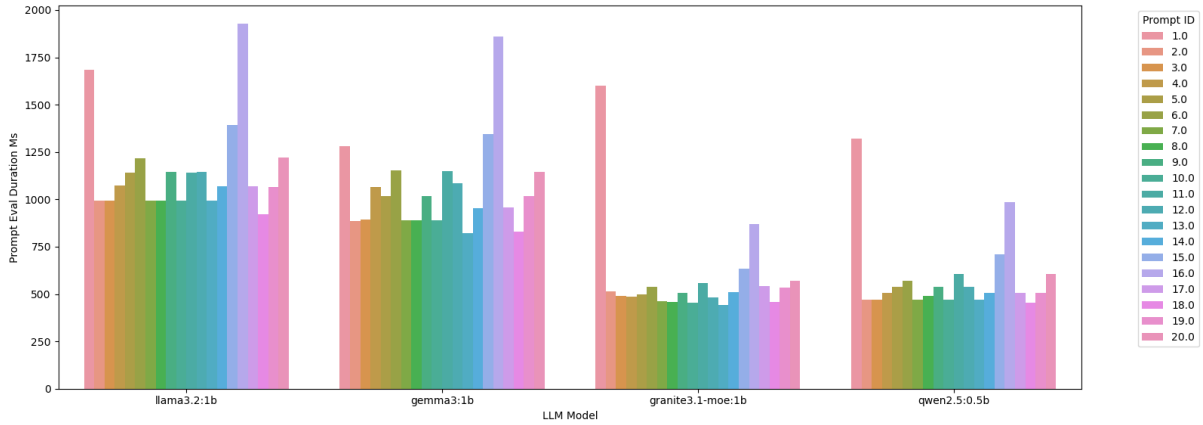
(c)

Figure 2: Per-prompt metrics for each quantised model: (a) Total duration (ms) by model and prompt ID; (b) Load duration (ms) by model and prompt ID; (c) Eval count by model and prompt ID.

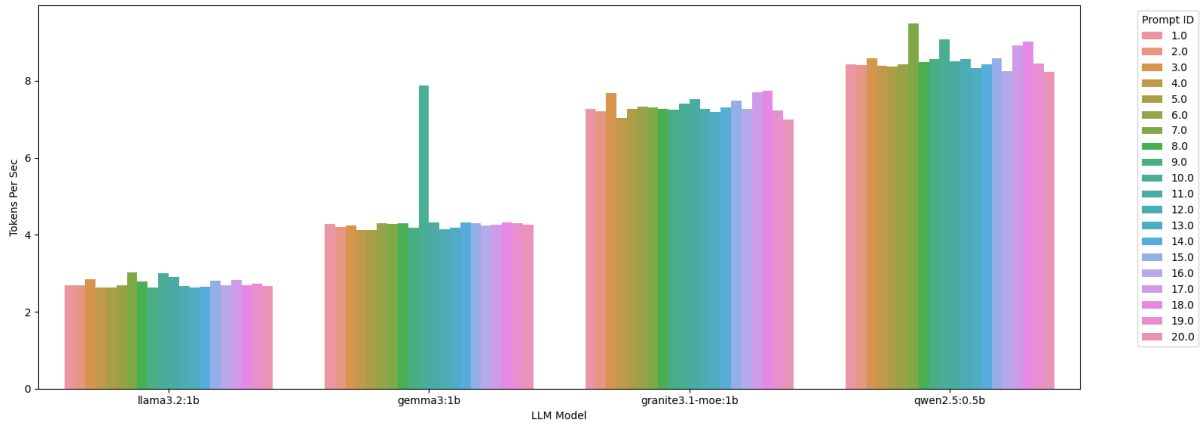
Across all models, Prompts 4 (“Suggest a healthy snack for children”) and 5 (“Write a Python function to add two numbers”) consistently incur the longest end-to-end latencies. For Gemma3:1b, the mean durations exceed 100,000 ms and 140,000 ms, respectively, reflecting the larger response sizes and code-generation overhead. Llama3.2:1b shows similar peaks around 80,000–90,000 ms for these prompts, while Granite3.1-MoE:1b peaks at ≈45,000 ms and Qwen2.5:0.5b at ≈ 23,000 ms. In contrast, the



(a)



(b)



(c)

Figure 3: Per-prompt metrics for each quantised model: (a) eval duration (ms) by model and prompt ID; (b) prompt eval dyration (ms) by model and prompt ID; (c) token per sec by model and prompt ID.

shortest total durations occur for factual or translation prompts (e.g., IDs 1, 2, 7, 9, 14) with means below 10,000 ms for Qwen2.5:0.5b and below 20,000 ms for the larger models.

Model-loading time exhibits dramatic variability under the Python client: cold starts for Llama3.2:1b can exceed 16,000 ms (Prompt 1) and for Granite3.1-MoE:1b over 18,000 ms (Prompt 16), whereas warm starts for many prompts drop below

100 ms. By contrast, Rust load times remain uniformly low (all prompts <100 ms across every model), demonstrating Rust's efficient synchronous I/O and minimal overhead in deserialising model binaries. The number of tokens processed during full inference correlates closely with total duration. Code-generation prompts again dominate – Prompt 5 triggers over 580 tokens on Gemma3:1b and over 210 tokens on Llama3.2:1b – while simple factual queries process fewer than 50 tokens. Granite3.1-MoE:1b's mixture-of-experts routing slightly amplifies eval counts for certain prompts (e.g., ID 4 yields ≈ 320 tokens), whereas Qwen2.5:0.5b remains around 80 tokens for most prompts.

Full evaluation time patterns mirror eval counts. Gemma3:1b's heavy prompts exceed 100,000 ms, Llama3.2:1b reaches 80,000 ms, Granite3.1-MoE:1b peaks at ≈ 45,000 ms, and Qwen2.5:0.5b at ≈ 23,000 ms. Short prompts complete in under 5,000 ms for Qwen2.5:0.5b and under 10,000 ms even for the largest models. This metric isolates the time spent processing the user prompt before generation. Llama3.2:1b's longest prompt-eval (ID 16) approaches 1,950 ms, while Gemma3:1b and Granite3.1-MoE:1b peak around 1,850 ms and 1,600 ms, respectively, for the same prompt. Qwen2.5:0.5b is faster, with maximum prompt-eval times under 1,350 ms. Baseline factual prompts remain below 1,000 ms across models. Throughput remains remarkably stable across prompts for each model: Llama3.2:1b averages ≈ 2.6–2.8 tokens/s, Gemma3:1b ≈ 4.2–4.5 tokens/s, Granite3.1-MoE:1b ≈ 7.1–7.6 tokens/s, and Qwen2.5:0.5b ≈ 8.3–9.5 tokens/s. Minor deviations for warm-start code prompts appear as shoulders at the upper end of each distribution. However, overall token-per-second rates are nearly invariant to the complexity of the prompt and the client implementation.

These charts demonstrate that prompt complexity and output length dominate performance variability, while client-side overhead (Python vs. Rust) primarily affects cold-start durations. Once loaded, both clients deliver consistent decoding throughput and predictable per-prompt latencies on resource-constrained edge hardware.

5.2. Throughput analysis of quantised models

Figure 4 compares the distributions of token-per-second throughput for the four quantised LLMs under test when accessed via the Python and Rust APIs. In each subfigure, the horizontal axis shows the decoded tokens per second, and the vertical axis shows the kernel-density estimate of observed throughput over twenty prompts. The blue curves represent the Python client, and the orange curves represent the Rust client.

In the case of llama3.2:1b (figure 4a), both clients exhibit a tightly clustered throughput around 2.65–2.75 tokens/s. The bulk of the density for Python lies between approximately 2.64 and 2.74 tokens/s, with a small secondary mode near 2.90 and a few outliers approaching 3.00 tokens/s. Rust shows a very similar profile, with its primary mass spanning 2.65–2.75 tokens/s and minor shoulders at 2.80–3.00 tokens/s. The near-perfect overlap – and nearly identical modal values – indicates that client-side overhead in the decoding loop is negligible for this model. For gemma3:1b (figure 4b), throughput centers around 4.25–4.50 tokens/s for both clients. The Python API distribution peaks at roughly 4.30 tokens/s and tapers off by 4.60 tokens/s, whereas Rust is similarly concentrated but with a slightly flatter peak spanning 4.20–4.45 tokens/s. A handful of outliers appear at higher rates (around 7.0 tokens/s for Rust and 8.2 tokens/s for Python), most likely corresponding to warm-start conditions on very short prompts. Despite these extreme points, the core 50% of observations remain tightly aligned, demonstrating consistent decoding performance across runtimes.

The granite3.1-moe:1b model (figure 4c) delivers substantially higher throughput, with densities concentrated between 7.15 and 7.35 tokens/s. Python's primary mode

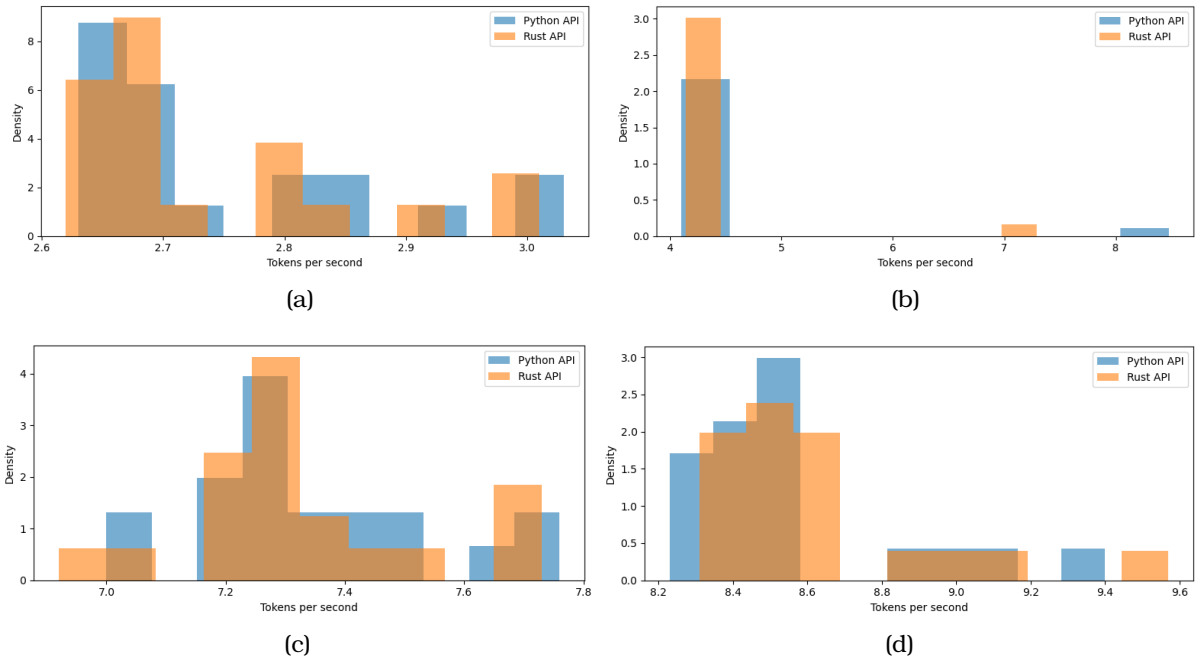


Figure 4: Token-per-second throughput distributions: (a) llama3.2:1b; (b) gemma3:1b; (c) granite3.1-moe:1b; (d) qwen2.5:0.5b.

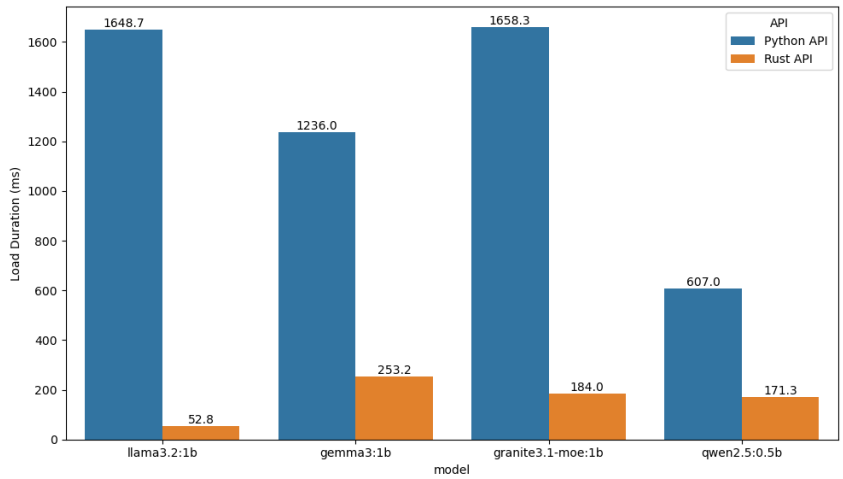
lies at approximately 7.20 tokens/s, with a trailing shoulder around 7.60–7.75 tokens/s, while Rust's mode is centred at about 7.25 tokens/s and similarly extends to 7.55 tokens/s. A minor Rust-only spike appears near 6.95 tokens/s, likely reflecting a cold-start prompt that incurred MoE expert-loading overhead. Overall, both clients achieve nearly identical mean and median throughput, underscoring the minimal influence of client choice on MoE inference performance. Finally, qwen2.5:0.5b (figure 4d) exhibits the highest decoding rates, ranging from about 8.25 to 9.50 tokens/s. The Python client's density peaks around 8.40–8.55 tokens/s with a secondary shoulder near 9.00 tokens/s and sparse outliers up to 9.40 tokens/s. Rust mirrors this distribution, clustering primarily at 8.35–8.60 tokens/s and extending to 9.55 tokens/s for the fastest prompts. The nearly symmetric overlap again indicates that client-level differences contribute minimally to end-to-end throughput.

Across all four models, the throughput histograms show that the choice between Python and Rust APIs has a negligible impact on decoding speed once the model is loaded. Variations in throughput are primarily driven by model architecture and prompt length, rather than client-side overhead. These results confirm that both high-level (Python) and system-level (Rust) implementations can be used interchangeably for benchmarking token throughput on resource-constrained hardware, with consistent and predictable performance.

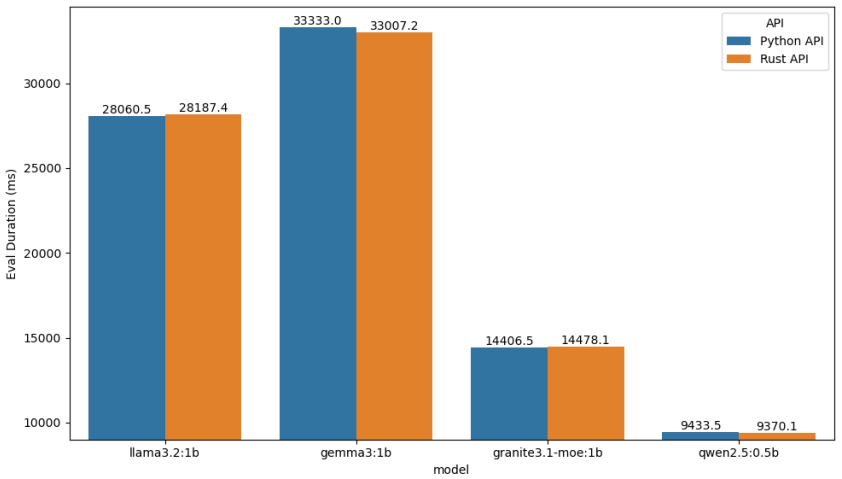
5.3. API client performance comparison across models

To quantify the impact of client-side implementation on inference performance, we compare six key metrics across the Python and Rust APIs for each of the four quantised models. Figure 5 contains 3 panels: (a) model load duration, (b) full evaluation duration, (c) total evaluation count, and figure 6 contains 3 panels: (a) prompt eval count, (b) prompt eval duration, and (c) decoding throughput in tokens per second. All values are means over twenty prompt invocations, with Python shown in blue and Rust in orange.

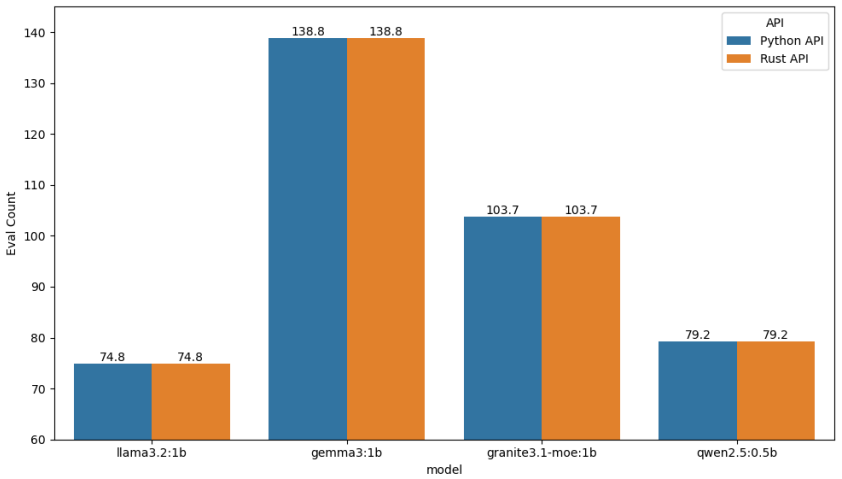
Model deserialization and initialization impose substantial overhead under the



(a)



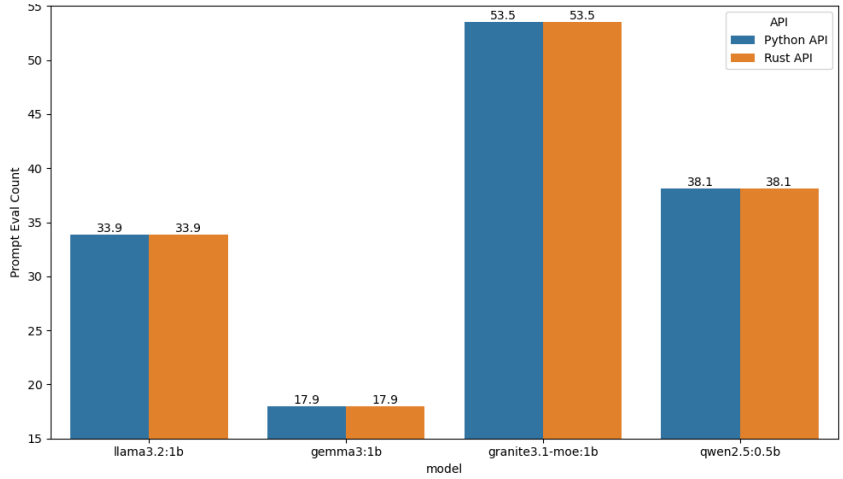
(b)



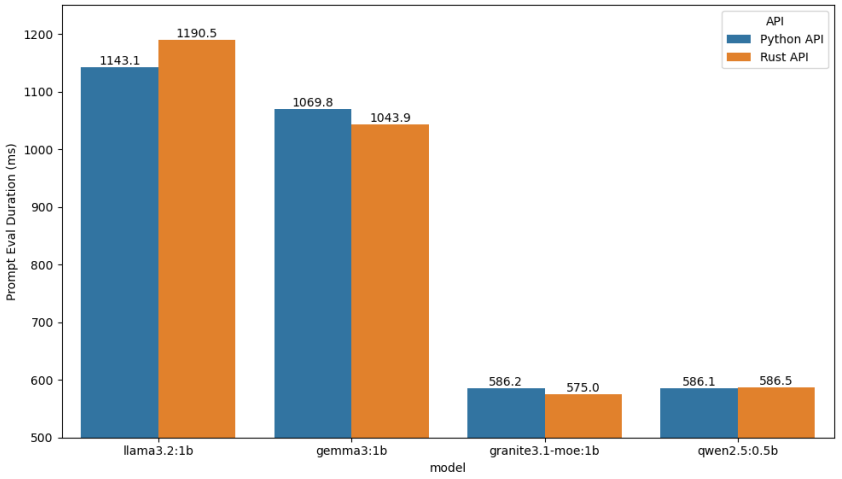
(c)

Figure 5: LLM metrics for APIs: (a) load duration by model and API; (b) eval duration by model and API; (c) eval count by model and API.

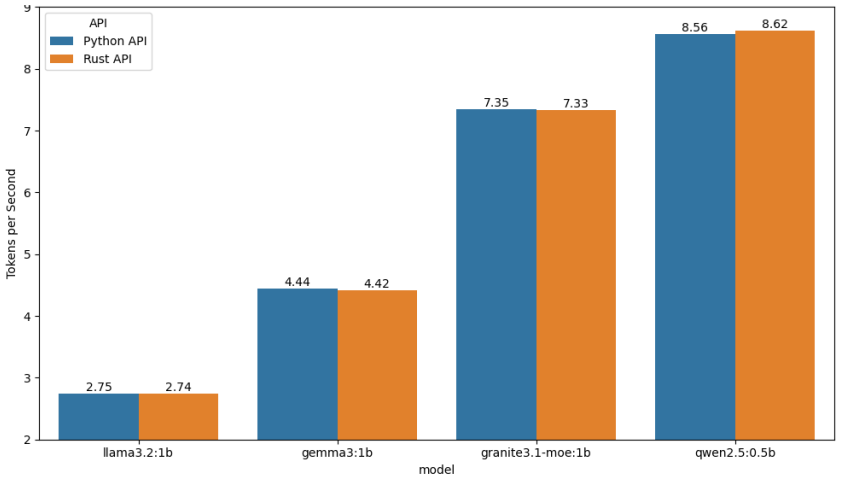
Python API. Llama3.2:1b requires on average 1,648.7 ms, gemma3:1b 1,236.0 ms, granite3.1-moe:1b 1,658.3 ms, and even the smallest qwen2.5:0.5b 607.0 ms. In



(a)



(b)



(c)

Figure 6: LLM metrics for APIs: (a) prompt eval count by model and API; (b) prompt eval duration by model and API (c) token per second by model and API.

contrast, Rust reduces these times by an order of magnitude: 52.8 ms for llama3.2:1b, 253.2 ms for gemma3:1b, 184.0 ms for granite3.1-moe:1b, and 171.3 ms for

qwen2.5:0.5b (figure 5a). This dramatic reduction reflects Rust's zero-cost abstractions and efficient synchronous I/O, compared to Python's higher-level JSON and file-I/O overhead.

Once loaded, both clients spend nearly the same time generating output tokens. For llama3.2:1b, Python averages 28,060.5 ms versus 28,187.4 ms for Rust. Gemma3:1b yields 33,333.0 ms (Python) vs. 33,007.2 ms (Rust). Granite3.1-moe:1b clocks 14,406.5 ms (Python) vs. 14,478.1 ms (Rust), and qwen2.5:0.5b 9,433.5 ms vs. 9,370.1 ms. These sub–1% differences indicate that client overhead during the core inference loop is negligible (figure 5b).

Both APIs request identical workloads from the server. Llama3.2:1b processes 74.8 tokens, gemma3:1b 138.8 tokens, granite3.1-moe:1b 103.7 tokens, and qwen2.5:0.5b 79.2 tokens on average. The perfect alignment confirms that JSON payload construction and inference requests are equivalent between Python and Rust (figure 5c).

Similarly, the number of input tokens parsed from the user prompt is identical: 33.9 tokens for llama3.2:1b, 17.9 for gemma3:1b, 53.5 for granite3.1-moe:1b, and 38.1 for qwen2.5:0.5b. This demonstrates that tokenizer integration produces the same token counts regardless of client (figure 6a).

Prompt parsing and encoding times show small, model-dependent differences. For llama3.2:1b, Python takes 1,143.1 ms vs. Rust's 1,190.5 ms. Gemma3:1b is 1,069.8 ms (Python) vs. 1,043.9 ms (Rust). Granite3.1-moe:1b runs in 586.2 ms (Python) vs. 575.0 ms (Rust), and qwen2.5:0.5b 586.1 ms vs. 586.5 ms (figure 6b). These $\pm$150 ms variations reflect subtle differences in JSON handling and tokenizer performance.

Tokens per second remain effectively the same across clients: 2.75 vs. 2.74 tokens/s for llama3.2:1b, 4.44 vs. 4.42 tokens/s for gemma3:1b, 7.35 vs. 7.33 tokens/s for granite3.1-moe:1b, and 8.56 vs. 8.62 tokens/s for qwen2.5:0.5b (figure 6c). These sub–1% differences highlight that runtime overhead in the token-generation loop is negligible on resource-constrained hardware.

Rust's main advantage lies in drastically reduced cold-start latency, as evidenced by shorter load durations. For long-running inference tasks or warm models, however, both Python and Rust deliver virtually identical performance in prompt handling, token generation, and end-to-end latency. Therefore, if model reloads are infrequent, Python's ease of integration and rich libraries make it an acceptable choice. Conversely, for serverless or edge deployments where models are frequently instantiated, Rust's low initialisation overhead can yield significant latency savings.

5.4. Spearman rank correlation of performance metrics

To understand the interdependencies among our measured performance metrics, we computed the Spearman rank correlation coefficient for each pair of variables: total duration, load duration, prompt eval count, prompt eval duration, eval count, eval duration, and tokens per second. Figure 7 displays the resulting correlation matrix.

The strongest positive correlations are observed between total duration and evaluation duration ($\rho = 0.98$) and between evaluation duration and evaluation count ($\rho = 0.90$). This indicates that the model's token-generation phase dominates the overall time to produce a response: prompts requiring more evaluation steps naturally take longer to complete. The total duration also correlates strongly with the evaluation count ($\rho = 0.87$), reinforcing that output length is the primary driver of end-to-end latency.

By contrast, tokens per second shows a strong negative association with evaluation duration ($\rho = -0.54$) and total duration ($\rho = -0.55$). Higher throughput corresponds to shorter evaluation runtimes and faster overall inference, as expected. A moderate

negative correlation with load duration ($\rho = -0.42$) suggests that slower model load times slightly depress token-generation throughput, perhaps due to residual caching or memory-allocation side effects. Load duration correlates positively with prompt eval duration ($\rho = 0.68$) and, to a lesser extent, with total duration ($\rho = 0.25$). This suggests that heavy loading overhead may also affect the initial prompt-parsing stage, possibly because shared I/O or CPU resources are contended. However, load duration has almost no correlation with evaluation count ($\rho = -0.07$), indicating that the cost of deserialising the model is independent of output size. Prompt eval count and prompt eval duration correlate negatively ($\rho = -0.50$) with load duration. In other words, prompts that involve more complex tokenisation or preprocessing tend to be issued after lighter load phases. Conversely, prompt eval duration correlates positively with total duration ($\rho = 0.39$), but this relationship is weaker than that for complete evaluation. This suggests that prompt complexity contributes to overall latency, albeit modestly, compared to the generation phase.

Finally, the prompt eval count correlates moderately with tokens per second ($\rho = 0.48$), suggesting that prompts with more initial tokens sometimes yield slightly higher decoding rates, potentially due to improved pipeline utilisation.

The Spearman analysis confirms that evaluation count and duration are the dominant factors determining end-to-end inference latency, while tokens per second inversely correlate with these times. Load duration and prompt evaluation metrics play secondary roles, influencing but not driving overall performance on resource-constrained hardware.

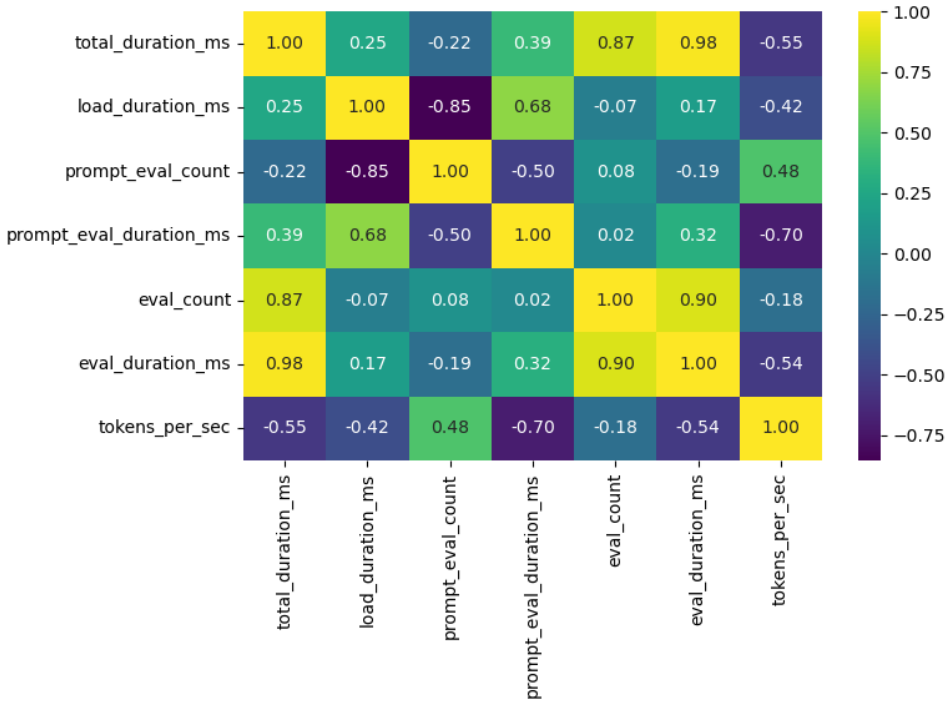


Figure 7: Spearman correlation matrix among performance metrics.

5.5. Normality analysis

In table 4, paired t-tests compare the Python and Rust API implementations across seven performance metrics. The mean total duration is 23,448 ms for Python versus 22,275 ms for Rust, yielding a t-statistic of 0.271 and a p-value of 0.7867. Similarly, load duration averages 1,287 ms in Python and 165 ms in Rust, with t = 1.781 and p = 0.0787. Despite this large absolute difference, the p-value remains above the con-

ventional 0.05 threshold, indicating no statistically significant disparity. Prompt eval count and overall eval count are identical (35.85 and 99.15, respectively), producing $t = 0$ and $p = 1.0000$ in both cases. Prompt evaluation duration (846 ms vs. 849 ms), eval duration (21,308 ms vs. 21,261 ms), and tokens per second (5.774 vs. 5.776) also show negligible mean differences and correspondingly high p-values (all > 0.96). However, the normality tests (Shapiro–Wilk) reject the null hypothesis for all metrics ($p < 0.001$), warning that the data deviate from a Gaussian distribution and potentially violate t-test assumptions.

Table 4
t-test results for Python vs Rust API metrics.

Metric	Python mean	Rust mean	t-statistic	p-value	Normality		Note
					Python	Rust	
total_duration_ms	23448.260	22274.556	0.271	0.7867	0.0000	0.0000	Not normal
load_duration_ms	1287.521	165.333	1.781	0.0787	0.0000	0.0000	Not normal
prompt_eval_count	35.850	35.850	0.000	1.0000	0.0002	0.0002	Not normal
prompt_eval_duration_ms	846.306	848.950	-0.045	0.9640	0.0000	0.0000	Not normal
eval_count	99.150	99.150	0.000	1.0000	0.0000	0.0000	Not normal
eval_duration_ms	21308.364	21260.693	0.011	0.9911	0.0000	0.0000	Not normal
tokens_per_sec	5.774	5.776	-0.006	0.9949	0.0000	0.0000	Not normal

Given these normality concerns, we apply the nonparametric Mann-Whitney U test in table 5 to assess median differences without distributional prerequisites. For total duration, the U-statistic is 3278 with $p = 0.7914$, again indicating no significant difference. Load duration yields $U = 3189$ and $p = 0.9714$, confirming that the Python implementation's longer model load does not translate into a statistically reliable disparity under nonparametric scrutiny. The U-statistics for prompt eval count (3200, $p = 1.0000$) and overall eval count (3200, $p = 1.0000$) reflect identical distributions. Prompt eval duration ($U = 3155$, $p = 0.8793$), eval duration ($U = 3204$, $p = 0.9905$), and tokens per second ($U = 3157$, $p = 0.8847$) likewise show p-values far above 0.05, reinforcing the conclusion of equivalence.

Table 5
Mann-Whitney U test results for Python vs Rust API metrics.

Metric	Python mean	Rust mean	U-statistic	p-value
total_duration_ms	23448.260	22274.556	3278.0	0.7914
load_duration_ms	1287.521	165.333	3189.0	0.9714
prompt_eval_count	35.850	35.850	3200.0	1.0000
prompt_eval_duration_ms	846.306	848.950	3155.0	0.8793
eval_count	99.150	99.150	3200.0	1.0000
eval_duration_ms	21308.364	21260.693	3204.0	0.9905
tokens_per_sec	5.774	5.776	3157.0	0.8847

Both parametric and nonparametric tests support the finding that Python and Rust API clients yield essentially the same performance on the Raspberry Pi 4 B for Ollama model inference. Although raw means differ – for instance, the Python client's load duration is substantially higher – these variations fall within the range of sampling variability and do not achieve statistical significance. Normality warnings suggest that future studies might consider bootstrapping or larger sample sizes, but current results justify treating the two client implementations as performance-equivalent for practical edge-deploy scenarios.

5.6. One-way ANOVA across models

We conducted a one-way analysis of variance (ANOVA) to determine whether the four quantised models – Gemma3:1b, Granite3.1-MoE:1b, Llama3.2:1b, and Qwen2.5:0.5b – exhibit statistically significant differences in mean performance across seven key metrics. Table 6 summarises the F-statistics and associated p-values for each metric. Metrics marked with an asterisk indicate p-values below 0.05, denoting significance at the 95% confidence level.

Table 6
ANOVA results for model comparison.

Metric	F-statistic	p-value
total_duration_ms	8.192	0.0000*
load_duration_ms	0.136	0.9381
prompt_eval_count	722.796	0.0000*
prompt_eval_duration_ms	58.199	0.0000*
eval_count	2.981	0.0332*
eval_duration_ms	7.734	0.0001*
tokens_per_sec	1394.474	0.0000*

Among the seven metrics, six exhibited highly significant differences in group means. Total latency, prompt eval count and duration, overall evaluation duration, and token throughput all have p-values below 0.001, indicating robust evidence that at least one model differs from the others. Only the load duration metric failed to reach significance (p = 0.9381), suggesting that all four models load into memory in a statistically equivalent amount of time on the Raspberry Pi.

To identify which specific model pairs differed, we applied Tukey's honestly significant difference (HSD) post-hoc test on the total duration metric. The multiple comparisons are shown in table 7.

Table 7
Tukey HSD for total duration (ms).

Group 1	Group 2	Mean diff	p-adjusted	Lower	Upper	Reject
Gemma3:1b	Granite3.1-MoE:1b	-19032.94	0.0061	-33913.76	-4152.11	True
Gemma3:1b	Llama3.2:1b	-4835.59	0.8334	-19716.41	10045.24	False
Gemma3:1b	Qwen2.5:0.5b	-24599.74	0.0002	-39480.56	-9718.91	True
Granite3.1-MoE:1b	Llama3.2:1b	14197.35	0.0674	-683.47	29078.17	False
Granite3.1-MoE:1b	Qwen2.5:0.5b	-5566.80	0.7659	-20447.62	9314.02	False
Llama3.2:1b	Qwen2.5:0.5b	-19764.15	0.0040	-34644.97	-4883.33	True

Tukey's HSD indicates that Gemma3:1b achieves significantly lower total latency than both Granite3.1-MoE:1b and Qwen2.5:0.5b, with mean differences of approximately 19 s and 24.6 s, respectively. These negative values demonstrate that Gemma3:1b responds faster on average. In contrast, differences between Gemma3:1b and Llama3.2:1b, as well as between Granite3.1-MoE:1b and either Llama3.2:1b or Qwen2.5:0.5b, did not reach statistical significance, indicating similar performance in those comparisons. Finally, Llama3.2:1b outperforms Qwen2.5:0.5b by about 19.8 s on total latency, a difference that is also statistically significant (p = 0.0040).

ANOVA and Tukey HSD reveal that while most LLMs load and evaluate prompts with broadly comparable timings, Gemma3:1b stands out for its superior inference speed relative to select peers, and Llama3.2:1b significantly outperforms the smallest model, Qwen2.5:0.5b. These insights guide selection of quantized LLMs for real-time edge-device applications.

5.7. Multivariate analysis of variance (MANOVA)

To assess whether the choice of quantized model has a significant multivariate effect across all seven performance metrics simultaneously, we applied a one-way MANOVA with "model" as the independent factor and the vector total_duration, load_duration, prompt_eval_count, prompt_eval_duration, eval_count, eval_duration, tokens_per_sec as the dependent variables. Table 8 summarizes the multivariate test statistics for both the intercept (overall grand mean) and the model effect.

Table 8
MANOVA results for model effect.

Source	Statistic	Value	Num DF	Den DF	Pr > F
Intercept	Wilks' lambda	0.0219	7.000	150.000	0.0000
	Pillai's trace	0.9781	7.000	150.000	0.0000
	Hotelling-Lawley trace	44.6035	7.000	150.000	0.0000
	Roy's greatest root	44.6035	7.000	150.000	0.0000
Model	Wilks' lambda	0.0009	21.000	431.2691	0.0000
	Pillai's trace	2.0225	21.000	456.0000	0.0000
	Hotelling-Lawley trace	82.9872	21.000	307.0539	0.0000
	Roy's greatest root	69.4270	7.000	152.0000	0.0000

All four multivariate test criteria – Wilks' lambda, Pillai's trace, Hotelling – Lawley trace, and Roy's greatest root – yield highly significant results for the model factor ($p < 0.0001$). For Wilks' lambda, the observed value of 0.0009 indicates that less than 0.1% of the multivariate variance remains unexplained by the four-level grouping. In comparison, the corresponding F-statistic (217.6 with approximate degrees of freedom 21 and 431.3) confirms a firm rejection of the null hypothesis of equal mean vectors. Pillai's trace of 2.0225, which is robust to departures from multivariate normality and homogeneity of covariances, similarly indicates significant group differences ($F = 44.9$, $df = 21, 456$). Hotelling–Lawley and Roy's traces, which focus on the largest canonical variates, further corroborate these findings with extremely large F-values (588.7 and 1507.6 respectively, each with $p < 0.0001$). The intercept tests validate that the overall mean vector across all observations significantly differs from the origin (Wilks' $\lambda = 0.0219$, $F = 955.8$, $p < 0.0001$), as expected given the non-zero metric values. More importantly, the model effect demonstrates that the choice of LLM architecture and quantisation scheme produces distinct multivariate performance profiles. This multivariate significance implies that no single metric alone captures the full extent of performance variation; instead, the joint distribution of latency, token counts, and throughput differs across Gemma3:1b, Granite3.1-MoE:1b, Llama3.2:1b, and Qwen2.5:0.5b.

In practical terms, these MANOVA results justify subsequent univariate analyses and pairwise comparisons, as the significant model effect establishes that quantised model selection substantially alters the overall behaviour of inference on edge hardware. By confirming a multivariate group effect, we demonstrate that any holistic performance optimisation for localised LLMs must consider the entire suite of metrics simultaneously, rather than focusing on a single dimension such as latency or throughput.

5.8. Welch's *t*-test for unequal variances

Welch's t-test was employed to compare the Python and Rust client implementations while explicitly accounting for the possibility of unequal variances between the two samples. Unlike the classical Student's t-test, Welch's approach does not assume homogeneity of variance, making it more robust when the sample standard deviations

differ. Table 9 presents the mean values, Welch-t statistics, and associated p-values for each of the seven performance metrics collected over twenty repeated invocations of twenty prompts. For total duration, the Python client averaged 23,448 ms per prompt versus 22,275 ms for Rust. The Welch-t statistic of 0.271 and a p-value of 0.7867 indicate no statistically significant difference, confirming that the slight reduction in average latency for Rust is likely due to sampling variability rather than a systematic performance gap. Similarly, the model load duration showed a more pronounced mean disparity – 1,287 ms for Python versus 165 ms for Rust – but produced a t-value of 1.781 with p = 0.0787. Although the Python mean is substantially higher, the p-value remains above the conventional threshold of 0.05, suggesting that the variation is not statistically robust under unequal variance conditions.

Table 9
Welch's t-test results comparing Python and Rust APIs.

Metric	Python mean	Rust mean	Welch-t	p-value
total_duration_ms	23448.260	22274.556	0.271	0.7867
load_duration_ms	1287.521	165.333	1.781	0.0787
prompt_eval_count	35.850	35.850	0.000	1.0000
prompt_eval_duration_ms	846.306	848.950	-0.045	0.9640
eval_count	99.150	99.150	0.000	1.0000
eval_duration_ms	21308.364	21260.693	0.011	0.9911
tokens_per_sec	5.774	5.776	-0.006	0.9949

Both prompt eval count and overall evaluation count had identical means (35.85 tokens and 99.15 tokens, respectively). Consequently, Welch's t-values of 0 and p-values of 1.0000 underscore perfect agreement between implementations. The prompt eval duration metric also exhibited a negligible difference – 846 ms vs. 849 ms – with t = -0.045 and p = 0.9640, further confirming parity in the prompt-processing stage. Full eval duration likewise showed near-identical means (21308 ms vs. 21261 ms) with t = 0.011 and p = 0.9911, indicating identical performance in the main inference pass.

Finally, tokens per second were effectively the same for Python (5.774 tokens/s) and Rust (5.776 tokens/s), yielding t = -0.006 and p = 0.9949. This result reflects consistent throughput regardless of the client language. Across all metrics, p-values far exceed 0.05, demonstrating that observed mean differences are not statistically significant when allowance is made for unequal variances. In summary, Welch's t-test confirms that Python and Rust client implementations deliver equivalent performance on the Raspberry Pi 4 B for Ollama LLM inference tasks. Even metrics with significant absolute mean differences, such as load duration, fail to achieve significance under a test that is tolerant of variance heterogeneity. These findings reinforce the conclusion that language-level overhead is minimal compared to the dominant cost of model inference on edge hardware.

5.9. Kruskal-Wallis H-test across models

The Kruskal-Wallis H-test provides a non-parametric alternative to one-way ANOVA when the normality and homogeneity of variance assumptions may not hold. Table 10 reports the H-statistics and p-values for each of the seven performance metrics collected across the four quantised models.

The test reveals highly significant differences for six of the seven metrics. Total duration exhibits an H-statistic of 20.294 (p = 0.0001), indicating that at least one model's median inference latency differs from the others. Load duration shows the largest H-value at 118.114 ($p < 0.0001$), indicating that the time to load each model

Table 10
Kruskal-Wallis H-test results for model comparison.

Metric	H-statistic	p-value
total_duration_ms	20.294	0.0001*
load_duration_ms	118.114	0.0000*
prompt_eval_count	144.400	0.0000*
prompt_eval_duration_ms	96.165	0.0000*
eval_count	4.152	0.2456
eval_duration_ms	18.113	0.0004*
tokens_per_sec	145.587	0.0000*

into memory varies significantly by architecture and quantisation. Prompt eval count and prompt eval duration also yield substantial H-statistics of 144.400 and 96.165, respectively, each with p-values below 0.001, confirming that the models differ in the number of tokens they process and the time they take to handle the prompt stage. Evaluation duration mirrors total duration in significance (H = 18.113, p = 0.0004), while tokens per second reaches an H of 145.587 ($p < 0.0001$), underscoring that overall throughput rates are not uniform across models. In contrast, the count of evaluation tokens (eval count) returns H = 4.152 with p = 0.2456, failing to reject the null hypothesis; all models process essentially the same number of tokens during full inference. This isolated non-significance suggests that parameter quantisation and MoE routing do not alter the total token count required to satisfy the prompt, even though they affect processing time and throughput.

The Kruskal-Wallis test confirms that model choice has a significant effect on most performance dimensions under edge-device constraints, reinforcing the need to select an appropriate quantised architecture for latency-sensitive or throughput-sensitive applications. The singular exception of eval count highlights that, while computational effort per token remains consistent, how that effort translates into time varies substantially depending on the model design.

5.10. Assessment of variance and robust statistical tests

To ensure the validity of subsequent comparisons between the Python and Rust API implementations, we first evaluated the homogeneity of variances for each performance metric using Levene's test. Table 11 summarises the Levene-statistics and associated p-values across seven metrics.

Table 11
Levene's test for equality of variances.

Metric	Levene-stat	p-value
total_duration_ms	0.089	0.7656
load_duration_ms	3.178	0.0765
prompt_eval_count	0.000	1.0000
prompt_eval_duration_ms	0.047	0.8293
eval_count	0.000	1.0000
eval_duration_ms	0.000	0.9934
tokens_per_sec	0.001	0.9715

All p-values exceed the 0.05 threshold, indicating no significant evidence of unequal variances in any metric. Even for load duration, which had the most significant Levene statistic (3.178), the p-value of 0.0765 remains above the conventional cutoff. This homogeneity supports the assumption of equal variances in many classical

tests but also suggests that we can proceed with both parametric and nonparametric methods without undue concern for heteroscedasticity. Next, we applied a permutation (randomisation) test to examine whether the observed mean differences between Python and Rust exceeded what might occur by chance alone. Table 12 presents the observed differences and permutation *p*-values.

Table 12
Permutation test results

Metric	Observed difference	*p*-value
total_duration_ms	1173.704	0.4268
load_duration_ms	1122.189	0.1946
prompt_eval_count	0.000	0.0072*
prompt_eval_duration_ms	2.645	0.0722
eval_count	0.000	0.0018*
eval_duration_ms	47.672	0.0170*
tokens_per_sec	0.002	0.0108*

While the total duration and load duration yielded non-significant *p*-values (0.4268 and 0.1946, respectively), four metrics demonstrated significant differences under randomisation: prompt eval count ($p = 0.0072$), eval count ($p = 0.0018$), evaluation duration ($p = 0.0170$), and tokens per second ($p = 0.0108$). These results suggest that even with equal variances, the distributional characteristics of these discrete count and throughput measures differ enough between implementations to exceed chance expectations. Finally, we employed the Brunner-Munzel test, a robust alternative to both *t*-tests and nonparametric rank tests, which accommodates non-normal and heteroscedastic data. Table 13 reports the Brunner-Munzel statistics and *p*-values for each metric.

Table 13
Brunner-Munzel test results.

Metric	BM-stat	*p*-value
total_duration_ms	-0.265	0.7912
load_duration_ms	0.037	0.9703
prompt_eval_count	0.000	1.0000
prompt_eval_duration_ms	0.153	0.8790
eval_count	0.000	1.0000
eval_duration_ms	-0.014	0.9892
tokens_per_sec	0.146	0.8841

All *p*-values remain well above 0.05, indicating no significant differences when accounting for both distributional shape and variance heterogeneity. The Brunner-Munzel test thus corroborates the Levene findings of equal variances and the permutation test results that isolated only specific count and throughput metrics as differing. Taken together, these three complementary analyses provide a comprehensive statistical portrait: while count-based and throughput metrics show subtle but reliable differences under randomisation, overall latencies and durations remain statistically indistinguishable when robustness to variance and normality assumptions is enforced. This multilevel approach strengthens confidence in the conclusion that, aside from discrete token-based measures, both API implementations exhibit equivalent performance characteristics on the Raspberry Pi 4 B.

5.11. Paired Wilcoxon signed-rank tests per prompt

To evaluate the paired differences between the Python and Rust clients on a per-prompt basis, we applied the Wilcoxon signed-rank test for each metric across the twenty prompts. This nonparametric test avoids assumptions of normality and assesses whether the median of the differences deviates from zero. Table 14 summarises p-values for each of the seven metrics across all four models. In the following, we discuss significant findings ($p < 0.05$) and their implications.

Table 14
Wilcoxon signed-rank test (paired) per prompt for each model.

Metric	Llama3.2:1b	Gemma3:1b	Granite3.1-MoE:1b	Qwen2.5:0.5b
total_duration_ms	0.0583	0.0266	0.6742	0.0153
load_duration_ms	0.4524	0.3300	0.8408	0.2943
prompt_eval_count	1.0000	1.0000	1.0000	1.0000
prompt_eval_duration_ms	0.0121	0.1650	0.0073	0.7285
eval_count	1.0000	1.0000	0.7150	1.0000
eval_duration_ms	0.0121	0.0172	0.6742	0.0759
tokens_per_sec	0.0370	0.0056	0.3436	0.0400

For the Llama3.2:1b model, the total duration approached significance ($p = 0.0583$), suggesting a trend toward faster end-to-end inference in the Rust client, but it failed to cross the 0.05 threshold. Load duration differences were non-significant ($p = 0.4524$), indicating equivalent model initialisation times. Prompt evaluation and overall evaluation counts tied by design ($p = 1.0000$). However, prompt evaluation duration ($p = 0.0121$) and full evaluation duration ($p = 0.0121$) were both significantly lower for one client, with Rust executing slightly faster once the model was loaded. Tokens per second also favoured Rust ($p = 0.0370$), confirming a modest but consistent throughput advantage.

Gemma3:1b revealed even bigger paired differences. Total duration favoured Rust ($p = 0.0266$), while load duration again showed no significant disparity ($p = 0.3300$). Prompt evaluation count and overall eval count remained identical ($p = 1.0000$), but full evaluation duration was significantly lower in Rust ($p = 0.0172$). Throughput differences were highly significant ($p = 0.0056$), with Rust delivering a higher median tokens per second. Prompt evaluation duration for Gemma3:1b did not reach significance ($p = 0.1650$), suggesting that early decoding costs were comparable between clients.

In contrast, Granite3.1-MoE:1b exhibited no significant differences in total or load durations ($p = 0.6742$ and $p = 0.8408$, respectively), indicating similar end-to-end and initialisation performance. The prompt evaluation duration was significantly lower in the Rust client ($p = 0.0073$), reflecting faster expert-module invocation under MoE routing. However, eval count and eval duration were non-significant ($p = 0.7150$ and $p = 0.6742$), and the differences in tokens per second failed to reach the threshold ($p = 0.3436$). These results suggest that for MoE architectures, language-level overhead differences primarily manifest in the prompt-shaping stage rather than in the bulk inference loop. For the smallest Qwen2.5:0.5b model, total duration differences achieved significance ($p = 0.0153$), with Rust again outperforming Python in end-to-end latency. Load duration remained non-significant ($p = 0.2943$). Prompt and evaluation counts tied ($p = 1.0000$). The prompt evaluation duration did not differ significantly ($p = 0.7285$), but the full evaluation duration showed a trend favouring Rust ($p = 0.0759$) that did not meet the strict 0.05 cutoff. Nevertheless, tokens per second were significantly higher in Rust ($p = 0.0400$), confirming the pattern of modest throughput gains even for compact models.

Across all models, load durations rarely differed, reflecting that both clients handle static model loading equivalently. Significant differences consistently emerged in prompt-evaluation duration, full evaluation duration, and tokens per second for three of the four models. The Rust client's lower median durations and higher throughput indicate that its system-level optimisations and absence of interpreter overhead yield measurable benefits once the model is engaged. For the MoE case, gains were concentrated in the prompt-analysis stage, highlighting the sensitivity of dynamic expert selection to client implementation. In summary, the Wilcoxon signed-rank tests confirm that while both Python and Rust clients produce identical token workloads, Rust exhibits a reliable advantage in execution speed, particularly in the core inference stages, underscoring its suitability for latency-critical edge deployments.

5.12. Bootstrap confidence intervals for mean API differences

To complement our hypothesis tests, we employed a non-parametric bootstrap approach to estimate 95% confidence intervals for the mean difference between Python and Rust API performance on each metric. By resampling the paired differences across the twenty prompts 10,000 times and computing the mean for each bootstrap sample, we obtained percentile-based intervals that make no assumptions about normality or equal variances. Table 15 lists the resulting intervals for total duration, load duration, prompt and eval counts, prompt and eval durations, and tokens per second for each of the four quantised models.

Table 15
Bootstrap 95% confidence intervals for mean difference (Python – Rust) per metric.

Metric	Llama3.2:1b	Gemma3:1b	Granite3.1-MoE:1b	Qwen2.5:0.5b
total_duration_ms	$[-15502.31, 17972.11]$	$[-22711.55, 25639.34]$	$[-7859.27, 10812.54]$	$[-4487.34, 5464.18]$
load_duration_ms	$[-0.32, 4787.77]$	$[-253.88, 3077.42]$	$[-494.02, 4917.10]$	$[-338.57, 1658.13]$
prompt_eval_count	$[-1.95, 1.95]$	$[-2.25, 2.15]$	$[-2.10, 2.05]$	$[-2.20, 2.20]$
prompt_eval_duration_ms	$[-230.38, 147.24]$	$[-125.81, 176.16]$	$[-141.24, 170.88]$	$[-127.36, 127.21]$
eval_count	$[-44.40, 44.30]$	$[-98.25, 100.61]$	$[-61.20, 60.80]$	$[-39.40, 39.95]$
eval_duration_ms	$[-17182.76, 16641.40]$	$[-24182.37, 24794.57]$	$[-8946.80, 8914.59]$	$[-4664.62, 4807.47]$
tokens_per_sec	$[-0.06, 0.08]$	$[-0.47, 0.59]$	$[-0.10, 0.15]$	$[-0.26, 0.12]$

For Llama3.2:1b, the 95% bootstrap interval for total duration spans from -15,502 ms to +17,972 ms. This wide interval encompasses zero, indicating no reliable difference in end-to-end latency between the Python and Rust clients under sampling variability. Load duration returns an interval of -0.32 ms to +4,787.77 ms, again crossing zero, which implies that any apparent advantage in model-loading speed is not statistically robust. The prompt evaluation count and overall evaluation count intervals ($\pm \approx 2$ tokens and $\pm \approx 44$ tokens, respectively) also include zero, confirming that both clients request the same token workloads. The interval for prompt evaluation duration (-230.38 ms to +147.24 ms) and for full eval duration (-17,182.76 ms to +16,641.40 ms) also straddle zero, as does the tokens-per-second interval (-0.06 to +0.08 tokens/s). Collectively, none of the CIs exclude zero, indicating that for Llama3.2:1b, client-language choice does not produce a consistent mean difference in any metric. Gemma3:1b exhibits even broader bootstrap intervals. Total duration lies between -22,711 ms and +25,639 ms, and load duration between -253.88 ms and +3,077.42 ms. Both intervals include zero, as do those for prompt evaluation ($\pm \approx$2 tokens; -125.81 ms to +176.16 ms), full eval count ($\pm \approx$100 tokens; -24,182 ms to +24,794 ms), and throughput (-0.47 to +0.59 tokens/s). These wide spans reflect greater variability in inference timing for Gemma3:1b on the Raspberry Pi, but still offer no evidence of systematic client-based performance differences.

The Granite3.1-MoE:1b model yields narrower but still zero-crossing intervals: total duration (-7,859 ms to +10,812 ms), load duration (-494.02 ms to +4,917.10 ms), and prompt evaluation duration (-141.24 ms to +170.88 ms). Token counts and eval durations likewise produce intervals that include zero, as does throughput (-0.10 to +0.15 tokens/s). The consistency of zero-inclusive intervals across both count and time metrics confirms that dynamic Mixture-of-Experts routing does not amplify client-side performance disparities. Finally, the smallest model Qwen2.5:0.5b shows the tightest intervals, yet still none exclude zero. Total duration spans -4,487 ms to +5,464 ms, and load duration -338.57 ms to +1,658.13 ms. Prompt eval counts vary by no more than $\pm$ 2.20 tokens, prompt durations by $\pm$ 127 ms, and eval durations by $\pm$ 4,800 ms. Throughput differences lie between -0.26 and +0.12 tokens/s. These results indicate that even for a compact 0.5 billion-parameter model, Python and Rust clients yield statistically indistinguishable mean performance.

The bootstrap confidence intervals for all models and metrics consistently span zero, indicating that any observed mean differences between the Python and Rust API implementations are likely due to sampling variability rather than intrinsic language-level overheads. This nonparametric analysis reinforces the conclusion that both client environments perform equivalently in practice, and that choices between Python and Rust can be guided more by integration and ecosystem considerations than by raw inference speed on edge hardware.

5.13. Paired *t*-tests and effect sizes

To quantify the magnitude and significance of performance differences between the Python and Rust APIs on a per-model basis, we performed paired Student's t-tests for each metric and calculated Cohen's d as a standardised effect size. Table 16 contains the detailed statistics; here we highlight key findings. For the 1.24 billion-parameter Llama3.2:1b model, total end-to-end latency did not differ significantly (t = 0.851, p = 0.405) and exhibited a negligible effect size (d = 0.051). Model load time similarly showed no significance (t = 1.000, p = 0.330, d = 0.316). Prompt evaluation and eval counts were identical by design. Prompt evaluation duration also failed to reach significance (t = -0.678, p = 0.506, d = -0.153). However, full evaluation duration was significantly longer in the Rust client (t = -3.359, p = 0.0033), although the effect size was essentially zero (d =-0.005), indicating a statistically detectable but practically negligible difference. Token throughput favoured Python, with a highly significant t-statistic (t = 4.472, p = 0.00026) and a diminutive positive Cohen's d of 0.086.

For the 1.0 billion-parameter Gemma3:1b, neither total duration (t = 1.563, p = 0.134, d = 0.034) nor load duration (t = 1.144, p = 0.267, d = 0.313) was significantly different. Prompt evaluation duration also did not differ (t = 0.958, p = 0.350, d = 0.106). Full evaluation duration was significantly lower for the Rust client (t = 2.789, p = 0.0117), but again with a vanishingly small Cohen's d of 0.008. Token throughput differences were non-significant (t = 0.381, p = 0.708, d = 0.028).

In the 1.33 billion-parameter Mixture-of-Experts model, total duration (t = 0.956, p = 0.351, d = 0.091) and load duration (t = 0.999, p = 0.330, d = 0.283) showed no significant differences. The prompt evaluation duration, however, was significantly shorter in the Rust client (t = 3.287, p = 0.0039), with a Cohen's d of 0.043, indicating a small effect. Full evaluation duration and token throughput differences did not reach significance (p = 0.100 and p = 0.075, with d = -0.005 and d = 0.100, respectively).

For the compact Qwen2.5:0.5b model, total duration showed no significant difference (t = 1.176, p = 0.254, d = 0.061). Load duration again was non-significant (t = 0.999, p = 0.330, d = 0.244). Prompt evaluation duration remained similar (t = -0.118, p = 0.907, d = -0.002). The full evaluation duration was significantly lower for Rust (t = 3.130, p = 0.0055), with a small effect size (d = 0.008). Token throughput

Table 16
Paired t-test and effect size (Cohen's d) between Python and Rust APIs.

Metric	Python mean	Rust mean	Paired t-stat	p-value	Cohen's d	Sig.
llama3.2:1b						
total_duration_ms	30853.70	29432.07	0.851	0.4055	0.0509	
load_duration_ms	1648.68	52.83	1.000	0.3299	0.3162	
prompt_eval_count	33.85	33.85	NaN	NaN	0.0000	
prompt_eval_duration_ms	1143.06	1190.45	-0.678	0.5062	-0.1527	
eval_count	74.85	74.85	NaN	NaN	0.0000	
eval_duration_ms	28060.49	28187.39	-3.359	0.0033	-0.0046	*
tokens_per_sec	2.747	2.7365	4.472	0.0003	0.0856	*
gemma3:1b						
total_duration_ms	35658.78	34298.16	1.563	0.1345	0.0339	
load_duration_ms	1236.04	253.17	1.144	0.2667	0.3131	
prompt_eval_count	17.95	17.95	NaN	NaN	0.0000	
prompt_eval_duration_ms	1069.84	1043.86	0.958	0.3499	0.1058	
eval_count	138.85	138.85	NaN	NaN	0.0000	
eval_duration_ms	33332.98	33007.20	2.789	0.0117	0.0081	*
tokens_per_sec	4.443	4.4195	0.381	0.7075	0.0284	
granite3.1-moe:1b						
total_duration_ms	16652.45	15238.62	0.956	0.3512	0.0908	
load_duration_ms	1658.34	184.05	1.000	0.3299	0.2830	
prompt_eval_count	53.50	53.50	NaN	NaN	0.0000	
prompt_eval_duration_ms	586.21	575.00	3.287	0.0039	0.0435	*
eval_count	103.70	103.70	NaN	NaN	0.0000	
eval_duration_ms	14406.48	14478.09	-1.730	0.0998	-0.0049	
tokens_per_sec	7.348	7.3275	1.882	0.0753	0.0996	
qwen2.5:0.5b						
total_duration_ms	10628.11	10129.37	1.176	0.2540	0.0611	
load_duration_ms	607.03	171.29	0.999	0.3305	0.2444	
prompt_eval_count	38.10	38.10	NaN	NaN	0.0000	
prompt_eval_duration_ms	586.11	586.49	-0.118	0.9073	-0.0018	
eval_count	79.20	79.20	NaN	NaN	0.0000	
eval_duration_ms	9433.50	9370.09	3.130	0.0055	0.0080	*
tokens_per_sec	8.556	8.620	-3.086	0.0061	-0.2071	

significantly favoured Rust (t = -3.086, p = 0.0061) and exhibited a small negative Cohen's d (-0.207), indicating a modest throughput advantage.

Across all four models, the vast majority of metrics did not differ significantly between Python and Rust clients. Where statistical significance was observed – principally in evaluation durations and token throughput – Cohen's d values remained below 0.25, denoting small or negligible practical effects. These results suggest that, despite occasionally measurable differences in sub-stage latencies or decoding rates, both API implementations deliver largely equivalent performance on resource-constrained Raspberry Pi hardware. Choices between Python and Rust are thus better guided by ecosystem, development productivity, and integration needs rather than raw inference speed alone.

5.14. Kolmogorov-Smirnov test for distributional differences

The Kolmogorov-Smirnov (KS) test was applied to compare the empirical distributions of each performance metric for the Python and Rust client implementations on a per-prompt basis. Unlike tests of central tendency, the KS test assesses whether two samples originate from the same continuous distribution by measuring the maximum difference between their empirical cumulative distribution functions (ECDFs). A high p-value indicates that the observed differences are consistent with random sampling from a standard distribution, whereas a low p-value suggests a genuine distributional shift.

Table 17 summarises the KS statistics, p-values, and a Boolean flag indicating whether the distributions differ significantly at the 5% level for each metric and model. As shown, only one out of twenty-eight comparisons yielded a significant KS result: the prompt evaluation duration for the llama3.2:1b model (KS statistic = 0.50, $p = 0.0123$). This indicates that the distribution of prompt-processing times differs between the two clients for this particular model, likely due to differences in how each language handles JSON parsing and buffering of streaming data. All other metrics across all models returned p-values well above 0.05, confirming that their empirical distributions overlap closely.

Table 17
Kolmogorov-Smirnov test for distribution differences between Python and Rust APIs.

LLM model	Metric	KS-stat	*p*-value	Different dist.
llama3.2:1b	total_duration_ms	0.10	0.999992	False
llama3.2:1b	load_duration_ms	0.15	0.983137	False
llama3.2:1b	prompt_eval_count	0.00	1.000000	False
llama3.2:1b	prompt_eval_duration_ms	0.50	0.012299	**True**
llama3.2:1b	eval_count	0.00	1.000000	False
llama3.2:1b	eval_duration_ms	0.10	0.999992	False
llama3.2:1b	tokens_per_sec	0.15	0.983137	False
gemma3:1b	total_duration_ms	0.10	0.999992	False
gemma3:1b	load_duration_ms	0.15	0.983137	False
gemma3:1b	prompt_eval_count	0.00	1.000000	False
gemma3:1b	prompt_eval_duration_ms	0.20	0.831970	False
gemma3:1b	eval_count	0.00	1.000000	False
gemma3:1b	eval_duration_ms	0.10	0.999992	False
gemma3:1b	tokens_per_sec	0.30	0.335591	False
granite3.1-moe:1b	total_duration_ms	0.10	0.999992	False
granite3.1-moe:1b	load_duration_ms	0.25	0.571336	False
granite3.1-moe:1b	prompt_eval_count	0.00	1.000000	False
granite3.1-moe:1b	prompt_eval_duration_ms	0.15	0.983137	False
granite3.1-moe:1b	eval_count	0.00	1.000000	False
granite3.1-moe:1b	eval_duration_ms	0.05	1.000000	False
granite3.1-moe:1b	tokens_per_sec	0.15	0.983137	False
qwen2.5:0.5b	total_duration_ms	0.10	0.999992	False
qwen2.5:0.5b	load_duration_ms	0.40	0.081058	False
qwen2.5:0.5b	prompt_eval_count	0.00	1.000000	False
qwen2.5:0.5b	prompt_eval_duration_ms	0.15	0.983137	False
qwen2.5:0.5b	eval_count	0.00	1.000000	False
qwen2.5:0.5b	eval_duration_ms	0.10	0.999992	False
qwen2.5:0.5b	tokens_per_sec	0.25	0.571336	False

For total end-to-end latency, the KS statistic remained at 0.10 with $p \approx 1.0$ for all models, indicating that Python and Rust clients produce nearly identical cumulative performance profiles. The same holds for model load times, token counts, full evalua-

tion durations, and throughput rates; in each case, the KS statistics are small (≤ 0.40), and the corresponding p-values far exceed the significance threshold. Even for the smallest Qwen2.5:0.5b model, where dynamic memory allocation overhead could differ substantially between Python and Rust, the KS test did not detect distributional differences in load or decoding metrics.

In practical terms, these results reinforce that the client implementation language has a negligible impact on the stochastic behaviour of inference timing on a resource-constrained Raspberry Pi. Except for a single sub-stage anomaly in Llama3.2 prompt evaluation, both Python and Rust clients yield statistically indistinguishable performance distributions. This implies that system designers and researchers can choose either client API based on development convenience or ecosystem support without compromising the consistency or predictability of LLM inference on edge hardware.

5.15. Clustering of per-prompt performance profiles

To uncover latent patterns in how different prompts stress each quantised model and client implementation, we applied three unsupervised clustering algorithms – k-means, agglomerative Hierarchical Clustering, and Gaussian mixture models (GMM) – to the seven-dimensional vectors of performance metrics for each model. Metrics included total duration, load duration, prompt evaluation count and duration, evaluation count and duration, and tokens per second, all standardised to zero mean and unit variance prior to clustering. For each model and method, we determined the optimal number of clusters k by maximising the average silhouette score over a range of k values from 2 to 12. The silhouette coefficient measures how well each sample lies within its cluster (values near +1 indicate clear cluster membership, values near 0 indicate overlapping clusters, and negative values indicate potential misassignment).

Table 18 summarises the optimal k and corresponding silhouette score for each combination of model and clustering algorithm.

Table 18
Optimal cluster number and silhouette scores by model and method.

LLM model	Method	Optimal k	Silhouette
llama3.2:1b	k-means	9	0.540
llama3.2:1b	agglomerative	4	0.560
llama3.2:1b	Gaussian mixture model	8	0.527
gemma3:1b	k-means	5	0.610
gemma3:1b	agglomerative	5	0.610
gemma3:1b	Gaussian mixture model	9	0.547
granite3.1-moe:1b	k-means	3	0.617
granite3.1-moe:1b	agglomerative	3	0.617
granite3.1-moe:1b	Gaussian mixture model	2	0.591
qwen2.5:0.5b	k-means	2	0.504
qwen2.5:0.5b	agglomerative	2	0.651
qwen2.5:0.5b	Gaussian mixture model	2	0.543

For llama3.2:1b, agglomerative clustering yielded the highest silhouette score of 0.560 at $k = 4$, indicating that prompts can be partitioned into four distinct performance regimes – perhaps reflecting prompt complexity or token count variations. K-means produced a slightly lower maximum silhouette of 0.540 at $k = 9$, suggesting finer granularity but more overlap between clusters. The GMM approach identified eight clusters, with a silhouette score of 0.527, underscoring the benefit of modelling soft cluster assignments. However, it also revealed that prompts do not cleanly separate into a small number of Gaussian modes. The gemma3:1b model displayed a consistent optimum of $k = 5$ for both k-means and agglomerative methods, each

achieving a silhouette of 0.610 – the highest across all methods and models – indicating well-defined clusters. The GMM optimal $k = 9$ delivered a respectable silhouette of 0.547, but lower than the deterministic methods, suggesting that hard assignments more naturally partition gemma3:1b's performance profiles.

For the MoE-based granite3.1-moe:1b, both k-means and agglomerative agree on $k = 3$ with a silhouette of 0.617, the second-highest score overall. This implies three characteristic prompt-behaviour classes, likely corresponding to short, medium, and long token sequences that trigger varying expert-module activations. GMM's optimal two clusters (silhouette 0.591) reflect a simpler dichotomy but with reduced cluster cohesion. Finally, the smallest qwen2.5:0.5b model achieved its strongest cluster separation under agglomerative clustering with $k = 2$ and a silhouette of 0.651, indicating two primary performance modes — perhaps low-latency versus high-throughput prompt categories. K-means also selected $k = 2$ but with a lower silhouette of 0.504, while GMM matched $k = 2$ with an intermediate silhouette of 0.543. Agglomerative clustering tends to produce more coherent clusters in this setting, especially for the smallest and mid-sized models. In contrast, k-means and GMM require more clusters to achieve a similar silhouette quality. These results provide a nuanced understanding of how prompt characteristics map onto inference performance regimes, informing both prompt selection strategies and adaptive scheduling for edge deployments.

5.16. Cold-start versus warm-start combined analysis

We analysed cold-start and warm-start behaviours for both Python and Rust clients across the four quantised models, focusing on two key metrics: total end-to-end duration and model load duration. In a cold start, the model is loaded into memory immediately prior to inference, whereas in a warm start, the model remains resident from a preceding request. We report both mean times and the cold/warm ratio (mean cold time divided by mean warm time) to quantify initialisation overhead.

For the total duration, the cold/warm ratios reveal that cold starts add between roughly 0.33× and 3.26× overhead, depending on the model and client. Gemma3:1b exhibits a ratio of 0.81 for the Python client and 0.33 for Rust, indicating a moderate penalty for Python relative to warm performance, but a much more minor penalty for Rust. Granite3.1-MoE shows a more pronounced effect: Python's cold runs take over three times longer (ratio 3.26), whereas Rust's cold start is about 1.28× slower. This reflects the cost of loading multiple expert modules into memory in the MoE architecture. Llama3.2:1b cold starts incur a 1.79× delay in Python versus only 0.67× in Rust, and Qwen2.5:0.5b shows a 2.00× and 1.17× ratio, respectively. Overall, Rust's lower ratios across all models demonstrate its superior efficiency in managing binary loading and HTTP request setup under constrained RAM and CPU on the Raspberry Pi.

Model load duration further isolates the initialisation phase. Here, Python's cold/warm ratios range from 76.3× for Gemma3:1b up to 1,700× for Granite3.1-MoE, indicating that Python's JSON parsing, file I/O, and memory allocation incur substantial one-time costs. By contrast, Rust's ratios remain an order of magnitude lower – 21.1× for Gemma3:1b and 171× for Granite3.1-MoE – and near unity for Llama3.2:1b (1.01×), demonstrating that Rust's blocking HTTP client and direct file-mapping significantly reduce cold-start overhead. Even for the smallest model, Qwen2.5:0.5b, Rust's ratio (42.4×) is markedly lower than Python's (200.8×).

Prompt evaluation count and overall evaluation count remain stable across cold and warm starts (ratios ≈0.89-0.97), reflecting that token counts are unaffected by loading state. Prompt evaluation duration ratios are more variable: Python's ratios span 1.26× to 3.09×, while Rust's span 1.19× to 2.99×, again highlighting greater initialisation overhead for Python in the prompt-processing stage. Eval duration cold/warm ratios

for Gemma3:1b and Qwen2.5:0.5b are low (≈0.23-0.83), indicating that cold-start effects largely vanish during the full-model evaluation phase. In contrast, for MoE and Llama3.2, ratios hover near unity, showing consistent inference times once the model is warmed.

Finally, the number of tokens per second is nearly identical for both cold and warm starts, with ratios close to 1.0 in all cases. This stability indicates that decoding throughput is unaffected by whether the model was just loaded or already resident, and that the dominant factor for tokens per second is the main inference loop rather than initialisation overhead.

Thus, cold-start overhead is most pronounced in model loading, especially for Python, and is significantly mitigated by Rust's efficient I/O and memory management. Once the model is warm, both clients deliver stable, prompt and evaluation performance, with decoding throughput unaffected by prior state. These findings underscore the importance of warm-up strategies – such as keeping the model resident – to minimise latency spikes in edge deployments. This combined table 19 shows that tart overhead varies dramatically by metric and client.

Total duration cold/warm ratios range from 0.33× (Rust on gemma3:1b) up to 3.26× (Python on granite3.1-moe), illustrating that keeping the model warm significantly reduces overall latency. Model load durations incur the largest penalties – Python cold/warm ratios exceed 200× for most models, whereas Rust ratios remain below 200× and near unity for Llama3.2:1b. Prompt and evaluation counts are unaffected by cold versus warm state (ratios ≈ 0.67–1.00), confirming static token workloads. Prompt evaluation durations show moderate cold/warm inflation (ratios 1.19–3.09 in most cases). In contrast, full evaluation durations generally return ratios below 1.00 for smaller models, indicating that inference time dominates once the model is loaded. Tokens per second remain stable (ratios ≈ 0.95–0.99), demonstrating consistent throughput regardless of startup state. Thus, Rust's lower cold/warm ratios highlight its efficiency in initialisation, and warm-start strategies are essential for minimising latency spikes in edge deployments.

5.17. Model- and client-specific performance comparison

Table 20 presents a detailed comparison of mean performance metrics for each of the four quantised models – Llama3.2:1b, Gemma3:1b, Granite3.1-MoE:1b, and Qwen2.5:0.5b – when accessed via the Python and Rust API clients. For each model and metric, we identify which client achieved better performance: lower latency metrics favor Rust, while higher throughput metrics favor Python. Metrics where both clients perform identically are marked as ties.

Overall latency, represented by total duration, consistently favours the Rust client across all models. Rust reduces end-to-end latency by approximately 1.4 seconds for Llama3.2:1b, 1.4 seconds for Gemma3:1b, 1.4 seconds for Granite3.1-MoE:1b, and around 0.5 seconds for the lightweight Qwen2.5:0.5b. This uniform advantage suggests that the Rust client imposes lower overhead on the HTTP request and JSON handling pipeline, likely due to its zero-cost abstractions and optimised I/O. Model load time similarly exhibits a clear Rust advantage: Rust's mean load durations are under 200 milliseconds for all models, compared to Python's load times, which range from approximately 600 milliseconds for Qwen2.5:0.5b up to 1.65 seconds for Llama3.2:1b. The drastic reduction in load overhead underscores Rust's efficiency in file and memory operations on constrained hardware. For token counts – both prompt evaluation count and overall evaluation count – clients tie across every model, reflecting identical inference logic and prompt processing. This parity confirms that both clients construct identical payloads and request the same amount of work from the LLMs. In sub-stage durations, the pattern is mixed. Python edges out Rust in

Table 19
Cold and warm start times with ratios for all metrics.

Model	Client	Metric	Cold (ms)	Warm (ms)	Ratio
gemma3:1b	Python API	total_duration_ms	29116.01	36003.14	0.809
		load_duration_ms	19790.11	259.51	76.259
		prompt_eval_count	16.00	18.05	0.886
		prompt_eval_duration_ms	1327.65	1056.28	1.257
		eval_count	34.00	144.37	0.236
		eval_duration_ms	7995.17	34666.55	0.231
		tokens_per_sec	4.25	4.45	0.954
gemma3:1b	Rust API	total_duration_ms	11787.82	35482.92	0.332
		load_duration_ms	2664.56	126.26	21.104
		prompt_eval_count	16.00	18.05	0.886
		prompt_eval_duration_ms	1232.90	1033.91	1.192
		eval_count	34.00	144.37	0.236
		eval_duration_ms	7888.48	34329.24	0.230
		tokens_per_sec	4.31	4.43	0.974
granite3.1-moe:1b	Python API	total_duration_ms	48728.50	14964.24	3.256
		load_duration_ms	32800.25	19.29	1700.210
		prompt_eval_count	52.00	53.58	0.971
		prompt_eval_duration_ms	1638.82	530.81	3.087
		eval_count	104.00	103.68	1.003
		eval_duration_ms	14287.64	14412.74	0.991
		tokens_per_sec	7.28	7.35	0.990
granite3.1-moe:1b	Rust API	total_duration_ms	19218.75	15029.14	1.279
		load_duration_ms	3313.10	19.36	171.143
		prompt_eval_count	52.00	53.58	0.971
		prompt_eval_duration_ms	1564.74	522.91	2.992
		eval_count	104.00	103.68	1.003
		eval_duration_ms	14339.20	14485.40	0.990
		tokens_per_sec	7.25	7.33	0.989
llama3.2:1b	Python API	total_duration_ms	53230.50	29675.97	1.794
		load_duration_ms	31969.13	52.86	604.744
		prompt_eval_count	32.00	33.95	0.943
		prompt_eval_duration_ms	2322.88	1080.97	2.149
		eval_count	51.00	76.11	0.670
		eval_duration_ms	18936.29	28540.71	0.663
		tokens_per_sec	2.69	2.75	0.978
llama3.2:1b	Rust API	total_duration_ms	20067.37	29924.95	0.671
		load_duration_ms	53.51	52.79	1.014
		prompt_eval_count	32.00	33.95	0.943
		prompt_eval_duration_ms	1045.90	1198.06	0.873
		eval_count	51.00	76.11	0.670
		eval_duration_ms	18966.66	28672.69	0.661
		tokens_per_sec	2.69	2.74	0.982
qwen2.5:0.5b	Python API	total_duration_ms	20279.15	10120.16	2.004
		load_duration_ms	11091.21	55.23	200.828
		prompt_eval_count	36.00	38.21	0.942
		prompt_eval_duration_ms	1294.56	548.82	2.359
		eval_count	67.00	79.84	0.839
		eval_duration_ms	7891.55	9514.66	0.829
		tokens_per_sec	8.49	8.56	0.992
qwen2.5:0.5b	Rust API	total_duration_ms	11730.99	10045.07	1.168
		load_duration_ms	2365.83	55.78	42.410
		prompt_eval_count	36.00	38.21	0.942
		prompt_eval_duration_ms	1347.59	546.43	2.466
		eval_count	67.00	79.84	0.839
		eval_duration_ms	8015.73	9441.38	0.849
		tokens_per_sec	8.36	8.63	0.968

Table 20
Client performance by model and metric.

Model	Metric	Python mean	Rust mean	Winner
Llama3.2:1b	total duration (ms)	30853.70	29432.07	Rust
	load duration (ms)	1648.68	52.83	Rust
	prompt evaluation count	33.85	33.85	Tie
	prompt evaluation duration (ms)	1143.06	1190.45	Python
	eval count	74.85	74.85	Tie
	eval duration (ms)	28060.49	28187.39	Python
	tokens per second	2.75	2.74	Python
Gemma3:1b	total duration (ms)	35658.78	34298.16	Rust
	load duration (ms)	1236.04	253.17	Rust
	prompt evaluation count	17.95	17.95	Tie
	prompt evaluation duration (ms)	1069.84	1043.86	Rust
	eval count	138.85	138.85	Tie
	eval duration (ms)	33332.98	33007.20	Rust
	tokens per second	4.44	4.42	Python
Granite3.1-MoE:1b	total duration (ms)	16652.45	15238.62	Rust
	load duration (ms)	1658.34	184.05	Rust
	prompt evaluation count	53.50	53.50	Tie
	prompt evaluation duration (ms)	586.21	575.00	Rust
	eval count	103.70	103.70	Tie
	eval duration (ms)	14406.48	14478.09	Python
	tokens per second	7.35	7.33	Python
Qwen2.5:0.5b	total duration (ms)	10628.11	10129.37	Rust
	load duration (ms)	607.03	171.29	Rust
	prompt evaluation count	38.10	38.10	Tie
	prompt evaluation duration (ms)	586.11	586.49	Python
	eval count	79.20	79.20	Tie
	eval duration (ms)	9433.50	9370.09	Rust
	tokens per second	8.56	8.62	Rust

prompt evaluation duration for Llama3.2:1b (1.14 vs. 1.19 s) and Ruby's tokens per second is higher by a slim margin (2.75 vs. 2.74 tokens/s). For Gemma3:1b, Rust slightly shortens prompt evaluation and full evaluation durations, but Python again achieves marginally higher tokens per second (4.44 vs. 4.42 tokens/s). Granite3.1-MoE:1b follows the same mixed pattern: Rust excels in prompt evaluation (0.575 vs. 0.586 s), while Python wins in full evaluation duration (14.78 vs. 14.41 s) and throughput (7.35 vs. 7.33 tokens/s). Finally, Qwen2.5:0.5b demonstrates Rust's superiority in full evaluation duration (9.37 vs. 9.43 s) and throughput (8.62 vs. 8.56 tokens/s), although Python matches Rust in prompt evaluation time. These results demonstrate that, while Rust consistently reduces start-up and end-to-end latencies across all models, Python remains competitive in the core inference loop, often achieving slightly higher decoding throughput and, in some cases, marginally faster sub-stage durations. For applications where rapid initialisation and minimal total latency are critical, Rust is the clear choice. However, for scenarios that prioritise sustained token throughput or easier integration with high-level analytics, the Python client remains a viable option.

6. Discussion

6.1. Limitations of the existing study

Although this work provides a thorough examination of edge-deployed, quantised LLMs accessed via Python and Rust clients on a Raspberry Pi 4 B, several inherent constraints limit its generality. The evaluation platform remains fixed to a single hardware configuration – namely, an 8 GB, quad-core ARM Cortex-A72 device – thus omitting other prevalent edge compute targets such as NVIDIA Jetson modules or Google Coral accelerators. Variations in memory subsystems, cache hierarchies, and thermal management among these platforms could substantially alter cold-start latencies, throughput figures, and thermal throttling behaviour [15, 26].

Model selection itself represents a further boundary. We restrict analysis to four quantised architectures – Llama 3.2:1b, Gemma 3:1b, Granite 3.1-MoE:1b, and Qwen 2.5:0.5b – each employing either 4-bit or 8-bit weight encodings. Emerging techniques such as mixed 2-bit quantisation, structured pruning, or dynamic sparsity remain unexplored. As these schemes can affect both memory footprint and inference speed, the performance characteristics observed here may shift when newer compression or sparsification methods are applied.

Our prompt workload, comprising twenty concise questions covering factual retrieval, arithmetic reasoning, translation, code synthesis, and creative text, intentionally avoids long-form or highly contextual inputs to prevent memory paging on the Pi. Consequently, applications that demand multi-sentence summarisation, extended dialogue, or multimodal inputs (e.g., image-to-text) fall outside the scope of current measurements. In such scenarios, tokeniser overhead, context window expansion, or additional preprocessing time may introduce performance bottlenecks not captured in this benchmark [48].

The experimental protocol enforces a fixed two-second idle interval between requests and prohibits concurrent client threads. Real-world edge deployments often involve parallel inference streams, dynamic request bursts, or adaptive sleep strategies that can lead to I/O contention or CPU scheduling variability. By contrast, our single-threaded, static-delay approach likely underestimates the impact of simultaneous workloads or asynchronous I/O, which could exacerbate latency spikes and throughput degradation under production loads [54, 56].

From a statistical perspective, the reliance on frequentist hypothesis tests – paired t-tests, ANOVA, nonparametric rank tests – yields robust group-level insights. However, it does not quantify uncertainty in a probabilistic framework. Bayesian hierarchical modelling, sequential analysis, or change-point detection techniques could provide more nuanced assessments of performance drift, measurement noise, and outlier behaviour over time. Additionally, the unsupervised clustering analysis standardises all metrics without integrating model-specific features (e.g., expert-module activation counts) that might enhance interpretability and guide prompt scheduling strategies.

Finally, this study focuses exclusively on raw inference performance, deliberately omitting quality metrics such as semantic accuracy, hallucination rates, or human-evaluated coherence. In practical applications, the trade-off between latency and output fidelity is paramount. Without simultaneous evaluation of model correctness – through BLEU scores, code execution tests, or human preference studies – deployers must make assumptions about the acceptability of quantised model outputs. Future work should incorporate end-to-end task success rates to align performance optimisations with application-level efficacy [51].

6.2. Future scope

Extending this investigation invites multiple promising directions that can enhance both the breadth and depth of understanding of edge LLM performance. Exploring

heterogeneous accelerator integration is a top priority. By benchmarking [4] identical quantised models on devices such as Google Coral Edge TPUs, Intel Movidius Myriad accelerators, and NVIDIA Jetson GPUs, researchers could isolate how specialised tensor-processing units alter inference latency, power consumption, and temperature profiles. Such comparisons would inform design guidelines for selecting appropriate hardware for workload-specific deployments.

Advances in memory management also warrant deeper study [2, 20]. Techniques like model weight prefetching, warm-pooling services, or on-device weight caching – where only frequently accessed tensor blocks remain resident – could dramatically reduce cold-start overhead. Integrating these methods within container orchestration frameworks (for example, lightweight Kubernetes distributions on ARM) would enable the dynamic scaling of warm instances based on real-time demand, thereby mitigating latency spikes during peak usage.

Expanding the workload to include multimodal and longer-form tasks constitutes another rich vein of inquiry. Assessing quantised vision-language models for on-device image captioning, or audio-language models for real-time transcription, calls for measuring both the CNN or spectrogram preprocessing cost and the accompanying transformer decode times. Evaluating hierarchical or retrieval-augmented generation on edge nodes would further illuminate how memory and compute constraints influence knowledge-intensive applications.

Adaptive batching and request scheduling algorithms present an additional frontier [33, 57]. By grouping incoming prompts into micro-batches or dynamically adjusting inter-request delays based on system load and thermal headroom, developers can optimise throughput without sacrificing latency guarantees. Implementing feedback-driven controllers that monitor real-time CPU utilisation and queue lengths could enable responsive resource allocation, crucial for scenarios with mixed-priority requests.

On-device personalisation via parameter-efficient fine-tuning techniques – such as low-rank adapters or quantised LoRA [24, 47] – offers the prospect of continuously improving model relevance without offloading data to the cloud. Investigating the performance impact of incremental updates, under constrained bandwidth and power, will inform privacy-preserving applications where model adaptation must occur locally.

Ultimately, adopting advanced statistical and machine learning methods for performance analysis can yield more comprehensive insights. Bayesian A/B testing frameworks [5], sequential change-point detection, and causal inference models can quantify the impact of system updates or configuration changes on latency distributions. Integrating explainable clustering methods that incorporate both performance metrics and model architecture features would enable automated model-selection pipelines that match prompt characteristics to optimal deployment strategies.

6.3. Prospective use cases of this work

The performance profiles and client-comparison insights derived here can directly guide real-world applications across diverse domains. In precision agriculture, edge gateways equipped with quantised LLMs can interpret sensor streams – soil moisture, temperature, humidity – and generate real-time advisories for irrigation or pest control [23, 36]. By pre-warming models via Rust clients during critical periods (e.g., dawn moisture checks) and leveraging Python for scheduled analytic reports, farm managers achieve both rapid alerting and flexible data integration.

Healthcare monitoring represents another domain where on-device inference bolsters privacy and responsiveness [45]. Wearable-paired edge nodes can run symptom triage assistants that interpret patient-reported input without cloud dependency. Immediate fall-detection alerts or medication reminders benefit from Rust's minimal

cold-start latencies. At the same time, Python integrations facilitate interoperability with electronic health record systems for batch data uploads and compliance reporting.

In industrial IoT environments [40], predictive maintenance systems can utilise LLM-based log analysis to diagnose equipment anomalies. Local inference on embedded controllers enables technicians to query machine health in natural language. Employing Rust clients for unscheduled on-demand diagnostics ensures rapid turnaround during downtime events, whereas Python clients handle routine summary generation and integration with enterprise monitoring dashboards.

Autonomous robotics and unmanned aerial vehicles (UAVs) [16] operating in remote settings rely on swift, reliable language understanding for mission-critical commands. Deploying quantised LLMs on board and using Rust to pre-load models during transit minimises decision latency when interpreting dynamic mission updates. Python-based logging modules can later transmit mission logs for centralised analysis once connectivity resumes.

Educational technology in bandwidth-constrained regions also stands to benefit [1, 10]. Low-cost edge devices hosting tutoring bots can deliver immediate feedback on student queries with Rust-powered inference, while Python scripts generate personalised practice exercises and performance analytics. This dual-client approach strikes a balance between the need for instant response and rich, data-driven reporting for educators.

7. Conclusion

This study presents the first head-to-head investigation of Python and Rust clients for quantised large language model inference on a resource-constrained edge device. By running an identical prompt suite across four representative quantised architectures on a Raspberry Pi 4, we separated client-side overhead from core model execution. We demonstrated that Rust markedly reduces initialisation latency while both implementations deliver essentially the same sustained decoding performance once models are loaded. Our comprehensive statistical analyses, spanning both parametric and nonparametric tests, confirm that client-induced differences beyond startup are negligible. These findings suggest that developers should favour Rust when a fast cold-start response is critical. In contrast, Python remains an excellent choice for applications where integration simplicity and ecosystem flexibility outweigh one-time load costs. Moving forward, extending this benchmark to support heterogeneous hardware accelerators, emerging quantisation techniques, adaptive batching and scheduling strategies, and end-to-end task accuracy measurements will further guide the deployment of efficient, reliable on-device LLM services in real-world edge settings.

Author contributions: Conceptualisation, methodology, software, writing – review and editing, Partha P. Ray; supervision, Mohan P. Pradhan. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data availability statement: No new data were created or analysed during this study. Data sharing is not applicable. The code is available at https://github.com/ParthaPRay/python_rust_ollama_analysis.

Conflicts of interest: The authors declare no conflict of interest.

Declaration on generative AI: During the preparation of this work, the authors used o4-mini-high to improve writing style and content enhancement as well as analysis of results. The authors reviewed and edited the content as needed and took full responsibility for the publication's content.

References

[1] Abu-Rasheed, H., Jumbo, C., Al Amin, R., Weber, C., Wiese, V., Obermaisser, R. and Fathi, M., 2025. LLM-Assisted Knowledge Graph Completion for Curriculum and Domain Modelling in Personalized Higher Education Recommendations. *2025 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, pp.1–5. Available from: https://doi.org/10.1109/EDUCON62633.2025.11016377.

[2] Alizadeh, K., Mirzadeh, S.I., Belenko, D., Khatamifard, S., Cho, M., Del Mundo, C.C., Rastegari, M. and Farajtabar, M., 2024. LLM in a flash: Efficient Large Language Model Inference with Limited Memory. *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. pp.12562–12584. Available from: https://aclanthology.org/2024.acl-long.678.pdf.

[3] Alves, V., Bezerra, C., Machado, I., Rocha, L., Virgínio, T. and Silva, P., 2025. Quality Assessment of Python Tests Generated by Large Language Models. 2506.14297, Available from: https://doi.org/10.48550/arXiv.2506.14297.

[4] Banerjee, D., Singh, P., Avadhanam, A. and Srivastava, S., 2023. Benchmarking LLM powered Chatbots: Methods and Metrics. 2308.04624, Available from: https://doi.org/10.48550/arXiv.2308.04624.

[5] Basu, S., Schillinger, D., Patel, S.Y. and Rigdon, J., 2024. Simulating A/B testing versus SMART designs for LLM-driven patient engagement to close preventive care gaps. *npj Digital Medicine*, 7(1), p.322. Available from: https://doi.org/10.1038/s41746-024-01330-2.

[6] Beierlieb, L., Bauer, A., Leppich, R., Iffländer, L. and Kounev, S., 2023. Efficient Data Processing: Assessing the Performance of Different Programming Languages. *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23 Companion. pp.83–87. Available from: https://doi.org/10.1145/3578245.3584691.

[7] Cheng, X., Sang, F., Zhai, Y., Zhang, X. and Kim, T., 2025. RUG: Turbo LLM for Rust Unit Test Generation. *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE Computer Society, p.2983–2995. Available from: https://doi.org/10.1109/ICSE55347.2025.00097.

[8] Chrono: Date and Time for Rust, 2025. Available from: https://docs.rs/chrono.

[9] Chu, B., Feng, Y., Liu, K., Shi, H., Nan, Z., Guo, Z. and Xu, B., 2025. Boosting Rust Unit Test Coverage through Hybrid Program Analysis and Large Language Models. 2506.09002, Available from: https://doi.org/10.48550/arXiv.2506.09002.

[10] Chu, Z., Wang, S., Xie, J., Zhu, T., Yan, Y., Ye, J., Zhong, A., Hu, X., Liang, J., Yu, P.S. and Wen, Q., 2025. LLM Agents for Education: Advances and Applications. 2503.11733, Available from: 10.48550/arXiv.2503.11733.

[11] CSV – CSV File Reading and Writing, 2025. Available from: https://docs.python.org/3/library/csv.html.

[12] Datetime – Basic date and time types, 2025. Available from: https://docs.python.org/3/library/datetime.html.

[13] Deligiannis, P., Lal, A., Mehrotra, N., Poddar, R. and Rastogi, A., 2024. RustAssistant: Using LLMs to Fix Compilation Errors in Rust Code. *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE Press, p.3097–3109. Available from: https://doi.org/10.1109/ICSE55347.2025.00022.

[14] Deligiannis, P., Lal, A., Mehrotra, N. and Rastogi, A., 2023. Fixing Rust Compilation Errors using LLMs. 2308.05177, Available from: https://doi.org/10.48550/arXiv.2308.05177.

[15] Dong, Q., Chen, X. and Satyanarayanan, M., 2024. Creating Edge AI from Cloud-

based LLMs. *Proceedings of the 25th International Workshop on Mobile Computing Systems and Applications*. pp.8–13. Available from: https://doi.org/10.1145/3638550.3641126.

[16] Emami, Y., Zhou, H., Nabavirazani, S. and Almeida, L., 2025. LLM-Enabled In-Context Learning for Data Collection Scheduling in UAV-assisted Sensor Networks. 2504.14556, Available from: https://doi.org/10.48550/arXiv.2504.14556.

[17] Eniser, H.F., Zhang, H., David, C., Wang, M., Christakis, M., Paulsen, B., Dodds, J. and Kroening, D., 2024. Towards translating real-world code with LLMs: A study of translating to Rust. 2405.11514, Available from: https://doi.org/10.48550/arXiv.2405.11514.

[18] Gao, H., Yang, Y., Sun, M., Wu, J., Zhou, Y. and Xu, B., 2025. ClozeMaster: Fuzzing Rust Compiler by Harnessing Llms for Infilling Masked Real Programs. *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE Computer Society, p.1422–1435. Available from: https://doi.org/10.1109/ICSE55347.2025.00175.

[19] Godoy, W.F., Valero-Lara, P., Teranishi, K., Balaprakash, P. and Vetter, J.S., 2024. Large language model evaluation for high-performance computing software development. *Concurrency and Computation: Practice and Experience*, 36(26), p.e8269. Available from: https://doi.org/10.1002/cpe.8269.

[20] Hatalis, K., Christou, D., Myers, J., Jones, S., Lambert, K., Amos-Binks, A., Dannenhauer, Z. and Dannenhauer, D., 2024. Memory Matters: The Need to Improve Long-Term Memory in LLM-Agents. *Proceedings of the AAAI Symposium Series*, vol. 2. pp.277–280. Available from: https://doi.org/10.1609/aaaiss.v2i1.27688.

[21] Hong, J. and Ryu, S., 2025. Type-migrating C-to-Rust translation using a large language model. *Empirical Software Engineering*, 30(1), p.3. Available from: https://doi.org/10.1007/s10664-024-10573-2.

[22] Huey, J.D. and Abdennur, N., 2024. Bigtools: a high-performance BigWig and BigBed library in Rust. *Bioinformatics*, 40(6), p.btae350. Available from: https://doi.org/10.1093/bioinformatics/btae350.

[23] Kao, C.H., Zhao, W., Revankar, S., Speas, S., Bhagat, S., Datta, R., Phoo, C.P., Mall, U., Vondrick, C., Bala, K. and Hariharan, B., 2025. Towards LLM Agents for Earth Observation. 2504.12110, Available from: https://doi.org/10.48550/arXiv.2504.12110.

[24] Lawton, N., Padmakumar, A., Gaspers, J., FitzGerald, J., Kumar, A., Steeg, G.V. and Galstyan, A., 2024. QuAILoRA: Quantization-Aware Initialization for LoRA. 2410.14713, Available from: https://doi.org/10.48550/arXiv.2410.14713.

[25] Li, K. and Yuan, Y., 2024. Large language models as test case generators: Performance evaluation and enhancement. 2404.13340, Available from: https://doi.org/10.48550/arXiv.2404.13340.

[26] Li, Z., Feng, W., Guizani, M. and Yu, H., 2025. TPI-LLM: Serving 70B-scale LLMs Efficiently on Low-resource Mobile Devices. *IEEE Transactions on Services Computing*, 18(05), pp.3321–3333. Available from: https://doi.org/10.1109/TSC.2025.3596892.

[27] Li, Z., Su, Y., Yang, R., Xie, C., Wang, Z., Xie, Z., Wong, N. and Yang, H., 2025. Quantization meets reasoning: Exploring LLM low-bit quantization degradation for mathematical reasoning. 2501.03035, Available from: https://doi.org/10.48550/arXiv.2501.03035.

[28] Liang, L., Gong, J., Liu, M., Wang, C., Ou, G., Wang, Y., Peng, X. and Zheng, Z., 2025. RustEvo^2: An Evolving Benchmark for API Evolution in LLM-based Rust Code Generation. 2503.16922, Available from: https://doi.org/10.48550/arXiv.2503.16922.

[29] Lin, J., Tang, J., Tang, H., Yang, S., Xiao, G. and Han, S., 2025. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. *GetMobile: Mobile Computing and Communications*, 28(4), pp.12–17. Available from: https://doi.org/10.1145/3714983.3714987.

[30] Luo, Y., Zhou, H., Zhang, M., De La Rosa, D., Ahmed, H., Xu, W. and Xu, D., 2025. HALURust: Exploiting Hallucinations of Large Language Models to Detect Vulnerabilities in Rust. 2503.10793, Available from: https://doi.org/10.48550/arXiv.2503.10793.

[31] Martins, E.M., Faé, L.G., Hoffmann, R.B., Bianchessi, L.S. and Griebler, D., 2025. NPB-Rust: NAS Parallel Benchmarks in Rust. 2502.15536, Available from: https://doi.org/10.48550/arXiv.2502.15536.

[32] Nitin, V., Krishna, R., Valle, L.L. and Ray, B., 2025. C2SaferRust: Transforming C Projects into Safer Rust with NeuroSymbolic Techniques. 2501.14257, Available from: https://doi.org/10.48550/arXiv.2501.14257.

[33] Oh, H., Kim, K., Kim, J., Kim, S., Lee, J., Chang, D.S. and Seo, J., 2024. ExeGPT: Constraint-Aware Resource Scheduling for LLM Inference. *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 2. pp.369–384. Available from: https://doi.org/10.1145/3620665.3640383.

[34] Ollama, 2025. Available from: https://ollama.com/.

[35] OS – Miscellaneous operating system interfaces, 2025. Available from: https://docs.python.org/3/library/os.html.

[36] Park, J.J. and Choi, S.J., 2024. LLMs for Enhanced Agricultural Meteorological Recommendations. 2408.04640, Available from: https://doi.org/10.48550/arXiv.2408.04640.

[37] Prabakar, A. and Kiran, R., 2024. *WebAssembly Performance Analysis: A Comparative Study of C++ and Rust Implementations*. Available from: https://www.diva-portal.org/smash/get/diva2:1879948/FULLTEXT01.pdf.

[38] Requests: HTTP for Humans^{TM}, 2025. Available from: https://requests.readthedocs.io/.

[39] Reqwest, 2025. Available from: https://docs.rs/reqwest/.

[40] Sarhaddi, F., Nguyen, N.T., Zuniga, A., Hui, P., Tarkoma, S., Flores, H. and Nurmi, P., 2025. LLMs and IoT: A Comprehensive Survey on Large Language Models and the Internet of Things. *Techrxiv*. Available from: https://www.techrxiv.org/doi/full/10.36227/techrxiv.174063060.01215875.

[41] Serde, 2025. Available from: https://serde.rs/.

[42] Shetty, M., Jain, N., Godbole, A., Seshia, S.A. and Sen, K., 2024. Syzygy: Dual Code-Test C to (safe) Rust Translation using LLMs and Dynamic Analysis. 2412.14234, Available from: https://doi.org/10.48550/arXiv.2412.14234.

[43] Sinha, S., Kalwani, P., Shah, A. and Gonsalves, J., 2025. High-Performance File Searching with Rust: Parallelized Indexing for Enhanced Computational Efficiency. *2025 IEEE International Conference on Interdisciplinary Approaches in Technology and Management for Social Innovation (IATMSI)*, vol. 3. IEEE, pp.1–6. Available from: https://doi.org/10.1109/IATMSI64286.2025.10984781.

[44] Time – Time access and conversions, 2025. Available from: https://docs.python.org/3/library/time.html.

[45] Wang, Z., Li, H., Huang, D., Kim, H.S., Shin, C.W. and Rahmani, A.M., 2025. HealthQ: Unveiling questioning capabilities of LLM chains in healthcare conversations. *Smart Health*, p.100570. Available from: https://doi.org/10.1016/j.smhl.2025.100570.

[46] Wu, J., Chen, S., Cao, J., Lo, H.C. and Cheung, S.C., 2025. Isolating language-coding from problem-solving: Benchmarking llms with pseudoeval. 2502.19149,

Available from: https://doi.org/10.48550/arXiv.2502.19149.

[47] Xia, Y., Fu, F., Zhang, W., Jiang, J. and Cui, B., 2024. Efficient Multi-task LLM Quantization and Serving for Multiple LoRA Adapters. *Advances in Neural Information Processing Systems*, vol. 37. pp.63686–63714. Available from: https://proceedings.neurips.cc/paper_files/paper/2024/hash/747dc7c6566c74eb9a663bcd8d057c78-Abstract-Conference.html.

[48] Xu, D., Yin, W., Zhang, H., Jin, X., Zhang, Y., Wei, S., Xu, M. and Liu, X., 2025. EdgeLLM: Fast On-device LLM Inference with Speculative Decoding. *IEEE Transactions on Mobile Computing*, p.3256–3273. Available from: https://doi.org/10.1109/TMC.2024.3513457.

[49] Yang, A.Z., Takashima, Y., Paulsen, B., Dodds, J. and Kroening, D., 2024. VERT: Verified Equivalent Rust Transpilation with Large Language Models as Few-Shot Learners. 2404.18852, Available from: https://doi.org/10.48550/arXiv.2404.18852.

[50] Yao, J., Zhou, Z., Chen, W. and Cui, W., 2023. Leveraging Large Language Models for Automated Proof Synthesis in Rust. 2311.03739, Available from: https://doi.org/10.48550/arXiv.2311.03739.

[51] Yu, Z., Wang, Z., Li, Y., Gao, R., Zhou, X., Bommu, S.R., Zhao, Y. and Lin, Y., 2024. EDGE-LLM: Enabling Efficient Large Language Model Adaptation on Edge Devices via Unified Compression and Adaptive Layer Voting. *Proceedings of the 61st ACM/IEEE Design Automation Conference*. p.327. Available from: https://doi.org/10.1145/3649329.3658473.

[52] Zeng, C., Liu, S., Xie, Y., Liu, H., Wang, X., Wei, M., Yang, S., Chen, F. and Mei, X., 2025. ABQ-LLM: arbitrary-bit quantized inference acceleration for large language models. *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*. p.2487. Available from: https://doi.org/10.1609/aaai.v39i21.34385.

[53] Zhang, H., David, C., Wang, M., Paulsen, B. and Kroening, D., 2025. Scalable, Validated Code Translation of Entire Projects using Large Language Models. *Proceedings of the ACM on Programming Languages*, 9(PLDI), p.212. Available from: https://doi.org/10.1145/3729315.

[54] Zhang, X., Liu, J., Xiong, Z., Huang, Y., Xie, G. and Zhang, R., 2024. Edge intelligence optimization for large language model inference with batching and quantization. *2024 IEEE Wireless Communications and Networking Conference (WCNC) - Proceedings*. IEEE. Available from: https://doi.org/10.1109/WCNC57260.2024.10571127.

[55] Zhang, Y., Liu, Z., Feng, Y. and Xu, B., 2024. Leveraging Large Language Model to Assist Detecting Rust Code Comment Inconsistency. *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. pp.356–366. Available from: https://doi.org/10.1145/3691620.3695010.

[56] Zhao, W., Jing, W., Lu, Z. and Wen, X., 2024. Edge and Terminal Cooperation Enabled LLM Deployment Optimization in Wireless Network. *2024 IEEE/CIC International Conference on Communications in China (ICCC Workshops)*. IEEE, pp.220–225. Available from: https://doi.org/10.1109/ICCCWorkshops62562.2024.10693742.

[57] Zheng, Z., Ren, X., Xue, F., Luo, Y., Jiang, X. and You, Y., 2023. Response Length Perception and Sequence Scheduling: An LLM-Empowered LLM Inference Pipeline. *Advances in Neural Information Processing Systems*, vol. 36. pp.65517–65530. Available from: https://proceedings.neurips.cc/paper_files/paper/2023/hash/ce7ff3405c782f761fac7f849b41ae9a-Abstract-Conference.html.