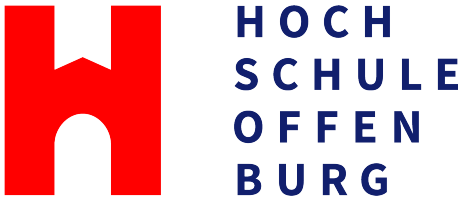

Effiziente lokale LLM-basierte Chatbots auf CPU-basierten Embedded Systems

Nico Jörger

BACHELORARBEIT

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

Studiengang Angewandte Informatik

Fakultät Elektrotechnik, Medizintechnik und Informatik
Hochschule für Technik, Wirtschaft und Medien Offenburg

29.04.2026

Betreuer
Prof. Dr. Christian Reich-Haag, Hochschule Offenburg
Prof. Dr. Axel Sikora, Hochschule Offenburg

Jörger, Nico:
Effiziente lokale LLM-basierte Chatbots auf CPU-basierten Embedded Systems / Nico Jörger. –
BACHELORARBEIT, Offenburg : Hochschule für Technik, Wirtschaft und Medien Offenburg, 2026.
57 Seiten.

Jörger, Nico:
Efficient Local LLM-based Chatbots on CPU-based Embedded Systems / Nico Jörger. –
BACHELOR THESIS, Offenburg : Offenburg University, 2026. 57 pages.

Vorwort

Diese Arbeit entstand aus dem Interesse an der praktischen Frage, unter welchen technischen Randbedingungen sich große Sprachmodelle auch auf ressourcenbeschränkter Hardware lokal betreiben lassen. Mein Dank gilt Prof. Dr. Christian Reich-Haag für die direkte Betreuung und die hilfreichen fachlichen Rückmeldungen während der Bearbeitung. Prof. Dr. Axel Sikora danke ich für die Übernahme der Zweitbegutachtung. Ebenso danke ich allen, die mich beim Aufbau und Testen der Hard- und Softwareumgebung unterstützt haben.

Eidesstattliche Erklärung

Hiermit versichere ich eidesstattlich, dass die vorliegende Bachelorarbeit von mir selbstständig und ohne unerlaubte fremde Hilfe angefertigt worden ist, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen, unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

Ich erkläre hiermit, dass ich beim Einsatz von IT-/KI-gestützten Schreibwerkzeugen diese Werkzeuge in der Rubrik „Übersicht verwendeter Hilfsmittel"mit ihrem Produktnamen, meiner Bezugsquelle und einer Übersicht des im Rahmen dieser Studienarbeit genutzten Funktionsumfangs vollständig aufgeführt habe. Davon ausgenommen sind diejenigen IT-/KI-gestützten Schreibwerkzeuge, die vom zuständigen Prüfer / von der zuständigen Prüferin zum Zeitpunkt der Abgabe meiner Studienleistung als nicht anzeigepflichtig eingestuft wurden („Whitelist"). Bei der Erstellung der vorgelegten Studienleistung habe ich durchgehend eigenständig und beim Einsatz IT-/KI-gestützter Schreibwerkzeuge steuernd gearbeitet.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Offenburg öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Achern, 29.04.2026

Nico Jörger

Übersicht verwendeter Hilfsmittel

Produktname	**Bezugsquelle**	**Genutzter Funktionsumfang**
ChatGPT (OpenAI)	chat.openai.com	Informationsbeschaffung, Programmierhilfe, Übersetzung, Textkorrektur und -formulierung, Feedback
Claude (Anthropic)	claude.ai	Informationsbeschaffung, Programmierhilfe, Übersetzung, Textkorrektur und -formulierung, Feedback
Meta LLaMA-2 & -3	huggingface.co	Informationsbeschaffung, Programmierhilfe, Übersetzung, Textkorrektur und -formulierung, Feedback
Passiv eingesetzt (ggf. integrierte KI-Funktionen, nicht aktiv genutzt):		
Google Suche	google.com	Informationsbeschaffung (ggf. KI-generierte Zusammenfassungen)
Microsoft Office	microsoft.com	Textverarbeitung, Tabellenkalkulation
Microsoft Paint	microsoft.com	Bildbearbeitung
Visual Studio Code	code.visualstudio.com	Entwicklungsumgebung

Zusammenfassung

Effiziente lokale LLM-basierte Chatbots auf CPU-basierten Embedded Systems

Diese Arbeit untersucht die effiziente lokale Ausführung von Large Language Model (LLM)-basierten Chatbots auf CPU-basierten Embedded Systems am Beispiel des Raspberry Pi 5. Verglichen werden TinyChatEngine 1 (MIT Han Lab, AWQ-Quantisierung) und llama.cpp hinsichtlich Inferenzgeschwindigkeit, Time to First Token, Speicherauslastung, Leistungsaufnahme und Modellgüte. Da der Raspberry Pi 5 über keine integrierte Schnittstelle zur Leistungsmessung verfügt, wird eine externe Messinfrastruktur auf Basis eines ESP32-Mikrocontrollers mit zwei INA219-Stromsensoren sowie InfluxDB und Grafana entwickelt. Die Modellgüte wird anhand der Benchmarks MMLU, MATH500 und TinyBenchmarks bewertet. Ergänzend erfasst LocalScore auf Basis von llamafile die Hardware-nahe Inferenzleistung. Die Ergebnisse zeigen, dass llama.cpp (Q4_K_M) mit 2,1–2,3 tok/s auf dem Raspberry Pi 5 konsistent schneller ist als TinyChatEngine (0,5–1,6 tok/s) und auch bei der Modellgüte höhere Benchmark-Werte erzielt. Den für interaktiven Dialogbetrieb angesetzten Schwellenwert von 3 tok/s erreichen 8B-Modelle nicht, Modelle der 1B- und 3–5B-Klasse hingegen schon. Für den praktischen Einsatz empfiehlt sich llama.cpp mit Q4_K_M-Quantisierung auf dem 16-GB-Modell.

Abstract

Efficient Local LLM-based Chatbots on CPU-based Embedded Systems

This thesis investigates the efficient local execution of Large Language Model (LLM)-based chatbots on CPU-based embedded systems, using the Raspberry Pi 5 as a representative platform. TinyChatEngine 1 (MIT Han Lab, AWQ quantization) and llama.cpp are compared with respect to inference speed, time to first token, memory usage, power consumption, and model quality. Since the Raspberry Pi 5 does not provide a built-in interface for power measurement, an external measurement infrastructure based on an ESP32 microcontroller with two INA219 current sensors, InfluxDB, and Grafana is developed. Model quality is assessed using the MMLU, MATH500, and TinyBenchmarks benchmarks. LocalScore based on llamafile is additionally used to measure hardware-oriented inference performance. The results show that llama.cpp (Q4_K_M) consistently outperforms TinyChatEngine on the Raspberry Pi 5, achieving 2.1–2.3 tok/s versus 0.5–1.6 tok/s, and also scores higher on model quality benchmarks. 8B models fall below the 3 tok/s threshold set for interactive dialogue, while models in the 1B and 3–5B class exceed it. For practical deployment, llama.cpp with Q4_K_M quantization on the 16 GB variant is recommended.

Inhaltsverzeichnis

1 Einleitung

Die Verfügbarkeit leistungsfähiger Sprachmodelle ist heute nahezu vollständig an Cloud-Infrastruktur geknüpft. Systeme wie ChatGPT, Claude, Grok oder Microsoft Copilot werden auf GPU[1]-Clustern mit mehreren hundert Teraflops Rechenleistung betrieben, eine Voraussetzung, die den lokalen Einsatz auf ressourcenbeschränkter Hardware bislang praktisch ausschließt. Gleichzeitig wächst das Interesse an dezentraler, netzwerkunabhängiger Inferenz: Datenschutzanforderungen, Offline-Szenarien und der Wunsch nach kontrollierbarer, kostengünstiger Infrastruktur schaffen einen konkreten Bedarf, dem die aktuelle Forschung und Entwicklung zunehmend begegnet.

Neuere Quantisierungsverfahren, darunter AWQ[2] [Li24], ermöglichen es, den Speicherbedarf von Modellen mit mehreren Milliarden Parametern auf wenige Gigabyte zu reduzieren, ohne die Ausgabequalität wesentlich zu verschlechtern. In Kombination mit auf Effizienz ausgelegten Inferenz-Engines eröffnet dies die Frage, ob ein praxistauglicher LLM[3]-basierter Chatbot auf CPU[4]-basierter Embedded-Hardware betrieben werden kann. Diese Frage soll in der vorliegenden Arbeit anhand des Raspberry Pi 5 untersucht werden.

1.1 Motivation und Problemstellung

Eingebettete Systeme auf CPU-Basis sind in vielen Anwendungskontexten präsent, etwa in industrieller Steuerung, Bildung, Prototypenentwicklung und Edge-Computing-Szenarien. Der Raspberry Pi 5 wurde als Untersuchungsplattform gewählt, da er als weit verbreitetes, gut dokumentiertes Einplatinencomputersystem mit einem ARM Cortex-A76-Prozessor und 8–16 GB LPDDR4X-RAM[5] den oberen Rand CPU-basierter Embedded-Hardware repräsentiert. Seine ARM-Architektur ist dabei besonders relevant, da TinyChatEngine 1 explizit optimierte Kernel für ARM-SIMD[6]-Einheiten (NEON) bereitstellt und der Raspberry Pi 5 damit als aussagekräftiger Stellvertreter für diese Geräteklasse dient.

[1] Graphics Processing Unit
[2] Activation-aware Weight Quantization
[3] Large Language Model
[4] Central Processing Unit
[5] Random Access Memory
[6] Single Instruction Multiple Data

Der Betrieb eines 8B-AWQ-Modells auf dem Raspberry Pi 5 mit 8GB RAM ist mit erheblichen Hürden verbunden: Speicherengpässe führen zu instabilem Verhalten. Die Inferenzgeschwindigkeit liegt ohne gezielte Optimierungen weit unterhalb dessen, was für eine flüssige Konversation erforderlich wäre. Offen ist dabei, wie sich verschiedene Inferenz-Engines und Quantisierungsgrade auf Geschwindigkeit, Speicherverbrauch und Ausgabequalität auswirken, und ob durch gezielte Optimierungen eine praxistaugliche Konfiguration erreichbar ist.

Hinzu kommt ein messtechnisches Problem: Der Raspberry Pi 5 verfügt über keine integrierte Schnittstelle zur Erfassung der Leistungsaufnahme. Für eine vollständige Effizienzbetrachtung, die über reine Geschwindigkeitsmessungen hinausgeht, muss diese Lücke durch externe Messtechnik geschlossen werden. Die zeitliche Synchronisation der Leistungsmessung mit den Inferenzläufen stellt dabei einen eigenständigen technischen Beitrag dieser Arbeit dar.

1.2 Zielsetzung

Die Arbeit verfolgt das Ziel, die lokale LLM-Inferenz auf dem Raspberry Pi 5 unter realen Bedingungen zu untersuchen und systematisch zu bewerten. Im Mittelpunkt steht dabei TinyChatEngine 1 des MIT Han Lab als AWQ-Fallstudie für ressourcenbeschränkte Systeme ohne dedizierte GPU. llama.cpp dient als etabliertes Referenz-Framework mit breiter Modell- und Quantisierungsunterstützung. Damit steht nicht nur die Frage im Vordergrund, welches Backend die höchste Rohleistung erzielt, sondern auch, welchen praktischen Nutzen AWQ auf einer realen ARM-Zielplattform unter den tatsächlichen Randbedingungen bringt.

Als Testmodelle kommen Vertreter verschiedener Modellklassen und -größen zum Einsatz, u. a. LLaMA-3, Gemma-3 und Qwen 2.5 in verschiedenen Quantisierungsstufen. Die Evaluation umfasst Inferenzgeschwindigkeit (Tokens/s), TTFT[7], Peak-Speicherauslastung, Leistungsaufnahme sowie Modellgüte anhand der Benchmarks MMLU[8], MATH500 und TinyBenchmarks. Hardware-Performance wird zusätzlich mit LocalScore erfasst.

Zur lückenlosen Erfassung dieser Metriken wird eine Messinfrastruktur auf Basis von InfluxDB und Grafana aufgebaut, die Laufzeitkennzahlen kontinuierlich erfasst und visualisiert. Für die Leistungsmessung wird eine externe Lösung auf Basis eines ESP32-Mikrocontrollers mit zwei INA219-Stromsensoren entwickelt, die eine zeitlich aufgelöste Korrelation von Leistungs- und Inferenzdaten erlaubt.

Daraus ergeben sich die folgenden Forschungsfragen[9], die diese Arbeit beantwortet:

[7] Time to First Token
[8] Massive Multitask Language Understanding
[9] RQ = *Research Question* (Forschungsfrage)

RQ1 Erreicht lokale LLM-Inferenz auf dem Raspberry Pi 5 eine für interaktive Nutzung ausreichende Token-Rate (≥ 3 Tokens/s) bei akzeptabler Modellgüte?

RQ2 Welche Unterschiede zeigen TinyChatEngine 1 (AWQ) und llama.cpp (GGUF[10]) hinsichtlich Inferenzgeschwindigkeit, Energieverbrauch pro Token und Modellgüte auf identischer Hardware?

RQ3 Welche Modell- und Quantisierungskonfiguration ist unter diesen Randbedingungen für den praktischen Einsatz auf dem Raspberry Pi 5 aktuell am besten geeignet?

1.3 Abgrenzung

Diese Arbeit beschränkt sich auf CPU-basierte Inferenz ohne dedizierte GPU oder neuronale Prozessoreinheit (NPU[11]). NPU-fähige Systeme sowie GPU-beschleunigte Embedded-Plattformen (z. B. NVIDIA Jetson) werden nicht untersucht. Eine parallele Arbeit von Lukas Heiming (Hochschule Offenburg, 2026) widmet sich der LLM-Inferenz auf GPU-basierten Systemen am Beispiel des NVIDIA Jetson Nano [He26]. Die Evaluation beschränkt sich auf den Raspberry Pi 5 als repräsentative Plattform. Eine Generalisierung auf andere CPU-basierte Embedded-Systeme ist möglich, erfordert jedoch eine separate Evaluation. Ebenfalls nicht Gegenstand dieser Arbeit sind Fine-Tuning, Retrieval-Augmented Generation und multimodale Modelle.

1.4 Vorgehen und Aufbau der Arbeit

Die Arbeit beginnt mit der Erarbeitung der notwendigen Grundlagen: der Architektur und den Einschränkungen CPU-basierter LLM-Inferenz sowie den eingesetzten Quantisierungsverfahren von TinyChatEngine 1 und llama.cpp. Parallel dazu erfolgt die Inbetriebnahme des Raspberry Pi 5 und die Einarbeitung in die verwendeten Werkzeuge.

Im praktischen Teil wird zunächst die Inferenzumgebung mit beiden Backends aufgebaut. Da TinyChatEngine 1 in der Standardversion über keine automatisierbare Schnittstelle verfügt, wird eine REST[12]-API[13] für standardisierte Inferenzanfragen entwickelt. Ein einheitlicher FastAPI-Proxy (LLMProxy) abstrahiert anschließend alle Backends hinter einer gemeinsamen OpenAI-kompatiblen Schnittstelle und erfasst Inferenzmetriken pro Request. Die Messinfrastruktur wird so integriert, dass alle relevanten Metriken automatisiert erfasst, in InfluxDB persistiert und über Grafana visualisiert werden.

[10] GPT-Generated Unified Format
[11] Neural Processing Unit
[12] Representational State Transfer
[13] Application Programming Interface

Die eigentliche Evaluation erfolgt über standardisierte Benchmark-Läufe (MMLU, MATH500, TinyBenchmarks, LocalScore), bei denen alle Backends unter identischen Bedingungen verglichen werden. Die Ergebnisse werden abschließend zusammengeführt, in Bezug auf die praktische Einsatzfähigkeit bewertet und im Kontext bestehender Arbeiten betrachtet.

Die Arbeit ist in fünf Kapitel gegliedert. Kapitel 2 vermittelt die notwendigen Grundlagen und gibt einen Überblick über den Stand der Technik im Bereich cloud-basierter und lokaler LLM-Inferenz, Quantisierungsverfahren und der eingesetzten Frameworks. Anschließend werden die spezifischen Herausforderungen der CPU-basierten Inferenz auf eingebetteten Systemen analysiert.

Kapitel 3 beschreibt die aufgebaute Mess- und Inferenzumgebung im Detail: den Hardware-Aufbau, die entwickelten Erweiterungen für TinyChatEngine 1, den LLMProxy, die externe Leistungsmessinfrastruktur sowie den eingesetzten Monitoring-Stack.

Kapitel 4 präsentiert die Ergebnisse der durchgeführten Benchmarks und Leistungsmessungen für alle getesteten Backends und Modellkonfigurationen.

Abschließend werden die Ergebnisse in Kapitel 5 diskutiert, in den Kontext bestehender Arbeiten eingeordnet und ein Ausblick auf weiterführende Fragestellungen gegeben.

2 Grundlagen und Stand der Technik

Dieses Kapitel legt die technischen Grundlagen für die in dieser Arbeit durchgeführten Untersuchungen dar. Es beginnt mit den Grundlagen großer Sprachmodelle: Transformer-Architektur, autoregressive Inferenz und KV[1]-Cache, und beschreibt anschließend den aktuellen Stand der Technik im Bereich LLM-Inferenz, von cloud-basierten Diensten über lokale Inferenz-Frameworks bis hin zu TinyChatEngine 1 als primärer Untersuchungsplattform. Abschließend werden die spezifischen Herausforderungen analysiert, die sich aus dem Einsatz ressourcenbeschränkter CPU-basierter Hardware ergeben.

2.1 Grundlagen großer Sprachmodelle

Dieser Abschnitt legt die konzeptionellen Grundlagen für die in dieser Arbeit untersuchten LLMs und ihre Inferenzeigenschaften dar. Die hier eingeführten Konzepte, insbesondere KV-Cache und GQA[2], bilden die Basis für die in Kapitel 3 beschriebenen Korrekturen und Optimierungen.

2.1.1 Transformer-Architektur

Moderne LLMs basieren auf der Transformer-Architektur [Va17], insbesondere auf dem *Decoder-only*-Aufbau, wie er in der GPT- und LLaMA-Modellreihe verwendet wird. Die Eingabe wird zunächst durch eine Embedding-Schicht in einen hochdimensionalen Vektorraum überführt. Anschließend durchläuft sie eine Folge von N identisch aufgebauten Decoder-Schichten.

Jede Decoder-Schicht besteht aus zwei Teilkomponenten:

- **Self-Attention**: berechnet für jede Token-Position gewichtete Abhängigkeiten zu allen vorherigen Positionen im Kontext.

[1] Key-Value
[2] Grouped Query Attention

- **FFN**[3]: eine positionsweise angewandte Vorwärts-Schicht, die typischerweise aus zwei linearen Transformationen mit einer nichtlinearen Aktivierungsfunktion besteht.

Am Ende der Schichtenfolge bildet ein linearer *LM-Head* den Ausgabevektor auf das gesamte Vokabular ab. Die resultierende Wahrscheinlichkeitsverteilung über alle möglichen Tokens bestimmt die Modellausgabe. LLaMA-3-8B [Ll24], das in dieser Arbeit als primäres Untersuchungsmodell dient, verwendet 32 solcher Schichten bei einer Einbettungsdimension von 4096.

2.1.2 Autoregressive Inferenz und KV-Cache

LLMs generieren Ausgaben *autoregressiv*: In jedem Schritt wird genau ein Token erzeugt, das dem bisherigen Kontext angehängt und als Eingabe für den nächsten Schritt verwendet. Dieser Prozess läuft so lange, bis ein End-of-Sequence-Token generiert oder eine maximale Länge erreicht wird.

Die Implikation für die Systemleistung ist wesentlich: Bei jedem Generierungsschritt müssen sämtliche Modellgewichte aller N Schichten aus dem Arbeitsspeicher geladen werden, obwohl pro Schritt nur ein einziger neuer Token verarbeitet wird. Dies ist der strukturelle Grund für die in Abschnitt 2.7.1 diskutierten Nachteile des Raspberry Pi 5 gegenüber GPU-Systemen.

Ohne weitere Optimierung müssten beim Verarbeiten des t-ten Tokens die Key- und Value-Vektoren aller vorherigen Positionen $1, \ldots, t-1$ neu berechnet werden. Der **KV-Cache** vermeidet diese Redundanz: Die Key- und Value-Vektoren jeder Schicht werden nach ihrer ersten Berechnung in einem Puffer gespeichert und bei nachfolgenden Schritten direkt ausgelesen. Der Speicheraufwand wächst damit linear mit der Kontextlänge, die Rechenarbeit pro Schritt bleibt jedoch auf den aktuellen Token beschränkt.

Der Inferenzprozess gliedert sich in zwei Phasen:

- **Prefill**[4]: Der gesamte Eingabe-Prompt wird parallel verarbeitet und der KV-Cache initial befüllt. Die Zeit bis zur Ausgabe des ersten Tokens wird als TTFT gemessen.
- **Decode**: Jeder nachfolgende Token wird schrittweise generiert. Der Cache wird pro Schritt um einen Eintrag erweitert. Der Durchsatz dieser Phase wird in Tokens/s gemessen.

Für automatisierte Benchmark-Läufe ist es zwingend erforderlich, den KV-Cache zwischen einzelnen Anfragen zurückzusetzen, um Kontext-Übertragungen zwischen Messungen zu

[3] Feed-Forward Network
[4] Prompt-Verarbeitungsphase (erstes Token)

vermeiden. Die Implementierung des Cache-Resets in TinyChatEngine ist in Abschnitt 3.6.4 beschrieben.

2.1.3 Attention-Varianten und Grouped Query Attention

Der Kern der Self-Attention ist die skalierte Dot-Product-Attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V \tag{2.1}$$

wobei Q, K, V die Query-, Key- und Value-Matrizen sind und d_k die Dimension der Key-Vektoren bezeichnet.

Bei **MHA**[5] verfügt jeder der H Attention-Heads über eigene Query-, Key- und Value-Projektionen. Der KV-Cache muss demnach für jeden Head separate Einträge speichern. Die Cache-Größe skaliert mit $H \cdot d_k \cdot L$, wobei L die Kontextlänge bezeichnet.

MQA[6] reduziert diesen Aufwand, indem alle Query-Heads dieselbe einzige Key- und Value-Projektion teilen. Dies verringert die Cache-Größe erheblich, kann jedoch die Modellqualität beeinträchtigen.

GQA [Ai23] stellt einen Kompromiss dar: Die H Query-Heads werden in G Gruppen aufgeteilt. Alle Heads innerhalb einer Gruppe teilen sich eine gemeinsame Key- und Value-Projektion. MHA entspricht dem Spezialfall $G = H$, MQA dem Fall $G = 1$. LLaMA-3-8B verwendet 32 Query-Heads und 8 KV-Heads ($G = 8$), was den KV-Cache gegenüber vollständigem MHA auf ein Viertel reduziert.

Für die Implementierung des KV-Caches hat GQA eine direkte Konsequenz: Die Puffergröße muss auf Basis der tatsächlichen KV-Kopfanzahl G und der KV-Dimension $kv\_dim = G \cdot d_k$ dimensioniert werden, nicht auf Basis der vollen Einbettungsdimension $embed\_dim = H \cdot d_k$.

2.2 Cloud-basierte LLM-Dienste

Der aktuell dominierende Ansatz für den Betrieb großer Sprachmodelle ist die zentralisierte Cloud-Inferenz. Anbieter wie OpenAI (ChatGPT), xAI (Grok) und Microsoft (Copilot) betreiben ihre Modelle auf dedizierten Rechenclustern, die meist auf NVIDIA-GPUs basieren. Die Kommunikation erfolgt über HTTP-APIs. Der Nutzer erhält Zugang zur Modellausgabe, ohne direkten Einblick in die zugrundeliegende Infrastruktur zu haben.

[5] Multi-Head Attention
[6] Multi-Query Attention

Dieses Betriebsmodell erlaubt den Einsatz sehr großer Modelle, kurze Antwortlatenzen und nahezu unbegrenzte Skalierbarkeit. Es setzt jedoch eine dauerhafte Netzwerkverbindung voraus, überträgt Nutzerdaten an externe Infrastruktur und verursacht laufende Kosten in Abhängigkeit vom Nutzungsvolumen. Für Szenarien mit erhöhten Datenschutzanforderungen oder eingeschränkter Konnektivität ist dieses Modell daher nicht ohne Weiteres anwendbar.

2.3 Lokale LLM-Inferenz

Als Reaktion auf die genannten Einschränkungen hat sich ein wachsendes Ökosystem für lokale LLM-Inferenz entwickelt. Frameworks wie llama.cpp ermöglichen den Betrieb quantisierter Modelle auf Consumer-Hardware ohne GPU-Abhängigkeit.

2.3.1 llama.cpp

llama.cpp [Gl25] ist ursprünglich als reine C++-Portierung des LLaMA-Modells entstanden und hat sich zu einem breit unterstützten Inferenz-Framework entwickelt, das eine Vielzahl von Modellarchitekturen und Quantisierungsformaten unterstützt. Es gilt als De-facto-Referenzimplementierung für CPU-basierte LLM-Inferenz und stellt über `llama-server` eine OpenAI-kompatible API bereit.

llamafile und LocalScore Eine für diese Arbeit relevante Variante des llama.cpp-Ökosystems ist `llamafile` [Mo26]. Dabei wird llama.cpp zusammen mit Modell- und Laufzeitkomponenten in eine portable, ausführbare Datei verpackt, sodass quantisierte GGUF-Modelle ohne separate Installation eines vollständigen Inferenz-Stacks ausgeführt werden können. In dieser Arbeit wird `llamafile` nicht als primäres Backend der eigenen Messinfrastruktur verwendet, sondern im Rahmen von LocalScore.

LocalScore ist ein quelloffenes Benchmarking-Werkzeug, das auf `llamafile` aufsetzt und die Hardware-Performance lokaler LLM-Inferenz anhand standardisierter Prompt- und Generierungsszenarien misst [Lo26]. Im Unterschied zu den übrigen Benchmarks dieser Arbeit bewertet LocalScore nicht die inhaltliche Modellgüte, sondern ausschließlich Laufzeitmetriken wie Prompt-Verarbeitung, Token-Generierung und TTFT. Dadurch dient LocalScore als ergänzende, portable Vergleichsbasis für die Leistungsfähigkeit des Raspberry Pi 5 gegenüber anderen Hardwareplattformen.

2.3.2 Ollama

Ollama ist ein auf llama.cpp aufbauendes Werkzeug, das den Betrieb von Sprachmodellen vereinfacht. Modelle lassen sich per Pull-Befehl herunterladen, Quantisierungsstufen sind frei wählbar. In dieser Arbeit wurde Ollama für erste Funktionstests sowie als Bezugsquelle für Modelle eingesetzt, die für den Vergleich mit der parallelen Arbeit von Lukas Heiming [He26] relevant sind.

2.3.3 Quantisierungsverfahren

Quantisierung reduziert die numerische Präzision der Modellgewichte, typischerweise von FP16[7] auf INT8 oder INT4, und verringert damit sowohl den Speicherbedarf als auch den Rechenaufwand. Das von llama.cpp verwendete GGUF-Format unterstützt mehrere Quantisierungsstufen, die in Tabelle 2.1 zusammengefasst sind.

Ein Vergleich zwischen AWQ- und GGUF-basierten Systemen ist dabei nicht als reiner Vergleich von Bitbreiten zu verstehen. In der Praxis gehen Quantisierung, Inferenz-Backend, Modellkonvertierung und Prompt-Pipeline ineinander über. Unterschiede in den späteren Messergebnissen können daher nicht ohne Weiteres ausschließlich auf das Gewichtsformat zurückgeführt werden.

Stufe	Bits	Charakteristik
Q2_K	≈2	Minimaler Speicherbedarf, deutlich spürbarer Qualitätsverlust
Q3_K_S / _M / _L	≈3	3-Bit-Varianten. _S kleiner, _L besser, für ressourcenbeschränkte Systeme
Q4_0 / Q4_1	≈4	Einfache 4-Bit-Quantisierung ohne Kanalgruppierung
Q4_K_S / Q4_K_M	≈4	K-Quant. _M: gute Balance aus Qualität und Modellgröße, weit verbreitet
Q5_0 / Q5_1	≈5	Einfache 5-Bit-Quantisierung, höhere Qualität als Q4
Q5_K_S / Q5_K_M	≈5	K-Quant, bessere Qualitätserhaltung bei moderatem Mehrspeicherbedarf
Q6_K	≈6	Sehr geringe Qualitätseinbuße, nahe an Q8 bei kleinerem Footprint
Q8_0	8	Sehr hohe Qualität, größerer Speicherbedarf
fp16 / bf16	16	Nahezu verlustfrei, hoher RAM-Bedarf

Tabelle 2.1: Übersicht der GGUF-Quantisierungsstufen (Quelle: Ollama Model Library)

[7] 16-Bit-Gleitkommazahl (Half Precision)

2.3.4 AWQ: Activation-aware Weight Quantization

Unter den aktuellen Quantisierungsverfahren gilt AWQ als besonders verlustarm. Anders als naive Rundungsverfahren berücksichtigt AWQ die Aktivierungsverteilung beim Quantisierungsprozess und priorisiert Gewichte, die einen überproportionalen Einfluss auf die Ausgabe haben. Dies resultiert in einer besseren Qualitätserhaltung bei 4-Bit-Quantisierung im Vergleich zu Verfahren wie GPTQ [Fr23]. SmoothQuant [Xi23] ist ebenfalls Teil von TinyChatEngine, wird jedoch für LLaMA-2 und LLaMA-3 standardmäßig nicht unterstützt und kam daher in dieser Arbeit nicht zum Einsatz.

2.4 TinyChatEngine

TinyChatEngine ist eine vom MIT Han Lab entwickelte Inferenz-Engine, die explizit für den Einsatz auf Edge-Geräten und ressourcenbeschränkten Systemen ohne dedizierte GPU konzipiert wurde. Im Unterschied zu llama.cpp, das auf breite Kompatibilität ausgelegt ist, verfolgt TinyChatEngine einen stärker fokussierten Ansatz: Die Engine ist eng mit den Optimierungstechniken des Han Lab verzahnt, insbesondere mit AWQ und speicheroptimierten Kernel-Implementierungen für ARM- und x86-Architekturen.

Version 1 der Engine wurde für diese Arbeit gewählt, da ausschließlich sie den CPU-Inferenzpfad für ARM-Architekturen implementiert. Version 2 des Projekts fokussiert sich auf CUDA[8]-fähige GPUs und bietet keinen CPU-Pfad für ARM. Sie scheidet für den Einsatz auf dem Raspberry Pi 5 damit aus. Die interne Struktur ist in C++ implementiert, ohne externe Laufzeitabhängigkeiten, und enthält optimierte Kernelmodule für neuronale Netzwerkoperationen (`nn_modules/`, `ops/`, `kernels/`). Im Gegensatz zu llama.cpp ist TinyChatEngine jedoch nur begrenzt generalisiert: Modellpfade, Layout-Annahmen und Teile der Prompt- und Laufzeitlogik sind eng an wenige unterstützte Modellfamilien gekoppelt. Dies macht die Engine als Untersuchungsobjekt für AWQ interessant, erschwert aber die Übertragung auf neuere Modellgenerationen und reduziert die praktische Flexibilität im Vergleich zu llama.cpp. Abbildung 2.1 zeigt den allgemeinen Workflow von TinyChatEngine. Ausgehend von PyTorch-Modellen werden die Gewichte über SmoothQuant/AWQ quantisiert und anschließend über einen Modellkonverter in backendspezifische Binärartefakte überführt. Alternativ lassen sich vorquantisierte Modelle direkt aus dem AWQ Quantized Model Zoo beziehen. Die Zielartefakte werden durch backendspezifische Compiler-Pfade (ARM, x86, CUDA) in ein geräteoptimiertes Binärformat übersetzt, das als TinyChatEngine auf dem Zielgerät läuft.

TinyChatEngine unterstützt unter anderem LLaMA-3 und bildet die primäre Untersuchungsplattform dieser Arbeit. llama.cpp dient als Referenz-Backend.

[8] Compute Unified Device Architecture

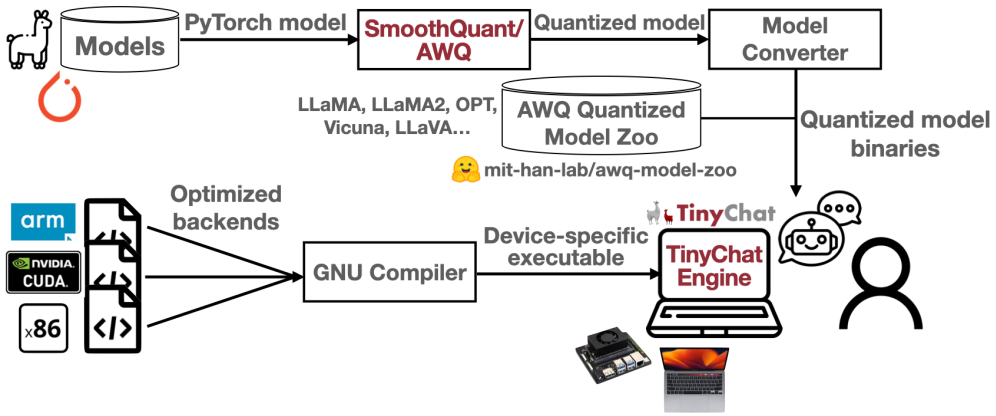

Abbildung 2.1: Allgemeiner Workflow von TinyChatEngine: von PyTorch-Modellen über AWQ-Quantisierung und geräteoptimierte Compiler-Pfade zur Ausführung auf dem Zielgerät (Quelle: [MI24])

2.5 Untersuchte Modelle

Die Evaluation dieser Arbeit konzentriert sich auf zwei primäre Untersuchungsmodelle, LLaMA-2-7B und LLaMA-3-8B, die sowohl mit TinyChatEngine als auch mit llama.cpp betrieben werden. Ergänzend werden eine Reihe weiterer Modelle ausschließlich über llama.cpp als Vergleichspunkt herangezogen. Die konkret eingesetzten Quantisierungsstufen sind im Kontext der jeweiligen Benchmarks in Abschnitt 3.9 aufgeführt.

LLaMA-2-7B [To23] (Meta AI) ist das ältere der beiden primären Untersuchungsmodelle und dient insbesondere als Grundlage für die AWQ-Quantisierungspipeline mit TinyChatEngine. Es verwendet 32 Decoder-Schichten bei einer Einbettungsdimension von 4096 und vollständige MHA mit 32 Query- und 32 KV-Heads. In dieser Arbeit wird die Instruct-Variante (LLaMA-2-7B-Chat) eingesetzt.

LLaMA-3-8B [Ll24] (Meta AI) dient als zweites primäres Untersuchungsmodell und ist bereits in Abschnitt 2.1.1 beschrieben. In dieser Arbeit wird die Instruct-Variante (LLaMA-3-8B-Instruct) eingesetzt. Zusammen mit LLaMA-2-7B bildet es die Grundlage aller TinyChatEngine-Messungen dieser Arbeit.

Weitere Vergleichsmodelle werden ausschließlich über llama.cpp evaluiert, um einen breiteren Referenzrahmen zu schaffen. Darunter befinden sich Gemma-3-4B [Ge25] (Google DeepMind) sowie Qwen 2.5 14B [Qw24] (Alibaba Cloud). Eine tiefergehende architekturelle Analyse dieser Modelle ist für die Kernfragestellung dieser Arbeit nicht erforderlich.

2.6 Verwandte Arbeiten

Die primäre Referenzarbeit für TinyChatEngine ist das AWQ-Paper [Li24] des MIT Han Lab, das zusammen mit der Engine veröffentlicht wurde. Es führt eine Roofline-Analyse für LLM-Inferenz durch und zeigt, dass die Token-Generierungsphase auf allen untersuch-

ten Plattformen speicherbandbreitenlimitiert ist, ein Befund, der die in Abschnitt 2.7.1 diskutierten Herausforderungen der CPU-Inferenz theoretisch untermauert. TinyChat wird im AWQ-Paper primär auf GPU-basierten Edge-Geräten (NVIDIA Jetson Orin Nano) evaluiert sowie ergänzend auf dem Raspberry Pi 4 als ressourcenbeschränktem ARM-CPU-System. Eine systematische Evaluation auf dem Raspberry Pi 5 sowie ein reproduzierbarer Vergleich mit llama.cpp unter Einbeziehung der Leistungsaufnahme sind dort nicht enthalten.

Für llama.cpp existieren in der Community umfangreiche informelle Benchmarks, die auf GitHub und im llama.cpp-Wiki gepflegt werden. Diese erfassen typischerweise Token-Raten für verschiedene Modelle und Quantisierungsstufen auf Consumer-Hardware, darunter auch ARM-basierte Systeme. Sie sind jedoch nicht nach wissenschaftlichen Standards reproduzierbar, enthalten keine Leistungsaufnahmemessungen und vergleichen llama.cpp nicht mit alternativen Inferenz-Engines wie TinyChatEngine.

Diese Arbeit schließt diese Lücke durch einen systematischen, reproduzierbaren Vergleich von TinyChatEngine 1 (AWQ) und llama.cpp (GGUF) auf identischer Hardware unter Einbeziehung der Leistungsaufnahme als Effizienzmetrik.

2.7 Herausforderungen der CPU-basierten LLM-Inferenz

Der Einsatz von LLMs auf ressourcenbeschränkten CPU-Systemen ist mit spezifischen Herausforderungen verbunden, die im Folgenden analysiert werden.

2.7.1 Strukturelle Unterschiede zwischen CPU- und GPU-Inferenz

LLM-Inferenz ist in ihrer Charakteristik ein speicherbandbreitenlimitierter Workload. Bei der autoregressiven Token-Generierung werden in jedem Schritt nahezu alle Modellgewichte aus dem Speicher geladen, während der rechnerische Aufwand pro Schritt vergleichsweise gering bleibt. GPUs profitieren hier von einer Speicherbandbreite im Bereich von mehreren Terabyte pro Sekunde sowie der Fähigkeit, Tausende von Operationen parallel auszuführen.

Der Raspberry Pi 5 bietet mit seinem LPDDR4X-Interface eine Speicherbandbreite von ca. 17 GB/s, strukturell bedingt deutlich weniger als GPU-Systeme. Tabelle 2.2 fasst die wesentlichen Unterschiede zusammen.

Auf der CPU erfolgt die Parallelisierung über SIMD-Instruktionen (auf ARM: NEON bzw. SVE) und Multi-Threading, was zwar eine gewisse Lastverteilung ermöglicht, die intrinsische Parallelität von GPU-Architekturen aber nicht erreicht. CPU-Caches sind für die großen, irregulären Speicherzugriffsmuster von LLM-Gewichtsmatrizen zudem wenig hilfreich, was die effektive Bandbreite weiter reduziert.

Merkmal	GPU (Bsp.: RTX 4090)	Raspberry Pi 5
Speicherbandbreite	1008 GB/s (GDDR6X)	≈17 GB/s (LPDDR4X)
Rechenkapazität	82,6 TFLOPS (FP16)	≈18,3 GFLOPS (FP32)
Parallele Kerne	16.384 CUDA-Kerne	4× Cortex-A76 + SIMD (NEON)
Arbeitsspeicher	24 GB VRAM (dediziert)	8–16 GB LPDDR4X (geteilt)
Leistungsaufnahme	≈450 W	≈5–7 W
Formfaktor / Kosten	Workstation / mehrere hundert bis tausende Euro	Einplatinencomputer / 80–200 €

Quellen: GPU-Spezifikation nach [Te22], RPi5-Rechenleistung: eigene Messung.

Tabelle 2.2: Strukturelle Unterschiede zwischen GPU- und CPU-Inferenz am Beispiel RTX 4090 vs. Raspberry Pi 5

2.7.2 Leistungsmessung auf eingebetteten Systemen

Die Bewertung der Effizienz eines LLM-Systems auf eingebetteter Hardware erfordert mehr als die Messung der Inferenzgeschwindigkeit. Für einen sinnvollen Vergleich verschiedener Konfigurationen sind insbesondere die Leistungsaufnahme in Watt und der Energieverbrauch pro generiertem Token relevante Kenngrößen. Der Raspberry Pi 5 bietet hierfür keine integrierte Hardwareschnittstelle. Die Stromaufnahme muss über externe Messtechnik erfasst werden. Die Implementierung dieser Messinfrastruktur stellt einen eigenen technischen Beitrag dieser Arbeit dar.

2.7.3 Zeitaufwand der Evaluation

Die Durchführung standardisierter Benchmarks auf ressourcenbeschränkter Hardware ist mit erheblichem Zeitaufwand verbunden. MMLU umfasst mehrere tausend Einzelfragen. Bei einer realistischen Inferenzgeschwindigkeit von 1–3 Tokens/s auf dem Raspberry Pi 5 kann ein vollständiger Durchlauf viele Stunden in Anspruch nehmen. Dies erfordert eine stabile, unterbrechungstolerante Messinfrastruktur sowie eine sorgfältige Planung der Evaluationsläufe, um innerhalb des verfügbaren Zeitrahmens aussagekräftige Ergebnisse zu erzielen. Als Reaktion darauf werden im Rahmen dieser Arbeit IRT-basierte TinyBenchmarks eingesetzt, die den Evaluationsaufwand bei vergleichbarer statistischer Aussagekraft drastisch reduzieren.

3 Konzept und Implementierung

Dieses Kapitel beschreibt die aufgebaute Mess- und Inferenzumgebung. Ausgangspunkt ist ein Überblick über den eingesetzten Technologie-Stack, gefolgt von einer detaillierten Beschreibung des Hardware-Aufbaus, der entwickelten Softwarekomponenten und der Messinfrastruktur.

3.1 Stack-Übersicht

Die Implementierungsumgebung setzt sich aus mehreren Komponenten zusammen, die gemeinsam eine automatisierte Inferenz-, Mess- und Evaluationsumgebung bilden. Abbildung 3.1 gibt einen Überblick über die eingesetzten Softwarekomponenten und ihre Datenflussbeziehungen.

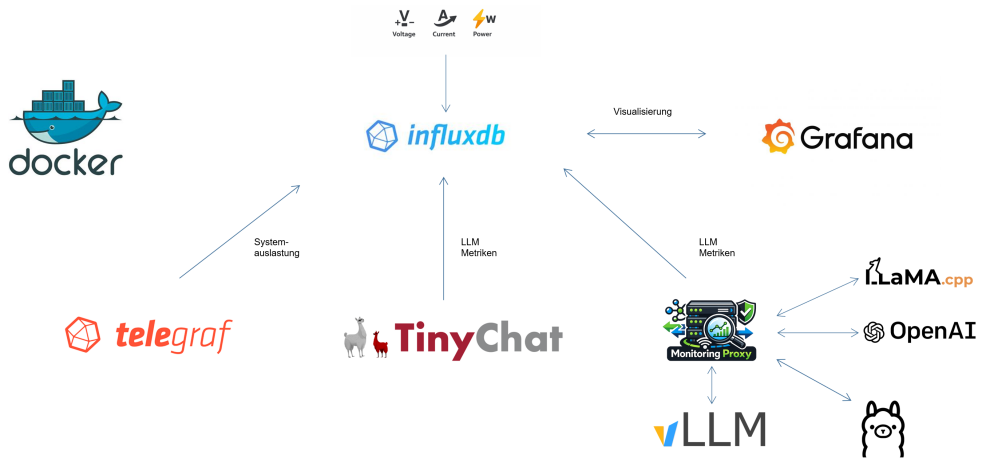

Abbildung 3.1: Visuelle Übersicht der eingesetzten Softwarekomponenten mit ihren Datenflussbeziehungen (eigene Darstellung)

Als primäres Inferenzsystem für alle Benchmarks dient der Raspberry Pi 5 mit 16 GB RAM. Der Raspberry Pi 4 übernimmt die Rolle des Monitoring-Servers und betreibt den gesamten Infrastruktur-Stack via Docker Compose. Der ESP32 erfasst die Leistungsaufnahme beider Raspberry-Pi-5-Systeme über zwei separate INA219-Leistungsmonitore und überträgt die

Messwerte in Echtzeit an InfluxDB. Tabelle 3.1 listet alle eingesetzten Komponenten im Überblick.

Komponente	**Technologie**
Inferenz-Engines	TinyChatEngine 1 (AWQ), llama.cpp (GGUF)
API-Abstraktion	LLMProxy (FastAPI, Python)
Leistungsmessung	ESP32 + 2× INA219
Datenspeicherung	InfluxDB 2.x (Zeitreihendatenbank)
Visualisierung	Grafana-Dashboard
System-Metriken	Telegraf
Infrastruktur	Docker Compose (auf RPi4)
Versionierung	GitLab (privat), GitHub (öffentlich)
Evaluation	lm-evaluation-harness, Jupyter Notebooks, LocalScore/llamafile
Hardware	RPi5 8 GB (NVMe), RPi5 16 GB (NVMe), RPi4, ESP32, 2× INA219

Tabelle 3.1: Übersicht der eingesetzten Technologien

3.2 Hardware-Aufbau

Abbildung 3.2 zeigt den physischen Aufbau: Alle Geräte sind auf einer gemeinsamen Trägerplatte montiert und über ein lokales Netzwerk verbunden.

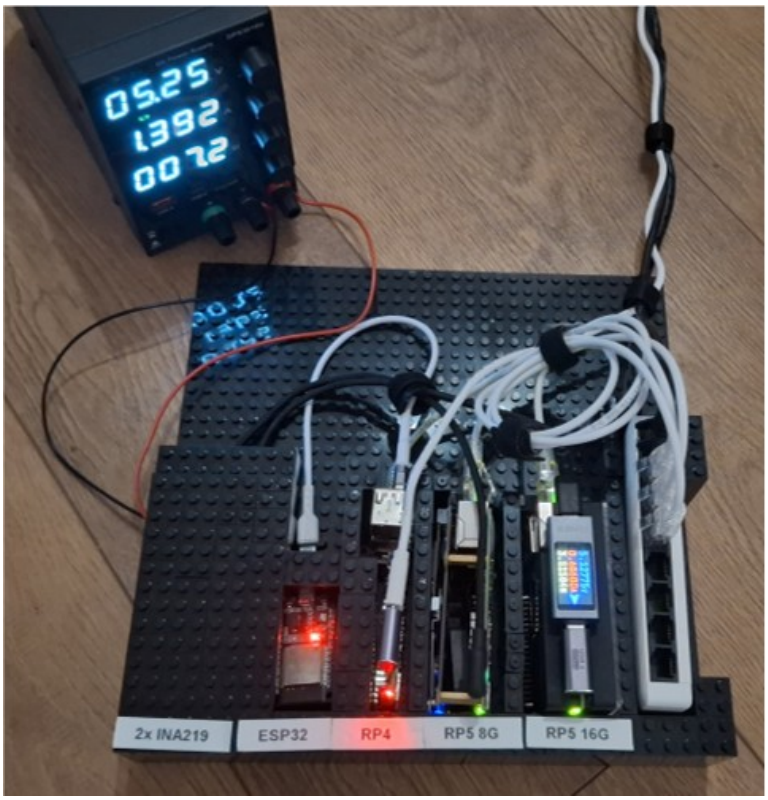

Abbildung 3.2: Physischer Aufbau der Messumgebung: INA219-Sensoren, ESP32, Raspberry Pi 4, Raspberry Pi 5 (8 GB und 16 GB), montiert auf einer gemeinsamen Trägerplatte. Im Hintergrund sind das Labornetzteil und das zur Kalibrierung verwendete USB-Multimeter zu sehen (eigene Aufnahme)

Tabelle 3.2 fasst die verwendeten Geräte und ihre Rollen zusammen.

Gerät	Spezifikation	Rolle
RPi5 8 GB	WD Black SN850 NVMe via Geekworm Shield (PCIe Gen 3)	Initiale Entwicklung, ersetzt durch RPi5 16 GB, Telegraf-Agent
RPi5 16 GB	Samsung NVMe via S2Pi M.2 Armor Case (PCIe Gen 3)	Primäres Inferenzsystem für alle Benchmarks, Telegraf-Agent
RPi4	Docker Compose	Monitoring: InfluxDB, Grafana, Telegraf und Stromversorgung ESP32
ESP32 + 2× INA219	Zwei Einzelkanal-Sensoren, je ein I2C-Bus pro Board	Leistungsmessung beider RPi5 gleichzeitig

Tabelle 3.2: Hardware-Aufbau

Auf dem RPi5 16 GB ist PCIe Gen 3 in `/boot/config.txt` aktiviert. Die NVMe-SSD dient als Boot-Medium und beschleunigt den Modell-Ladevorgang gegenüber einer SD-Karte erheblich. Bei der initial eingesetzten WD Black SN850 (1 TB) war auf dem Raspberry Pi 5 eine zusätzliche Boot-Konfiguration erforderlich. Die Geekworm-

Kompatibilitätsdokumentation nennt die WD-Black-SN850-Serie unter den NVMe[1]-SSDs[2] mit Phison-Controller, für die Boot-Inkompatibilitäten beobachtet wurden [Ge26]. Eine Samsung-NVMe-SSD funktionierte dagegen ohne diese zusätzliche Boot-Anpassung.

3.3 Modellbereitstellung und Konvertierung

Für die Evaluation wurden die eingesetzten Modelle nicht in ihrer Ausgangsform verwendet, sondern abhängig vom Backend in unterschiedliche Zielformate überführt. Methodisch entscheidend ist dabei, dass beide Backends, TinyChatEngine (AWQ) und llama.cpp (GGUF), dieselben Originalgewichte als gemeinsamen Ausgangspunkt verwenden. Damit sind beobachtete Unterschiede in Modellgüte ausschließlich auf Quantisierungsformat, Inferenz-Backend und Prompt-Pipeline zurückzuführen, nicht auf ein Fine-Tuning oder eine datensatzspezifische Anpassung der Gewichte.

3.3.1 Ausgangsformate und Bezugsquellen

Ausgangspunkt sind die Originalgewichte der Meta-LLaMA-Modelle aus dem Hugging-Face-Repository. Dabei ist zwischen zwei Ausgangsformaten zu unterscheiden:

- **LLaMA-2** liegt im klassischen Meta-Format mit `.pth`-Gewichten vor und muss vor der Weiterverarbeitung zunächst in ein Hugging-Face-kompatibles Format überführt werden.
- **LLaMA-3-8B-Instruct** liegt bereits im modernen Hugging-Face-Format mit `safetensors` vor und kann direkt weiterverarbeitet werden.

Diese Unterscheidung wirkt sich unmittelbar auf die verwendeten Konvertierungswerkzeuge aus. Für llama.cpp muss LLaMA-2 deshalb über `convert_legacy_llama.py` nach GGUF überführt werden, während für LLaMA-3 das neuere Skript `convert_hf_to_gguf.py` verwendet wird. Auch in der TinyChatEngine-AWQ-Pipeline ist für LLaMA-2 zunächst eine HF-Konvertierung notwendig, bei der zusätzlich der Tokenizer manuell kopiert und eine `tokenizer_config.json` erzeugt wird. Bei LLaMA-3 entfällt dieser Schritt, da das Modell bereits im HF-Format vorliegt.

Abbildung 3.3 fasst den Bereitstellungsprozess zusammen.

[1] Non-Volatile Memory Express
[2] Solid State Drives

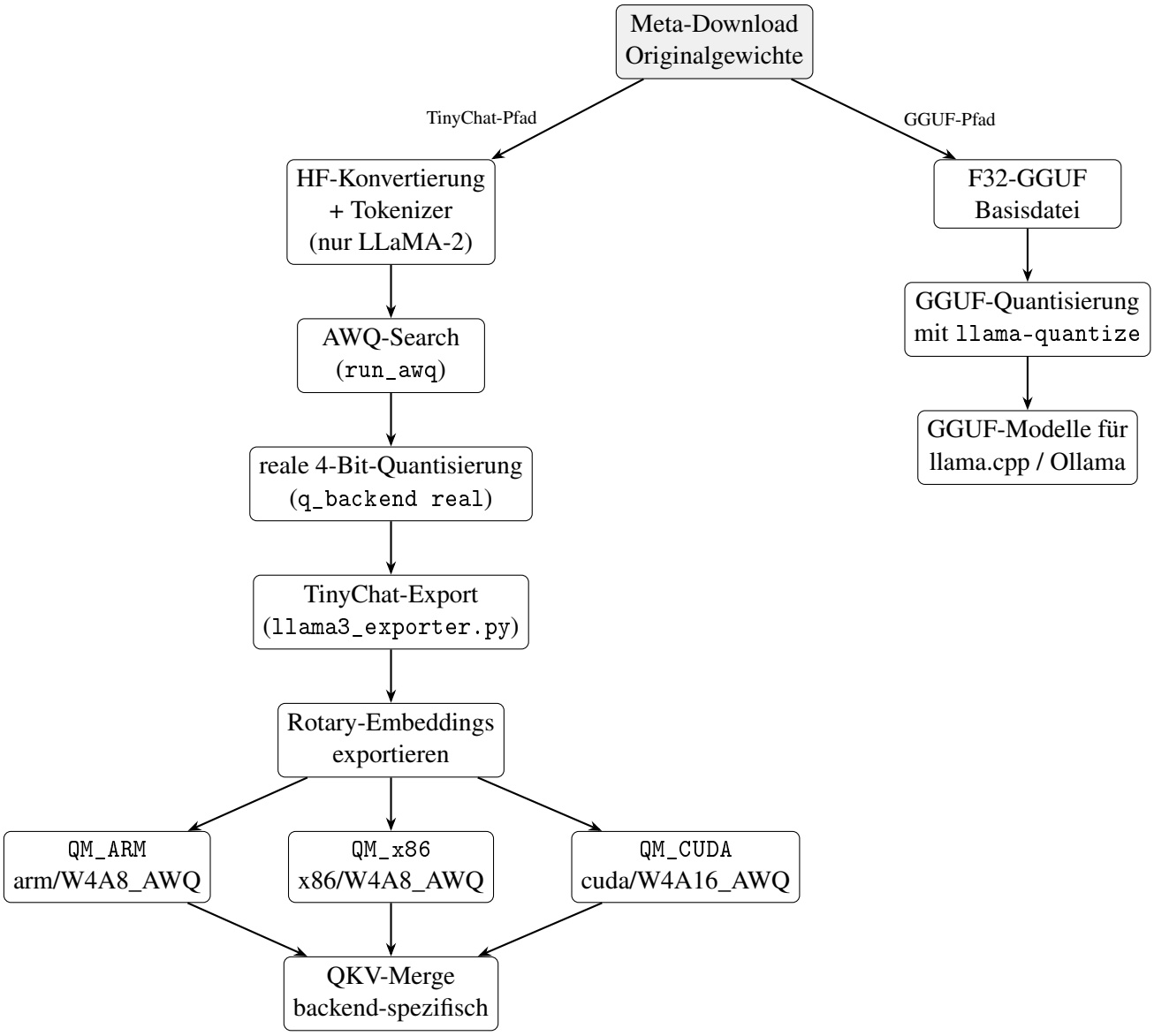

Abbildung 3.3: Bereitstellungspfade der Modelle für llama.cpp/Ollama und TinyChatEngine

3.3.2 GGUF-Erzeugung für llama.cpp und Ollama

Für llama.cpp und Ollama werden die Ausgangsgewichte zunächst in eine unquantisierte F32-GGUF-Datei konvertiert. Diese Datei bildet die gemeinsame Basis für alle weiteren Quantisierungsstufen. Anschließend erfolgt die backendseitige Quantisierung mit `llama-quantize` in die jeweils benötigten Varianten, darunter `Q4_K_M`, `Q5_K_M`, `Q8_0`, `bf16` und `fp16`.

Wesentlich ist dabei, dass alle Quantisierungsstufen direkt aus der F32-Basis erzeugt werden. Für die Evaluationsläufe auf dem Raspberry Pi 5 wurden insbesondere `Q4_K_M` als praxisnahe 4-Bit-Konfiguration sowie `Q8_0` und `bf16` als höherwertige Referenzpunkte verwendet.

Backend	Verwendete Quantisierung	Bemerkung
TinyChatEngine	W4A8_AWQ (ARM/x86), W4A16_AWQ (CUDA)	AWQ-Pipeline mit fester 4-Bit-Gewichtsquantisierung und Benchmark-Messungen für LLaMA-2-7B-Chat und LLaMA-3-8B-Instruct
llama.cpp / Ollama	`bf16`, `f16`, `q8_0`, `q6_k`, `q5_k_m`, `q5_k_s`, `q5_1`, `q5_0`, `q4_k_m`, `q4_k_s`, `q4_1`, `q4_0`, `iq4_nl`, `iq4_xs`, `q3_k_l`, `q3_k_m`, `q3_k_s`, `iq3_m`, `iq3_s`, `q2_k`	In den Benchmark-Tabellen berücksichtigte Quantisierungsstufen

Tabelle 3.3: Für die Benchmarks verwendete Quantisierungsschemata

3.3.3 AWQ-Pipeline für TinyChatEngine

Die Modellbereitstellung für TinyChatEngine folgt einem mehrstufigen AWQ-Workflow. Zunächst wird das Ausgangsmodell in ein Hugging-Face-kompatibles Layout gebracht, sofern es nicht bereits in dieser Form vorliegt. Darauf aufbauend führt `llm-awq` zunächst eine AWQ-Search zur Bestimmung geeigneter Quantisierungsparameter durch und erzeugt anschließend die real quantisierten Gewichte.

Diese Gewichte werden danach mit dem TinyChat-Exporter in das interne Modelllayout der Engine überführt. Ergänzend werden die Rotary-Embedding-Tabellen separat exportiert, da sie für den späteren Inferenzpfad explizit benötigt werden. Abschließend folgt die backendabhängige Quantisierung in die von TinyChat unterstützten Ziellayouts `QM_ARM`, `QM_x86` und `QM_CUDA`, gefolgt von der QKV[3]-Zusammenführung für die von TinyChat erwartete Speicherstruktur.

Praktisch wurde diese Exportkette zunächst auf einem Desktop-System mit CUDA-Unterstützung validiert, da die AWQ-Werkzeuge und Teile der Quantisierungskette dort robuster und deutlich schneller ausführbar waren. Für die eigentliche Evaluation auf dem Raspberry Pi 5 ist jedoch nicht der Desktop-Lauf selbst relevant, sondern das resultierende TinyChat-Modellartefakt, das anschließend auf das Zielsystem übertragen und dort über den CPU-Inferenzpfad verwendet wird.

[3] Query-Key-Value

3.4 Leistungsmessung (ESP32 + INA219)

3.4.1 Entwicklungsgeschichte: INA3221 zu INA219

Zunächst wurde ein INA3221-Breakout-Board (Texas Instruments, 3-Kanal) für die Strommessung eingesetzt. Dieses verursachte einen unakzeptablen Spannungsfall in der Versorgungsleitung des Raspberry Pi 5. Durch paralleles Schalten von drei 100 mΩ-Widerständen (effektiv 33,33 mΩ) wurde der Shunt-Widerstand reduziert. Der Spannungsfall blieb jedoch auch damit zu hoch. Als wahrscheinliche Ursache wurde das Breakout-Board selbst identifiziert.

Für den finalen Messaufbau kamen daher INA219-Sensoren zum Einsatz. Mit drei parallel geschalteten 100 mΩ-Widerständen war der Betrieb am ersten Board (RPi5 16 GB) problemlos. Für das zweite Board (RPi5 8 GB) wurden zwei parallel geschaltete Widerstände (50 mΩ) getestet, ebenfalls mit akzeptablem Spannungsfall. Beide Firmware-Projekte (`power_monitor_INA219`, `power_monitor_INA3221`) sind im Repository als eigenständige ESP-IDF-Projekte versioniert.

3.4.2 Messaufbau und Kalibrierung

Zur Messung des Leistungsverbrauchs beider Raspberry-Pi-5-Systeme wird je ein Texas Instruments INA219 eingesetzt, ein präziser, I2C[4]-basierter Leistungsmonitor mit integriertem 12-Bit-ADC[5]. Der INA219 misst kontinuierlich die Shunt-Spannung sowie die Bus-Spannung und berechnet daraus intern Strom und Leistung. Im Gegensatz zu einem Mehrkanal-Sensor wie dem INA3221 ist der INA219 ein Einzelkanal-Gerät. Für die zwei unabhängigen Boards werden daher zwei separate Sensor-Instanzen betrieben.

Elektrische Anbindung Beide Sensoren sind über den I2C-Bus mit dem Mikrocontroller verbunden, verwenden jedoch unterschiedliche Busse und GPIO-Pins, um Adresskonflikte zu vermeiden, da beide Sensoren dieselbe I2C-Adresse 0x40 besitzen:

Parameter	**Board A (RPi5 16 GB)**	**Board B (RPi5 8 GB)**
I2C-Bus	Bus 1	Bus 0
SDA	GPIO 21	GPIO 25
SCL	GPIO 22	GPIO 26
I2C-Adresse	0x40	0x40

Tabelle 3.4: Elektrische Anbindung der INA219-Sensoren am Mikrocontroller

[4] Inter-Integrated Circuit
[5] Analog-Digital-Wandler

Shunt-Widerstände Als Shunt-Widerstände werden handelsübliche 100 mΩ-Präzisionswiderstände verwendet, die den zuvor eingesetzten INA3221-Breakout-Boards entnommen wurden und zur Reduzierung des Gesamtwiderstands sowie des Spannungsabfalls parallel verschaltet werden. Für Board A sind drei Widerstände parallel geschaltet (theoretischer Gesamtwiderstand: 33,33 mΩ), für Board B zwei Widerstände (theoretisch: 50,00 mΩ).

INA219-Konfiguration Die Sensoren werden mit folgenden Registereinstellungen betrieben (gesetzt via `ina219_configure()` im Treiber):

Parameter	Einstellung
Bus-Spannungsbereich	16 V (`INA219_BUS_RANGE_16V`)
PGA-Verstärkung (Shunt)	±40 mV (`INA219_GAIN_0_125`)
ADC-Auflösung	12 Bit, 128-Sample-Mittelung (`INA219_RES_12BIT_128S`)
Messmodus	Kontinuierlich, Shunt + Bus (`INA219_MODE_CONT_SHUNT_BUS`)

Tabelle 3.5: INA219-Registerkonfiguration

Die 128-Sample-Mittelung wird sowohl für den Shunt-Kanal als auch für den Bus-Spannungskanal aktiviert. Laut INA219-Datenblatt ergibt sich dabei eine Konversionszeit von 68,1 ms pro Kanal [Te15, Table 5], was einem vollständigen Messzyklus (Shunt + Bus) von ca. 136,2 ms entspricht. Die Firmware liest die Messwerte in einem FreeRTOS-Task mit 100-ms-Takt aus. Da der Sensor intern kontinuierlich mittelt, wird bei jedem Lesevorgang der aktuell verfügbare gemittelte Wert abgerufen.

Kalibrierung Die Kalibrierung erfolgte in zwei Schritten mithilfe eines Labornetzteils als Referenzquelle sowie eines kalibrierten Digitalmultimeters, das intern auf dem Texas Instruments INA226 basiert, mit einer spezifizierten Genauigkeit von ±(1 % + 5 Counts).

Shunt-Widerstand: Der tatsächliche Parallelwiderstand weicht aufgrund von Bauteil-Fertigungstoleranzen vom theoretischen Wert ab. Der reale Widerstandswert wurde durch Messung einer bekannten Referenzspannung und des resultierenden Stroms bestimmt und als kalibrierter Shunt-Wert im System hinterlegt.

Spannungsskalierung: Ein dimensionsloser Korrekturfaktor kompensiert systematische Abweichungen in der Bus-Spannungs- und Leistungsmessung. Dieser Faktor wird nach der Rohmessung auf die Bus-Spannung und die berechnete Leistung multipliziert (in `INA219Sensor::read_measurements()`).

Die kalibrierten Werte sind pro Board im Konfigurationssystem (ESP-IDF sdkconfig) hinterlegt (Tabelle 3.6). Die Abweichung der kalibrierten Shunt-Werte vom theoretischen Wert

beträgt bei Board A ca. $+1{,}3$ % (33,758 vs. 33,33 mΩ) und bei Board B ca. $+1{,}4$ % (50,680 vs. 50,00 mΩ), was im typischen Toleranzbereich handelsüblicher Metallfilm-Widerstände liegt. Die Spannungsskalierungsfaktoren von rund 0,972–0,974 weisen auf eine systematische Überschätzung der Bus-Spannung von ca. 2,7 % hin, die durch Referenzmessung mit dem Multimeter korrigiert wurde. Als möglicher Einflussfaktor kommen Leitungswiderstände sowie Kontaktwiderstände von Steckverbindungen infrage, die zu einem messbaren Spannungsabfall zwischen INA219-Messpunkt und dem tatsächlichen Versorgungseingang des Raspberry Pi führen können.

Messunsicherheit Unter Berücksichtigung der Referenzgerätgenauigkeit (±1 %) sowie des verbleibenden INA219-Eigenfehlers nach Kalibrierung (max. ±0,5 % laut Datenblatt [Te15]) ergibt sich eine geschätzte Gesamtmessunsicherheit von ca. ±1,5 %. Für den Zweck dieser Arbeit, den relativen Vergleich der Leistungsaufnahme verschiedener Backends und Konfigurationen, ist diese Genauigkeit ausreichend.

Parameter	**Board A (RPi5 16 GB)**	**Board B (RPi5 8 GB)**
Shunt-Widerstand (theoretisch)	33,33 mΩ	50,00 mΩ
Shunt-Widerstand (kalibriert)	33,758 mΩ	50,680 mΩ
Spannungs-skalierungsfaktor	0,972121	0,973554

Tabelle 3.6: Kalibrierte Werte der INA219-Sensoren pro Board

3.4.3 Datenfluss und Auswertung

Der ESP32 überträgt die Messwerte über das Netzwerk im InfluxDB-Line-Protocol direkt an die InfluxDB-Instanz auf dem RPi4. Die zeitliche Auflösung der Messung erlaubt die Unterscheidung von Modell-Laden, Prefill-Berechnung und Token-Generierung im Leistungsprofil. Aus der Gegenüberstellung von Leistungsaufnahme und Inferenzgeschwindigkeit lässt sich der **Energieverbrauch pro generiertem Token** als vergleichbare Effizienzmetrik berechnen. Das zugehörige Grafana-Dashboard mit den aufgezeichneten Leistungsverläufen beider INA219-Kanäle ist in Abbildung A.3 im Anhang dargestellt.

3.5 LLMProxy

3.5.1 Architektur und API

Der LLMProxy ist ein eigens entwickelter FastAPI-Proxy (Python), der als Abstraktionsschicht zwischen Benchmark-Clients und den Inferenz-Backends sitzt. Er stellt eine einheitliche OpenAI-kompatible `/v1/chat/completions`-API bereit, unabhängig vom jeweiligen Backend.

Unterstützte Backends:

- TinyChatEngine 1 (über SSE[6]-Protokoll via `/settings` und `/chat`)
- Ollama, llama.cpp sowie vLLM (direkt OpenAI-kompatibel)

Die Konfiguration erfolgt über `config.json` mit Backend-URLs, Modellnamen, Datenformaten und InfluxDB-Verbindungsparametern. Dadurch sind automatisierte Benchmark-Läufe über alle Engines hinweg ohne Code-Änderungen möglich.

3.5.2 Metrik-Erfassung

Der LLMProxy erfasst pro Request folgende Metriken direkt in InfluxDB (Tabelle 3.7):

	Zeitpunkt	Felder
`generation_start`	1× pro Request	`temperature`, `top_p`, `top_k`, `max_tokens`
`token_timing`	Pro Token	`time_from_previous_ms` (Inter-Token-Abstand bzw. TTFT für Token 0), `position`
`token_cumulative`	Pro Token	`tokens_generated`, `total_time_ms`, `tokens_per_second`, `avg_token_time_ms`, `avg_token_time_no_ttft_ms`

Tabelle 3.7: In InfluxDB erfasste Metriken des LLMProxy. Alle Measurements tragen die Tags `model_name`, `data_format`, `request_id` und `source=proxy`

Das Feld `avg_token_time_no_ttft_ms` berechnet die mittlere Inter-Token-Zeit exklusive des ersten Tokens. Damit lässt sich die reine Decode-Phase unabhängig vom Prefill-Overhead bewerten.

[6] Server-Sent Events

3.6 TinyChatEngine

3.6.1 Kompilierung und Konfiguration

TinyChatEngine 1 wird aus dem Quellcode des MIT Han Lab kompiliert und für den Einsatz auf dem Raspberry Pi 5 konfiguriert. Die Engine setzt AWQ-quantisierte Modellgewichte voraus, die entweder mit der zugehörigen Quantisierungs-Toolchain erzeugt oder als vorbereitete Artefakte bezogen werden können. Entwicklung und erste Erprobung erfolgten zunächst auf einem Desktop-System (64-Bit AMD-CPU, NVIDIA-GPU), bevor die Umgebung auf den Raspberry Pi 5 portiert wurde. Dieser zweistufige Ansatz ermöglichte es, quellcodeseitige Probleme von plattformspezifischen Einschränkungen getrennt zu behandeln. Die angepassten Pfade der quantisierten Modellgewichte werden in architekturspezifischen Unterverzeichnissen abgelegt (`arm/W4A8_AWQ/` für ARM, `x86/W4A8_AWQ/` für x86, `cuda/W4A16_AWQ/` für GPU), damit Artefakte verschiedener Zielarchitekturen sich nicht gegenseitig überschreiben.

3.6.2 Korrekturen und Stabilitätsverbesserungen

CUDA-Kompatibilität und automatische GPU-Architekturerkennung

Obwohl der Fokus dieser Arbeit auf CPU-basierter Inferenz liegt, war die Desktop-Entwicklungsumgebung mit CUDA-fähiger GPU ein wesentlicher Bestandteil des Entwicklungs- und Verifikationsprozesses. Sie ermöglichte es, Korrekturen an TinyChatEngine iterativ zu testen, ohne für jeden Schritt den deutlich langsameren Zyklus auf dem Raspberry Pi 5 durchlaufen zu müssen, sowie zur Bestimmung der Modellgüte. Damit derselbe Quellcode ohne manuelle Anpassungen auf unterschiedlichen Systemen kompilierbar ist, wurden zwei Probleme im originalen Build-System behoben.

Hardcodierte GPU-Zielarchitektur Das originale `Makefile` von TinyChatEngine setzt die Compute Capability der Ziel-GPU fest auf `sm_86` (NVIDIA Ampere, z. B. RTX 3080/3090):

```
1 # Please modify 'arch=compute_86,code=sm_86' according to your GPU
2 CXXFLAGS = ... -gencode arch=compute_86,code=sm_86 ...
```

Listing 3.1: Original: hardcodierte sm_86-Architektur (org/TinyChatEngine/llm/Makefile, Zeile 58–59)

Wird TinyChatEngine auf einem System mit einer anderen GPU-Generation kompiliert, schlägt der Build fehl oder erzeugt Code, der auf der vorhandenen Hardware nicht problemlos ausführbar ist.

Im modifizierten Makefile wurde dies durch eine automatische Erkennung der Compute Capability zur Build-Zeit ersetzt:

```
NVIDIA_SMI := $(shell which nvidia-smi 2>/dev/null)
ifneq ($(NVIDIA_SMI),)
    GPU_COMPUTE_CAP := $(shell nvidia-smi --query-gpu=compute_cap \
        --format=csv,noheader | head -n 1 | tr -d '.')
    ifneq ($(GPU_COMPUTE_CAP),)
        GPU_ARCH    := sm_$(GPU_COMPUTE_CAP)
        GPU_GENCODE := arch=compute_$(GPU_COMPUTE_CAP),code=sm_$(
            GPU_COMPUTE_CAP)
    else
        GPU_ARCH := sm_86 # Fallback
    endif
else
    GPU_ARCH := sm_86 # Fallback
endif
```

Listing 3.2: TinyChatEngineAPI: automatische GPU-Architekturerkennung via nvidia-smi (tinychatengineapi/llm/Makefile, Zeile 47–65)

Das Build-System ruft `nvidia-smi --query-gpu=compute_cap` auf, liest die Compute Capability der installierten GPU direkt aus und leitet daraus `GPU_ARCH` und `GPU_GENCODE` ab. Ist `nvidia-smi` nicht verfügbar oder liefert es keinen Wert, greift ein Fallback auf `sm_86` zurück. Damit ist das Build-System portabel: Es kompiliert korrekt für RTX 30xx (sm_86), RTX 40xx (sm_89), NVIDIA Jetson und weitere Zielplattformen, ohne dass eine manuelle Änderung im Makefile erforderlich ist.

nvcc-Erkennung außerhalb des PATH Das originale Makefile beschränkt die `nvcc`-Suche auf den `$PATH`: Beide Plattformzweige (Windows und Linux) verwenden dafür identisch `command -v nvcc 2> /dev/null`, sodass CUDA-Installationen außerhalb des `$PATH` (typischerweise unter `/usr/local/cuda/`) nicht erkannt werden. Im modifizierten Build-System ersetzt eine zweistufige Erkennungslogik die bisherigen getrennten Plattformzweige: Zunächst wird `nvcc` über den `$PATH` gesucht. Schlägt diese Prüfung fehl, wird `/usr/local/cuda/bin/nvcc` als Fallback herangezogen:

```
CUDA_AVAILABLE := $(shell command -v nvcc 2> /dev/null)
ifeq ($(CUDA_AVAILABLE),)
    CUDA_AVAILABLE := $(shell command -v /usr/local/cuda/bin/nvcc 2> /dev
        /null)
endif
```

Listing 3.3: TinyChatEngineAPI: erweiterter nvcc-Suchpfad (tinychatengineapi/llm/Makefile, Zeile 37–41)

Beide Anpassungen zusammen stellen sicher, dass dasselbe Repository auf unterschiedlichen Desktop-Entwicklungssystemen mit CUDA-fähiger GPU ohne Quellcode-Änderungen kompiliert werden kann.

Out-of-Memory auf dem Raspberry Pi 5 (8 GB)

Auf dem Raspberry Pi 5 mit 8 GB RAM führte der Versuch, ein 7B/8B-Modell zu laden, zu Speicherengpässen bis hin zum erzwungenen Prozessabbruch durch den Linux-OOM[7]-Killer. Das verfügbare RAM ist bei einem AWQ-quantisierten 7B-Modell knapp bemessen, da neben den Modellgewichten auch KV-Cache, Aktivierungen und Betriebssystem-Overhead Speicher beanspruchen.

Als Maßnahme wurde ein Memory-Priority-Skript eingesetzt, das den Inferenzprozess gegenüber anderen Systemprozessen priorisiert. Darüber hinaus wurde die Kontextlänge auf ein Minimum reduziert und der Betrieb paralleler Anwendungen während der Inferenz vermieden. Unter diesen Bedingungen erwies sich der Betrieb als hinreichend stabil für erste Benchmarks, ist jedoch für den produktiven Einsatz mit 8 GB nicht empfehlenswert. Für die meisten finalen Messungen wurde daher auf den RPi5 16 GB gewechselt.

Unterstützung des offiziellen LLaMA-3-Prompt-Templates

Während der Integration der OpenAI-kompatiblen Schnittstelle traten bei LLaMA-3 auffällige Unterschiede im Antwortverhalten auf, die auf die verwendete Prompt-Repräsentation zurückgeführt werden konnten. Um diesen Faktor kontrolliert untersuchen zu können, wurde neben dem bisherigen TinyChatEngine-Prompt-Format das offizielle LLaMA-3-Instruct-Template implementiert.

LLaMA-3-Instruct-Modelle erwarten ein Prompt-Schema mit den Special-Tokens `<|begin_of_text|>`, `<|start_header_id|>`, `<|end_header_id|>` sowie `<|eot_id|>`. Diese Tokens sind in den Modellgewichten mit fest definierten IDs (128000–128009) kodiert und werden vom BPE[8]-Tokenizer nicht im Standardpfad erzeugt. Für die experimentelle Gegenüberstellung wurden deshalb zwei Formatierungsmodi in der API und im Prompt-Builder implementiert, deren Auswirkungen in Kapitel 4 evaluiert werden.

Die technische Umsetzung umfasst einen Pre-Scan in der `encode()`-Funktion, der exakt fünf Special-Tokens vor dem regulären BPE-Durchlauf erkennt und die hardcodierten Token-IDs direkt in den Tokenstream einbettet. Das gewünschte Prompt-Format wird in der

[7] Out of Memory
[8] Byte Pair Encoding

`config.json` über das Feld `llama3_prompt_format` ausgewählt. Ein Beispiel ist in Listing A.1 im Anhang dargestellt.

Korrekturen im Tokenizer und KV-Cache

Im Tokenizer von TinyChatEngine wurde für den Triple-Merge-Pfad des BPE-Algorithmus ein zu knapper Puffer allokiert (`max_token_length * 2` statt `* 3 + 3`). Bei seltenen Zeichenkombinationen führte dies zu einem Buffer-Overflow mit undefiniertem Verhalten. Zusätzlich wurden fehlerhafte `free()`-Aufrufe auf mit `new[]` allokiertem Speicher behoben. Beide Fehler können bei bestimmten Eingaben zu Abstürzen oder stillen Datenkorruptionen führen und damit Benchmark-Ergebnisse verfälschen.

Der KV-Cache in `Int4llamaAttention` verwendete rohe `float***`-Pointer ohne definierte Lebensdauer. Dies wurde auf `std::vector<std::array<float*, 2>>` mit expliziter `free_cache_buffers()`-Funktion umgestellt. `initialized_memory()` ruft nun zuverlässig Cleanup auf, bevor neuer Speicher allokiert wird. Ohne diese Korrektur führten wiederholte Benchmark-Läufe auf dem RPi5 zu akkumulierendem Speicherverbrauch und destabilisierten das System schrittweise.

Eine weitere Anpassung betraf den CUDA-Pfad für LLaMA-3-Modelle mit GQA. LLaMA-3-8B verwendet 8 KV-Heads und 32 Query-Heads, wodurch `kv_dim` kleiner als `embed_dim` ist. Im ARM- und x86-Pfad von TinyChatEngine war diese Aufteilung bereits abgebildet, während der CUDA-Pfad an dieser Stelle noch um die LLaMA-3-spezifische GQA-Behandlung ergänzt werden musste.

3.6.3 Logging-Erweiterung

TinyChatEngine gibt in der Standardkonfiguration nur begrenzte Laufzeitinformationen aus. Für die Zwecke dieser Arbeit wurde das Logging-Verhalten der Engine um `influx_logger.cc` erweitert, das relevante Inferenzmetriken (Tokens/s, TTFT sowie Speicherauslastung) strukturiert und maschinenlesbar direkt in InfluxDB schreibt. Die gemeinsame Datenbasis ermöglicht die zeitliche Korrelation mit den Leistungsmessungen des ESP32-Systems.

Das Logging-Modul wurde nachträglich auf ein asynchrones Design umgestellt. Ein Background-Worker-Thread verarbeitet Schreiboperationen aus einer bounded Deque (maximal 1024 Einträge), sodass Netzwerklatenzen zur InfluxDB-Instanz den Token-Callback-Pfad nicht blockieren. Dies ist für die Messgüte relevant: Ein synchrones Design würde die Netzwerklatenz in die gemessene Token-Rate einrechnen und das Ergebnis systematisch verfälschen. Bei Überlastung des Schreibpfads werden älteste Einträge verworfen und über eine

`dropped_count()`-Metrik nachvollziehbar gemacht. Ergänzend schreibt ein `TokenLogger` JSON-Logs lokal auf Disk, die als Fallback bei Netzwerkproblemen dienen. Das resultierende Grafana-Dashboard mit den erfassten Inferenzmetriken ist in Abbildung A.4 im Anhang abgebildet.

3.6.4 Gesprächsverwaltung und Kontextlänge

Für automatisierte Benchmark-Läufe ist es zwingend, dass kein Konversationszustand zwischen den Messungen erhalten bleibt. TinyChatEngine verfügte in der Originalversion über keinen definierten Reset-Mechanismus. Es wurden daher `LLaMA3ResetConversationState()` sowie analoge Funktionen für alle weiteren unterstützten Modellfamilien (LLaMA, Mistral, StarCoder, OPT) implementiert. `Int4llamaAttention::reset_cache()` setzt dabei ausschließlich Indizes und Zähler zurück, ohne Speicher freizugeben, da der KV-Cache-Puffer eine feste Größe hat. Statische KV-Variablen wurden von function-local auf file-scope verschoben, damit sie über die Reset-Funktionen adressierbar sind. Der Reset ist über den `POST /reset`-Endpunkt der REST-API exponiert und wird vor jedem Benchmark-Lauf automatisch aufgerufen.

In der Originalversion von TinyChatEngine war die effektive Laufzeitgrenze für LLaMA-3-8B-Instruct im ARM/INT4-Pfad durch die Modellkonfiguration `max_sqlen = 2048` bestimmt. Dieser Wert wurde zur Allokation der Decoder-, Attention- und Rotary-Puffer verwendet und stellte damit die harte Kontextgrenze dieses Inferenzpfads dar. Andere Kontextwerte, etwa `generation_config.n_ctx` oder die in den Exportwerkzeugen erzeugten Rotary-Tabellen, bestimmten diese ARM-Laufzeitgrenze nicht unmittelbar.

Bei Überschreitung dieser Grenze bot die Originalversion keinen definierten, nutzerseitig auswertbaren Fehlerpfad. Je nach Codepfad und Build-Konfiguration konnte der Prozess durch ein fehlschlagendes `assert()` im Rotary-Pfad abbrechen oder, bei deaktivierten Assertions, mit Zugriffen außerhalb der vorgesehenen Puffer weiterlaufen. Zusätzlich war bereits die Tokenisierung langer Prompts kritisch, da der Eingabepuffer auf 2048 Tokens dimensioniert war und die Tokenizer-Routine keine ausreichende Begrenzung beim Schreiben in diesen Puffer erzwang.

Für die im Rahmen dieser Arbeit entwickelte TinyChatEngineAPI wurde die maximale Sequenzlänge durch die Konstante `MAX_SEQUENCE_LENGTH = 6144` explizit gemacht und die betroffenen Modellkonfigurationen wurden entsprechend angepasst. Sequenzlängen-Überschreitungen werden nicht mehr als harter Prozessabbruch behandelt, sondern als `std::runtime_error` bis in den API-Layer propagiert und dort als HTTP-Fehler beantwortet. Dieses Verhalten ist relevant für Benchmarks mit langen Few-Shot-Kontexten: Ein überlanger Request schlägt definiert fehl, ohne den Serverprozess zu beenden oder nach-

folgende Benchmark-Läufe zu beeinflussen. Die Anzahl der Inferenz-Threads ist über das Konfigurationsfeld `threads` in `config.json` einstellbar.

3.6.5 REST-API (TinyChatEngineAPI)

Um automatisierte Benchmark-Läufe zu ermöglichen, wurde TinyChatEngine 1 um eine umfangreiche REST-API erweitert. Vier Subkomponenten bilden die API-Schicht.

Grundlegende API-Architektur

Die API (`api.cc`) basiert auf der `cpp-httplib`-Bibliothek (als Git-Submodul eingebunden) und läuft im selben Prozess wie die Inferenz-Engine. Ein `APIServerState`-Struct mit `std::atomic<bool>` Concurrency-Lock verhindert parallele Inferenzanfragen, da TinyChatEngine grundsätzlich kein paralleles Request-Handling unterstützt. Ein zweiter Request erhält HTTP 503, solange eine Inferenz läuft. Antworten werden über SSE gestreamt: Der Token-Callback schreibt jeden generierten Token als Ereignis in die HTTP-Response. Exponierte Endpunkte: `GET /health`, `POST /chat` (SSE-Inferenz), `POST /settings` (Laufzeit-Parameteranpassung ohne Prozess-Neustart) sowie `POST /reset` (Gesprächszustand zurücksetzen).

OpenAI-kompatible Schnittstelle

Für die Integration mit dem lm-evaluation-harness und dem LLMProxy wurde eine OpenAI-kompatible API-Schicht implementiert (`openai_api.h`). Exponierte Endpunkte: `GET /v1/models` sowie `POST /v1/chat/completions` mit Unterstützung für Streaming (SSE) und Non-Streaming-Antworten. Multi-Turn-Konversationen werden durch `build_full_conversation_prompt()` unterstützt, das eine Liste von Message-Objekten im `role/content`-Format in ein vollständiges Prompt im modellspezifischen Template überführt. Bekannte Einschränkungen gegenüber der vollständigen OpenAI-API (kein Function Calling, kein paralleles Request-Handling) sind dokumentiert und haben keinen Einfluss auf die in dieser Arbeit verwendeten Benchmark-Verfahren. Ein `POST /change_model`-Endpunkt ermöglicht den Modellwechsel über einen `execv()`-basierten Prozess-Neustart. Ein `restart_api.sh`-Daemon-Wrapper sorgt dafür, dass der Prozess nach dem Neustart automatisch wieder erreichbar ist.

Prefix-KV-Cache

Wird ein gemeinsamer Präfix-Kontext, etwa ein System-Prompt oder ein Few-Shot-Block, für viele aufeinanderfolgende Anfragen verwendet, verarbeitet das Modell diesen Anteil standardmäßig bei jeder Anfrage neu. Bei langen Präfixen ist dies ein erheblicher Rechenaufwand. Mit der Prefix-KV-Cache-Funktion wird der KV-Zustand des Präfixes einmalig vorberechnet und für alle nachfolgenden Anfragen wiederverwendet.

Ein `PrefixKVCache`-Struct speichert den vorberechneten KV-Zustand und persistiert diesen in einem binären Format auf Disk, sodass der Cache auch nach einem Server-Neustart sofort verfügbar ist. Die Funktionen `build_llama_prefix_cache()` und `LLaMA3SetPrefixKVCache()` integrieren den Cache in den Generierungspfad. Die Cache-Identifikation erfolgt inhaltsbasiert: Eine Hash-ID wird aus Modellname, Konfigurationsdimensionen, Vokabular-Fingerprint und der Präfix-Token-Sequenz berechnet. Ändern sich Modell oder Konfiguration, entsteht eine andere Hash-ID und der gespeicherte Cache wird nicht mehr geladen. Die alten Binärdateien verbleiben dabei auf Disk und werden erst durch einen expliziten `POST /prefix_cache/clear`-Aufruf entfernt. Die Implementierung ist ausschließlich im CPU-Pfad verfügbar. Eine Unterstützung des CUDA-Pfads hätte zusätzliche Anpassungen an der dortigen Cache-Verwaltung und am Generierungspfad erfordert. Da der Fokus dieser Arbeit auf CPU-basierter Inferenz auf dem Raspberry Pi 5 liegt und für GPU-Systeme nur ein begrenzter Performancegewinn durch diesen Mechanismus zu erwarten ist, wurde diese Erweiterung nicht umgesetzt. Eine `/prefix_cache`-Endpunktgruppe in der API ermöglicht das Aufbauen, Laden und Invalidieren des Caches zur Laufzeit.

Logprob-Pfad und lm-eval-Integration

Der lm-evaluation-harness verwendet für Multiple-Choice-Benchmarks wie MMLU und TinyBenchmarks nicht die generative API, sondern einen Loglikelihood-Pfad: Für jede Antwortoption wird die Log-Wahrscheinlichkeit $\log P(\text{Option} \mid \text{Kontext})$ unter dem Modell berechnet. Die Option mit dem höchsten Wert gilt als Modell-Antwort. Da die originale TinyChatEngine diesen Pfad nicht anbietet, war eine Eigenimplementierung erforderlich.

Zwei komplementäre Wege wurden implementiert. Erstens: `POST /logprobs` mit dem Modul `LogProbCompute.h/cc`, das `compute_logprobs_llama3()` und `compute_logprobs_llama()` für klassische Multiple-Choice-Anfragen bereitstellt. Zweitens: Der standardisierte lm-eval-Protokollpfad über `POST /v1/completions` mit den Parametern `echo=true` und `logprobs>=1`. Intern wird dabei `compute_sequence_logprobs()` aufgerufen, das die Log-Wahrscheinlichkeiten für eine vorgegebene Token-Sequenz berechnet.

3.7 llama.cpp und Ollama

llama.cpp wird nativ auf dem RPi5 kompiliert und als Server (`llama-server`) betrieben, der eine OpenAI-kompatible API bereitstellt. Getestete GGUF-Quantisierungsstufen: Q4_K_M, Q8_0, fp16, bf16, QAT.

Speculative Decoding ist ein in llama.cpp integriertes Verfahren zur potenziellen Beschleunigung der Token-Generierung. Ein kleineres Draft-Modell schlägt in jedem Schritt mehrere Token vor. Das Hauptmodell verifiziert diese anschließend in einem einzigen Forward-Pass. Bei hoher Akzeptanzrate der Draft-Token können so mehrere autoregressive Schritte bei einem einzigen vollständigen Gewichtsladevorgang des Hauptmodells realisiert werden, was auf GPU-Systemen regelmäßig zu messbaren Durchsatzgewinnen führt.

Auf dem Raspberry Pi 5 wurde LLaMA 3.2 1B Instruct (Q4_K_M) als Draft-Modell in Kombination mit LLaMA 3.1 8B Instruct (Q4_K_M) als Hauptmodell evaluiert. Die Konfiguration sah maximal 16 Draft-Token (`draft_max=16`), mindestens 2 (`draft_min=2`) sowie einen minimalen Sampling-Schwellenwert von 0,9 (`draft_p_min=0.9`) vor. Die Messergebnisse und ihre Interpretation sind in Abschnitt 4.1.5 dargestellt.

Ollama vereinfacht den Betrieb erheblich: Modelle werden per Pull-Befehl heruntergeladen, Quantisierungsstufen sind frei wählbar. Intern basiert Ollama ebenfalls auf llama.cpp und wird über den LLMProxy eingebunden.

vLLM bietet offiziell ARM-Support und wurde als weiteres Backend evaluiert. Der Build-Prozess scheiterte auf dem Raspberry Pi 5 16 GB aufgrund unzureichender RAM-Ressourcen: Der Compiler überschritt den verfügbaren Arbeitsspeicher, sodass vLLM nicht erfolgreich gebaut werden konnte und daher nicht weiter eingesetzt wurde.

3.8 Monitoring-Stack

Der Monitoring-Stack wird via Docker Compose auf dem Raspberry Pi 4 betrieben und umfasst:

- **InfluxDB 2.x**: Zeitreihendatenbank für alle Metriken (Inferenz und Leistung)
- **Grafana**: Visualisierung und zeitliche Korrelation von Leistungs- und Inferenzdaten
- **Telegraf**: Erfassung von System-Metriken (CPU, RAM, Temperatur) des Inferenz-RPi

Der Stack erfüllte dabei zwei komplementäre Funktionen. Einerseits bildet die gemeinsame Zeitreihendatenbank die Grundlage für die Messdatenerfassung: Leistungs- und Inferenzdaten lassen sich in Grafana in einem einheitlichen Dashboard zusammenführen und zeitlich korrelieren.

Andererseits war der Monitoring-Stack während der gesamten Entwicklungsphase ein wichtiges Werkzeug zur Überwachung der Systemstabilität. TinyChatEngine erwies sich insbesondere in frühen Entwicklungsstadien und nach Code-Änderungen als gelegentlich instabil. Bei langen Benchmark-Läufen, die auf dem Raspberry Pi 5 mehrere Tage in Anspruch nehmen können, ermöglichte das Grafana-Dashboard eine kontinuierliche Fernüberwachung: Abstürze, Speicherlecks oder thermische Probleme ließen sich so frühzeitig erkennen, ohne den laufenden Test zu unterbrechen. Ein kombiniertes Übersichts-Dashboard, das Leistungs- und Inferenzdaten einer Messsitzung zusammenfasst, ist in Abbildung A.2 im Anhang dargestellt. Die detaillierten Einzel-Dashboards für Leistungsmessung und TinyChatEngine-Inferenzmetriken finden sich in den Abbildungen A.3 und A.4.

3.9 Benchmarks und Datensätze

Die Bewertung der Modellgüte erfolgt über zwei ergänzende Evaluationswege. Für TinyBenchmarks wird das lm-evaluation-harness (EleutherAI, v0.4.0+) eingesetzt, ein standardisiertes Framework mit OpenAI-API-Backend, das sich direkt mit dem LLMProxy kombinieren lässt. Für die vollständigen MMLU- und MATH500-Auswertungen werden ergänzend die Notebooks von Sebastian Raschka [Ra25] verwendet, um die Vergleichbarkeit mit der parallel entstandenen Arbeit von Heiming [He26] sicherzustellen.

Die Benchmark-Läufe und Auswertungen sind als Jupyter-Notebooks organisiert, die je nach Benchmark über den LLMProxy mit den jeweiligen Inferenz-Backends kommunizieren oder die generierten Antworten und Logprob-Bewertungen nachgelagert auswerten. Im Verlauf der Arbeit wurden bei TinyBenchmarks zwei Varianten gegenübergestellt: eine Variante mit explizitem Chat-Template (`chat template`) sowie eine Variante ohne zusätzliche Formatierung (`unformatted`). Zwei Notebooks bilden den Kern der Evaluationsinfrastruktur:

- `tinybenchmarks_eval_v3 llamacpp.ipynb`: Benchmark- und Auswertungsnotebook für llama.cpp/Ollama einschließlich der getesteten GGUF-Quantisierungsstufen.
- `tinybenchmarks_eval_v3 tinychatengine.ipynb`: analoges Notebook für TinyChatEngine und die Gegenüberstellung verschiedener Prompt-Formatierungen.

Die Ergebnisse werden innerhalb der Notebooks ausgewertet und für die in Kapitel 4 dargestellten Tabellen und Grafiken aufbereitet.

Tabelle 3.8 gibt eine Übersicht der eingesetzten Benchmarks.

Benchmark	Beschreibung	Zweck
MMLU	Multiple-Choice, 57 Wissensgebiete, über 14 000 Fragen	Allgemeines Sprachwissen und Reasoning, Vergleich mit publizierten Referenzwerten
MATH500	500 mathematische Aufgaben, verschiedene Schwierigkeitsstufen	Strukturiertes Schlussfolgern
TinyBenchmarks	IRT-basierte Teilmengen (≈100 Beispiele pro Benchmark)	Drastisch reduzierter Evaluationsaufwand bei geringer Schätzfehlerrate
LocalScore	Misst Inferenzgeschwindigkeit auf der jeweiligen Hardware (Prompt-Verarbeitung, Generierung, TTFT)	Vergleichbarkeit der Hardware-Performance über Konfigurationen hinweg

Tabelle 3.8: Übersicht der eingesetzten Benchmarks

3.9.1 MMLU

Der Massive Multitask Language Understanding (MMLU) Benchmark [He21a] umfasst Multiple-Choice-Fragen aus 57 Wissensgebieten, von Naturwissenschaften über Recht bis hin zu Geschichte. Er gilt als Standardmaß für allgemeines Sprachwissen und ermöglicht einen direkten Vergleich mit publizierten Referenzwerten. Aufgrund des Umfangs von mehreren tausend Fragen ist die Durchführungszeit auf dem Raspberry Pi 5 erheblich.

3.9.2 MATH500

MATH500 ist ein Subset des MATH-Benchmarks [He21b] und umfasst 500 mathematische Aufgaben aus verschiedenen Schwierigkeitsstufen und Teilgebieten. Er eignet sich besonders zur Beurteilung strukturierter Schlussfolgerungsfähigkeiten und ist für den Vergleich verschiedener Modellklassen (Basismodelle vs. Reasoning-Modelle) von besonderem Interesse.

3.9.3 TinyBenchmarks

TinyBenchmarks [Po24] stellt IRT[9]-basierte statistisch repräsentative Teilmengen etablierter Benchmarks bereit, die mit deutlich reduziertem Evaluationsaufwand vergleichbare Aussagen wie die Vollversionen liefern sollen. Für den Einsatz auf ressourcenbeschränkter Hardware ist dies besonders relevant: Die Reduktion des Evaluationsaufwands ermöglicht es, innerhalb des verfügbaren Zeitrahmens aussagekräftige Ergebnisse zu erzielen.

[9] Item Response Theory

Zunächst wurde direkt das `tinyBenchmarks`-Repository verwendet. Später erfolgte der Wechsel auf die in lm-evaluation-harness integrierte TinyBenchmarks-Version [Ga24], die eine nahtlose Kombination mit den übrigen Benchmark-Läufen erlaubt. Methodisch ergab sich dabei ein Zielkonflikt: Das lm-evaluation-harness empfiehlt für Chatmodelle die Verwendung des modellspezifischen Prompt-Formats und unterstützt dafür auch Multi-Turn-Few-Shot-Formatierung. Die TinyBenchmarks-Dokumentation empfiehlt hingegen, zusätzliche Chat-Formatierung nicht zu aktivieren, um möglichst unverzerrte Schätzwerte für den Vollbenchmark zu erhalten.

Für diese Arbeit wurde nicht die bestmögliche Approximation des vollständigen Benchmark-Scores priorisiert, sondern eine realitätsnähere Nutzung von Instruct- und Chatmodellen. Daher wurde die Evaluation primär mit aktivierter Chat-Formatierung durchgeführt. Diese Entscheidung spiegelt den vorgesehenen Anwendungskontext der Arbeit besser wider, da die untersuchten Systeme nicht als rohe Basis-Modelle, sondern als dialogorientierte Chatbots eingesetzt werden sollen. Die Auswirkungen der Formatierungswahl auf die Benchmarkwerte wurden ergänzend durch Vergleichsläufe mit und ohne Chat-Template untersucht und in den Ergebnisnotizen separat dokumentiert.

Prompt-Formatierung und TinyChatEngine

Die Frage der Prompt-Formatierung war insbesondere für TinyChatEngine relevant. Für eine kontrollierte Evaluation wurde die TinyChatEngine-API so erweitert, dass für LLaMA-3 zwei unterschiedliche Prompt-Protokolle explizit unterstützt werden:

- **historisches Format**: kompatibel zum bereits vorhandenen Verhalten der Engine.
- **natives LLaMA-3-Instruct-Format**: entsprechend dem empfohlenen Template mit Special-Tokens.

Damit konnten TinyBenchmarks- und MMLU-Läufe sowohl im historischen Zustand als auch mit nativer LLaMA-3-Formatierung durchgeführt werden. Diese Gegenüberstellung ist methodisch relevant, weil sie den Einfluss des Prompt-Formats von anderen Faktoren wie Quantisierung, Inferenz-Backend und Hardware trennt.

3.9.4 LocalScore

LocalScore ist ein quelloffenes Benchmarking-Tool des Mozilla-Builders-Projekts, das ausschließlich die Inferenzgeschwindigkeit von LLMs auf der jeweiligen Hardware misst, unabhängig von der inhaltlichen Modellgüte. Erfasst werden drei Metriken: Prompt-Verarbeitungsgeschwindigkeit (Tokens/s), Token-Generierungsgeschwindigkeit (Tokens/s) sowie TTFT (ms). Diese werden zu einem einheitlichen Score zusammengefasst. Ein Wert von 1 000 gilt als exzellent, 250 als noch akzeptabel, unter 100 ist mit spürbaren Einschränkungen in der Nutzererfahrung zu rechnen.

Die Testsuite umfasst neun definierte Prompt-/Output-Kombinationen, die typische Anwendungsfälle abdecken, von kurzen Klassifikationsanfragen (1024 Prompt-Token, 16 Output-Token) über RAG-Szenarien (4096/256) bis hin zu langer Codegenerierung (1280/3072). Intern setzt LocalScore auf Llamafile, läuft damit unabhängig von den in dieser Arbeit eingesetzten Backends und liefert eine portable Vergleichsbasis.

4 Ergebnisse

Dieses Kapitel präsentiert die Ergebnisse der durchgeführten Messungen und Benchmarks. Die Evaluation umfasst zwei primäre Inferenz-Backends – TinyChatEngine 1 (AWQ INT4) und llama.cpp bzw. llamafile (GGUF) – sowie mehrere Modellklassen und Quantisierungsstufen, deren Bereitstellung in Abschnitt 3.3 beschrieben ist. Als primäres Inferenzsystem diente der Raspberry Pi 5 mit 16 GB RAM. Ergänzende Untersuchungen wurden auf dem Raspberry Pi 5 mit 8 GB RAM durchgeführt. Die Qualitätsbenchmarks MMLU und MATH500 wurden aufgrund des erheblichen Laufzeitaufwands auf einem Desktop-System ausgeführt. Die gemessenen Accuracy-Werte sind plattformunabhängig und direkt mit den übrigen Konfigurationen vergleichbar. LocalScore nimmt eine Sonderrolle ein: Es basiert ausschließlich auf llamafile 0.9.2 und misst die Hardware-Performance unabhängig vom in dieser Arbeit entwickelten Evaluationsstack. Die in dieser Arbeit ermittelten Raspberry-Pi-5-Ergebnisse wurden, soweit sie dort noch fehlten, an `localscore.ai` übermittelt und sind damit zusammen mit den Messungen anderer Hardware-Setups verfügbar und vergleichbar. Neben den primären Untersuchungsmodellen LLaMA-3-8B-Instruct und LLaMA-2-7B-Chat wurden Llama 3.2 1B, Gemma-3-4B-IT und Qwen 2.5 14B als Vergleichspunkte für kleinere bzw. größere Modellklassen herangezogen. Für die LocalScore- und Plattformvergleiche wurde zusätzlich Llama 3.1 8B eingesetzt, da diese Messreihe an die veröffentlichten Vergleichswerte anschließt. Die daraus abgeleiteten Aussagen zur Inferenzgeschwindigkeit beziehen sich daher auf die 8B-Klasse. Qualitätsaussagen werden hingegen modellgenau ausgewiesen.

4.1 Inferenzgeschwindigkeit und Time to First Token

4.1.1 Modellladezeit in Abhängigkeit vom Speichermedium

Neben der eigentlichen Inferenzgeschwindigkeit beeinflusst auch die Ladezeit des Modells die praktische Nutzbarkeit eines lokalen LLM-Systems. Insbesondere bei wiederholten Benchmark-Läufen, Modellwechseln oder Server-Neustarts ist relevant, wie schnell die Gewichte aus dem Speichermedium in den Arbeitsspeicher übernommen werden können. Zur

Einordnung dieses Effekts wurde TinyChatEngine auf demselben Raspberry Pi 5 sowohl mit einer NVMe-SSD als auch mit einer SD-Karte als Modellspeicher betrieben.

Tabelle 4.1 zeigt die gemessenen Ladezeiten für zwei repräsentative AWQ-Modelle. Als Massenspeicher kamen eine WD Black NVMe-SSD sowie eine Amazon Basics SD-Karte zum Einsatz.

Modell	**Medium**	**Ladezeit (s)**	**Speedup ggü. SD**
LLaMA-3-8B-Instruct	WD Black NVMe-SSD	8,1	9,2×
LLaMA-3-8B-Instruct	Amazon Basics SD-Karte	74,3	1,0×
LLaMA-2-7B-Chat	WD Black NVMe-SSD	5,8	8,6×
LLaMA-2-7B-Chat	Amazon Basics SD-Karte	49,9	1,0×

Tabelle 4.1: Vergleich der Modellladezeit zwischen NVMe-SSD und SD-Karte auf dem Raspberry Pi 5

Die Ergebnisse zeigen einen deutlichen Vorteil der NVMe-SSD: Die Ladezeit sinkt je nach Modell um den Faktor 8,6 bis 9,2. Dieser Effekt verbessert nicht die Token-Generierungsrate selbst, reduziert jedoch die Wartezeit bis zur Einsatzbereitschaft des Systems erheblich und beschleunigt experimentelle Vergleichsläufe mit häufigen Modellwechseln deutlich.

4.1.2 TinyChatEngine vs. llama.cpp: Direktvergleich LLaMA-3-8B

Ein für die Forschungsfrage RQ2 zentrales Teilergebnis lässt sich bereits aus den vorliegenden Rohdaten ableiten. Für das Modell LLaMA-3-8B-Instruct auf dem Raspberry Pi 5 16 GB liegen Laufzeitmessungen beider Backends unter vergleichbaren Testbedingungen vor.

Die llamafile-Referenzmessung im LocalScore-Testprotokoll (Q4_K_M) ergibt für das 8B-Modell eine mittlere Token-Generierungsrate von 2,08 tok/s. TinyChatEngine 1 (AWQ INT4, W4A8) wurde denselben neun Testfällen unterzogen. Tabelle 4.2 zeigt die Ergebnisse pro Testfall.

Abbildung 4.1 visualisiert die Token-Generierungsrate derselben Messreihe in Abhängigkeit von der gesamten Kontextgröße. Als Kontextgröße wird die Summe aus Prompt- und Ausgabe-Tokens des jeweiligen Testfalls verwendet. Bei mehrfach auftretenden Kontextgrößen wurde der Mittelwert gebildet.

Testfall	TinyChatEngine (AWQ)		llamafile (Q4_K_M)	
	PP (tok/s)	TG (tok/s)	PP (tok/s)	TG (tok/s)
pp1024+tg16	2,01	1,32	8,01	2,21
pp4096+tg256	1,37	0,54	6,88	1,83
pp2048+tg256	1,75	0,85	7,61	2,06
pp2048+tg768	1,75	0,79	7,60	2,02
pp1024+tg1024	2,01	1,03	8,05	2,10
pp1280+tg3072	1,94	0,72	7,99	1,94
pp384+tg1152	2,21	1,29	8,24	2,17
pp64+tg1024	2,32	1,55	8,49	2,22
pp16+tg1536	2,32	1,40	7,75	2,19

Tabelle 4.2: Prompt-Processing- und Token-Generierungsrate: TinyChatEngine 1 (AWQ INT4) vs. llamafile (Q4_K_M), jeweils LLaMA-3-8B-Instruct auf Raspberry Pi 5 16 GB

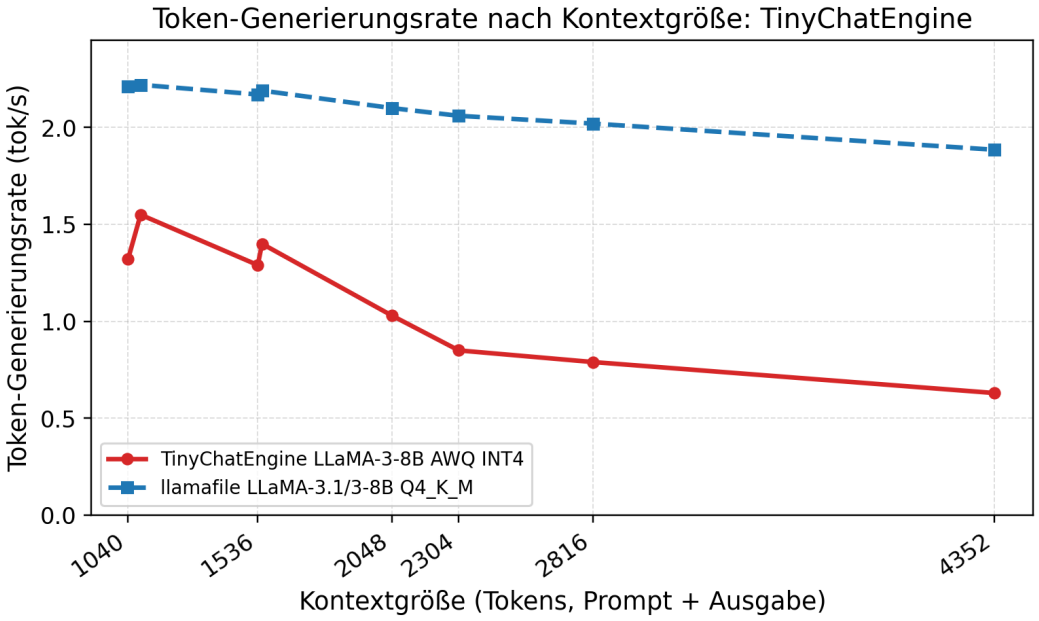

Abbildung 4.1: Token-Generierungsrate von TinyChatEngine 1 und llamafile in Abhängigkeit von der gesamten Kontextgröße auf dem Raspberry Pi 5 16 GB

TinyChatEngine erzielt in allen neun Testfällen niedrigere Raten als llamafile. Die Token-Generierungsrate liegt bei TinyChatEngine je nach Testfall zwischen 0,54 und 1,55 tok/s, während llamafile 1,83 bis 2,22 tok/s erreicht – ein konsistenter Faktor von etwa 1,5 bis 3,4 zugunsten von llamafile. Beim Prompt-Processing ist der Unterschied noch ausgeprägter: llamafile verarbeitet Prompt-Token rund drei- bis viermal schneller als TinyChatEngine. Beide Backends liegen unterhalb des RQ1-Schwellenwerts von 3 tok/s für interaktive Nutzung.

4.1.3 Quantisierungseinfluss auf Inferenzgeschwindigkeit

Der Einfluss der Quantisierungsstufe auf Token-Generierungsrate, Prompt-Processing und Time-to-First-Token wurde mit llamafile 0.9.2 (llama.cpp-Backend) auf dem Raspberry Pi 5 16 GB gemessen. Als Modell diente durchgängig LLaMA-3-8B-Instruct. Die Messungen folgen dem LocalScore-Protokoll (neun Testfälle, vgl. Abschnitt 3.9.4). Tabelle 4.3 fasst die gemittelten Metriken zusammen.

Quantisierung	TG (tok/s)	PP (tok/s)	TTFT (s)
bf16	0,64	4,30	317,3
Q4_K_M	2,33	8,41	166,2
Q5_K_M	2,04	7,23	191,5
Q6_K	1,77	6,18	223,4
Q8_0	1,27	9,75	146,5

Tabelle 4.3: Inferenzgeschwindigkeit von LLaMA-3-8B-Instruct nach Quantisierungsstufe (llamafile 0.9.2, Raspberry Pi 5 16 GB). TG = Token-Generierungsrate, PP = Prompt-Processing-Rate, TTFT = Time-to-First-Token (Mittelwert über alle Testfälle).

Die Ergebnisse zeigen einen klaren Trade-off zwischen Kompressionsgrad und Durchsatz. Q4_K_M erreicht mit 2,33 tok/s die höchste Token-Generierungsrate und unterstreicht damit den praktischen Wert aggressiver Quantisierung auf ressourcenbeschränkter Hardware. Q8_0 weist trotz geringerer TG (1,27 tok/s) die kürzeste TTFT (146,5 s) auf, was auf eine effizientere Verarbeitung langer Prefill-Sequenzen bei höherer Bitbreite hindeutet. Das bf16-Vollformat ist mit 0,64 tok/s und einer TTFT von über fünf Minuten für interaktive Anwendungen auf dem Raspberry Pi 5 praktisch nicht nutzbar. Keine der getesteten Konfigurationen erreicht den für flüssige Konversation angesetzten Schwellenwert von 3 tok/s (vgl. RQ1).

4.1.4 Ergänzende Messung: Gemma-3-4B-IT

Als zusätzlicher Vergleichspunkt für den Übergangsbereich zwischen kleinen interaktiven Modellen und 8B-Modellen wurde Gemma-3-4B-IT in der Quantisierung Q4_K_M auf dem Raspberry Pi 5 16 GB gemessen. Tabelle 4.4 zeigt die gemessenen Durchsatzwerte.

Modell	Quantisierung	TG (tok/s)	PP (tok/s)
Gemma-3-4B-IT	Q4_K_M	4,15	19,24

Tabelle 4.4: Inferenzgeschwindigkeit von Gemma-3-4B-IT auf dem Raspberry Pi 5 16 GB

Mit 4,15 tok/s überschreitet Gemma-3-4B-IT den in RQ1 definierten Schwellenwert von 3 tok/s. Die Messung belegt damit, dass nicht nur 1B-Modelle, sondern auch ein Modell der 3–5B-Klasse auf dem Raspberry Pi 5 interaktiv nutzbare Token-Raten erreichen kann. Gleichzeitig liegt die Prompt-Processing-Rate mit 19,24 tok/s deutlich über den gemessenen

8B-Konfigurationen, aber weiterhin erheblich unter dem 1B-Modell aus dem LocalScore-Vergleich.

4.1.5 Speculative Decoding

Speculative Decoding (vgl. Abschnitt 3.7) liefert auf dem Raspberry Pi 5 trotz hoher Akzeptanzraten keinen Durchsatzgewinn. Tabelle 4.5 zeigt die effektive Akzeptanzrate und den normierten Speedup-Faktor (Baseline-ms/Token ÷ Speculative-ms/Token, Werte über 1,0 bedeuten Beschleunigung) für alle 18 Konfigurationen aus einem vollständigen Benchmark über drei Prompt-Längen (*short*: 64 Wörter, *medium*: 256 Wörter, *long*: 1024 Wörter) und drei Generierungsbudgets.

Prompt-Länge	max_tokens	Akzeptanzrate	Speedup-Faktor	ign. Drafts
short	64	0,902	1,056	14
short	512	0,879	0,999	79
short	4096	0,879	1,005	79
medium	64	0,844	0,942	28
medium	512	0,832	0,893	125
medium	4096	0,832	0,895	125
long	64	0,725	0,894	16
long	512	0,760	0,887	111
long	4096	0,747	0,985*	146

Tabelle 4.5: Speculative Decoding auf dem Raspberry Pi 5: Akzeptanzrate, Speedup-Faktor und ignorierte Draft-Schritte. Draft-Modell: LLaMA 3.2 1B Instruct Q4_K_M. Hauptmodell: LLaMA 3.1 8B Instruct Q4_K_M. `draft_max=16`, `draft_min=2`, `draft_p_min=0.9`. ign. Drafts: Generierungsschritte, in denen das Draft-Modell weniger als `draft_min=2` Token vorgeschlagen hat. * Baseline und Speculative erzeugten in dieser Konfiguration unterschiedlich lange Antworten (1472 vs. 636 Token), weshalb die Vergleichbarkeit eingeschränkt ist.

Die Wahl des Prompt-Korpus war für diese Evaluation methodisch relevant. Benchmarks mit fixen Generierungslängen wie LocalScore greifen typischerweise auf synthetische oder stark repetitive Texte zurück. Ein Draft-Modell kann dort Token des Hauptmodells besonders zuverlässig vorhersagen, was die gemessene Akzeptanzrate gegenüber dem realen Einsatz systematisch überschätzen würde. Daher wurden natürlichsprachliche Anfragen aus dem OpenAssistant/oasst1-Datensatz verwendet, die einen realistischeren Anwendungsfall abbilden. Dies hat eine direkte Konsequenz für die Vergleichbarkeit: Bei inhaltlich abgeschlossenen Antworten beendet das Modell die Generierung vor Erreichen von `max_tokens` (`finish_reason=stop`), sodass Baseline und Speculative unterschiedlich lange Ausgaben erzeugen können. Der ausgeprägteste Fall ist *long*/4096, bei dem die Antwortlängen um den Faktor 2,3 differieren (1472 vs. 636 Token, vgl. Fußnote in Tabelle 4.5). Konfigurationen, bei denen das Tokenbudget vollständig ausgeschöpft wurde (`finish_reason=length`), sind von diesem Effekt nicht betroffen und direkt vergleichbar.

Nur in der Konfiguration *short*/64 ist ein geringfügiger Vorteil von 5,6 % erkennbar. Alle übrigen Konfigurationen zeigen keinen Speedup oder sind messbar langsamer. Bemerkenswert ist dabei, dass die Akzeptanzraten zwischen 0,73 und 0,90 liegen – Werte, die auf GPU-Systemen üblicherweise einen deutlichen Durchsatzgewinn zur Folge haben. Dass sich dieser Effekt auf dem Raspberry Pi 5 nicht einstellt, ist auf die in Abschnitt 2.1.2 beschriebene Speicherbandbreitenlimitierung zurückzuführen: Das Laden der Gewichte dominiert den Rechenaufwand, nicht die eigentliche Matrix-Multiplikation. Speculative Decoding verringert zwar die Anzahl der Forward-Pässe des Hauptmodells, kann den zusätzlichen Overhead durch das Laden der Draft-Modell-Gewichte auf dieser Hardware jedoch nicht ausgleichen. Erschwerend kommt hinzu, dass das Draft-Modell in vielen Generierungsschritten die Mindestanzahl von 2 Draft-Token nicht erreichte (Spalte *ign. Drafts*): In langen Läufen traten bis zu 146 solcher Schritte auf, sodass ein erheblicher Anteil der Generierungsschritte ohne spekulativen Pfad ausgeführt wurde. In den durchgeführten Tests konnte Speculative Decoding auf dem Raspberry Pi 5 keinen Durchsatzgewinn erzielen.

4.2 Speicherauslastung

Die Speicherauslastung ist auf dem Raspberry Pi 5 ein kritischer Faktor, da Modellgewichte, KV-Cache, Aktivierungspuffer und Betriebssystem-Overhead gemeinsam aus einem einzigen Arbeitsspeicherpool gespeist werden.

4.2.1 Raspberry Pi 5 mit 8 GB RAM

Der Betrieb eines AWQ-quantisierten 8B-Modells auf dem Raspberry Pi 5 mit 8 GB RAM ist grundsätzlich möglich, jedoch nicht empfehlenswert. Wie in Abschnitt 3.6.2 beschrieben, beanspruchen Modellgewichte, KV-Cache und laufende Systemdienste den verfügbaren Speicher nahezu vollständig. Der verbleibende Headroom für das Betriebssystem und Hintergrundprozesse ist so gering, dass ohne Swap-Speicher keine dauerhafte Systemstabilität gewährleistet werden kann. Sporadische OOM-Ereignisse und erzwungene Prozessabbrüche sind die Folge. Darüber hinaus ist das nutzbare Kontextfenster stark eingeschränkt: Die Kontextlänge muss auf ein Minimum reduziert werden, um Speicherüberläufe im KV-Cache zu vermeiden. Für reproduzierbare Benchmark-Läufe und produktiven Einsatz ist das 8-GB-System daher nicht geeignet.

4.2.2 Raspberry Pi 5 mit 16 GB RAM

Auf dem Raspberry Pi 5 mit 16 GB RAM ergibt sich ein grundlegend anderes Bild. Alle getesteten Konfigurationen konnten ohne Swap-Nutzung geladen und stabil betrieben werden. Für TinyChatEngineAPI war das verfügbare Kontextfenster mit der in Abschnitt 3.6.4 beschriebenen angepassten Sequenzgrenze auch für die Few-Shot-Benchmark-Läufe des lm-evaluation-harness ausreichend, ohne dass es zu speicherbedingten Einschränkungen kam. Die llama.cpp-Konfigurationen liefen ebenfalls stabil. Bis zu 14B-Parameter-Modelle (Qwen 2.5 14B, Q4_K_M) wurden ohne Swap betrieben. Der zugehörige LocalScore-Testlauf erreichte Sequenzlängen von bis zu 4352 Tokens und verlief ohne Speicherfehler.

4.3 Leistungsaufnahme und Energieeffizienz

Die Leistungsaufnahme des Raspberry Pi 5 wurde über die in Abschnitt 3.4 beschriebene externe Messinfrastruktur (ESP32 + 2× INA219) erfasst und in InfluxDB persistiert. Die zeitliche Auflösung der Messung erlaubt die Unterscheidung charakteristischer Phasen eines Inferenzlaufs anhand des Leistungsprofils: Modell-Laden, Prefill-Berechnung und Token-Generierung zeigen jeweils unterschiedliche Lastcharakteristika. Abbildung A.3 im Anhang illustriert einen typischen Verlauf.

Ruhezustand und Spitzenlast Tabelle 4.6 fasst die gemessenen Leistungswerte im Ruhezustand und unter Spitzenlast zusammen.

Betriebszustand	RPi5 16 GB (Rev 1.1)	RPi5 8 GB (Rev 1.0)
Idle (Ruhezustand)	≈2,5 W	≈3,0 W
Peak (Spitzenlast)	11,4 W	13,8 W

Tabelle 4.6: Leistungsaufnahme der beiden Raspberry-Pi-5-Systeme im Ruhezustand und unter Spitzenlast

Rev 1.1 (16 GB) ist in beiden Betriebszuständen sparsamer als Rev 1.0 (8 GB): Der Idle-Verbrauch liegt um 0,5 W niedriger, die Spitzenlast um 2,4 W. Dieser Befund ist bemerkenswert, da der größere Arbeitsspeicher des 16-GB-Modells einen höheren Grundverbrauch erwarten ließe. Als wahrscheinliche Ursache kommt eine verbesserte Siliziumrevision des BCM2712 infrage. Einschränkend ist anzumerken, dass die beiden Systeme nicht vollständig identisch ausgestattet sind: Kühlkörper, NVMe-SSDs und verwendete Steckverbinder unterscheiden sich, sodass ein Teil der Differenz auf diese Faktoren zurückzuführen sein kann.

Mittlere Leistungsaufnahme während der Inferenz Die mittlere Leistungsaufnahme während der Inferenz hängt wesentlich von der Anfragedauer und der Kontextlänge ab. Bei kurz-

en Requests, bei denen die CPU über die gesamte Laufzeit nahezu vollständig ausgelastet ist, wurden auf dem RPi5 16 GB im Mittel ≈8,5 W gemessen. Bei langen Antworten mit mehr als 2000 generierten Token wächst der KV-Cache auf einen Umfang, bei dem die Speicherbandbreite zur dominierenden Engstelle wird. Die CPU-Auslastung sinkt in diesem Regime, und die mittlere Leistungsaufnahme reduziert sich entsprechend auf ≈6,5 W.

Aus diesen Messwerten lässt sich für llama.cpp (Q4_K_M) ein näherungsweiser Energieverbrauch pro generiertem Token ableiten. Die LocalScore-Referenzmessung mit Llama 3.1 8B ergibt eine mittlere Token-Generierungsrate von 2,08 tok/s. Bei langen Antworten über 2000 Token sinkt sie speicherbandbreitenbedingt auf ≈1,25 tok/s:

- **Kurze Requests** (≈8,5 W, 2,08 tok/s): $\frac{8{,}5\ \text{W}}{2{,}08\ \text{tok/s}} \approx 4{,}1\ \text{J/tok}$
- **Lange Requests** (≈6,5 W, 1,25 tok/s): $\frac{6{,}5\ \text{W}}{1{,}25\ \text{tok/s}} \approx 5{,}2\ \text{J/tok}$

Obwohl die Leistungsaufnahme bei langen Anfragen sinkt, fällt die Token-Rate noch stärker ab, sodass der Energieverbrauch pro Token bei langen Kontexten höher ausfällt als bei kurzen. Diese Werte sind als Näherung zu verstehen, da die Leistungsaufnahme nicht für jede Modell- und Quantisierungskonfiguration separat erfasst wurde und die Messpunkte unterschiedliche Kontextlängen abdecken.

Plattformvergleich: Energieeffizienz Tabelle 4.7 stellt den Raspberry Pi 5 einer RTX 3090-basierten Desktop-Plattform gegenüber. Beide Plattformen wurden im Rahmen dieser Arbeit vermessen. Als Modell dient jeweils LLaMA-3.1-8B-Instruct (Q4_K_M). Der GPU-Verbrauch der RTX 3090 wurde über die NVIDIA-Treibertelemetrie ausgelesen und lag bei ≈340 W. Der angegebene Gesamtsystemwert von ≈390 W ist keine direkte Messung an der Steckdose, sondern eine Abschätzung auf Basis des GPU-Werts zuzüglich eines angenommenen Restsystemverbrauchs der x86-Plattform von ≈50 W.

Plattform	PP (tok/s)	TG (tok/s)	Leistung (W)	J/tok
RPi5 16 GB, kurze Req.	7,85	2,08	≈8,5	≈4,1
RPi5 16 GB, lange Req.	—	1,25	≈6,5	≈5,2
RTX 3090 (GPU only)	2059,71	61,78	≈340	≈5,5
RTX 3090 (Gesamtsystem)	2059,71	61,78	≈390	≈6,3

Tabelle 4.7: Plattformvergleich der Energieeffizienz: RPi5 16 GB vs. RTX 3090 (LLaMA-3.1-8B-Instruct Q4_K_M, eigene Messungen)

In den hier verglichenen Messpunkten liegt der RPi5 bei kurzen Anfragen mit ≈4,1 J/tok unter den für die RTX 3090 ausgewiesenen Werten – sowohl gegenüber dem ausgelesenen GPU-Verbrauch (5,5 J/tok) als auch gegenüber dem abgeschätzten Gesamtsystemwert (6,3 J/tok). Dieser auf den ersten Blick überraschende Befund ist nicht als generelle Überlegenheit der CPU-Plattform zu verstehen, sondern als Folge der konkreten Messbedingungen:

Die GPU erreicht beim 8B-Modell nur einen Bruchteil ihrer theoretischen Rechenkapazität, sodass die hohe absolute Leistungsaufnahme nicht vollständig in Durchsatz umgesetzt wird. Bei langen Kontexten kehrt sich das Verhältnis um: Der RPi5 (5,2 J/tok) liegt dann auf dem Niveau der GPU-only-Messung. Der robustere Unterschied liegt damit weniger in einer pauschalen Energieeffizienz pro Token als in der absoluten Leistungsaufnahme – 8,5 W gegenüber einem abgeschätzten RTX-3090-Gesamtsystemwert von 390 W – die den RPi5 für dauerhaften Edge-Betrieb ohne aktive Kühlung und mit einfacher Stromversorgung qualifiziert.

CPU-Auslastung im Backend-Vergleich Die unterschiedlichen Durchsatzwerte beider Backends spiegeln sich direkt im CPU-Auslastungsprofil wider. Bei llama.cpp hält die CPU-Auslastung während der Prefill-Phase nahezu dauerhaft bei 100 % und bleibt auch in der Decode-Phase über weite Strecken hoch. Erst bei sehr langen Generierungsläufen mit großem KV-Cache fällt die Auslastung langsam ab, da die Speicherbandbreite zunehmend limitiert. TinyChatEngine zeigt in beiden Phasen eine deutlich geringere CPU-Auslastung. Im Prefill-Segment ist dies die unmittelbare Ursache für die in Tabelle 4.2 dokumentierte bis zu viermal niedrigere Prompt-Processing-Rate. In der Decode-Phase setzt sich der Auslastungsunterschied fort, konsistent mit den geringeren Token-Generierungsraten. Da TinyChatEngine bei niedrigerer CPU-Last auch eine geringere absolute Leistungsaufnahme aufweist, gleichzeitig aber weniger Token pro Sekunde erzeugt, lässt sich aus den vorliegenden Messungen kein eindeutiger Vorteil bezüglich des Energieverbrauchs pro Token ableiten.

4.4 Modellgüte

4.4.1 MMLU

Tabelle 4.8 zeigt die MMLU-Gesamtaccuracy der untersuchten Backends und Modelle. Die vollständige MMLU-Suite umfasst 14 042 Multiple-Choice-Fragen aus 57 Wissensgebieten und erfordert auf dem Raspberry Pi 5 mit realitätsnahen Inferenzgeschwindigkeiten von 1–3 Tokens/s mehrere Tage Laufzeit. Die Evaluationen wurden daher auf einem Desktop-System durchgeführt. Die gemessene Accuracy ist plattformunabhängig: Modellgewichte und Antwortverteilung sind identisch mit den übrigen Konfigurationen. Die Bewertung erfolgte über den Logprob-Pfad des LLMProxy. Für TinyChatEngine wurde die in Abschnitt 3.6.5 beschriebene eigene Logprob-Implementierung verwendet.

Die Evaluationsmethodik orientiert sich an den MMLU-Notebooks von Sebastian Raschka [Ra25]. Diese Vorlage wurde gewählt, um die Vergleichbarkeit mit der parallel entstandenen Arbeit von Lukas Heiming [He26] zu gewährleisten, die denselben Ansatz verwendet. Zunächst wurde die MMLU-Evaluation mit einem eigenständig entwickelten Notebook

durchgeführt. Im weiteren Verlauf erfolgte aus Gründen der Reproduzierbarkeit und Vergleichbarkeit der Wechsel auf die Raschka-Vorlage.

Backend	Modell / Quantisierung	Accuracy	Korrekt / Gesamt
TinyChatEngine	LLaMA-3-8B-Instruct (AWQ INT4)	57,27 %	8 042 / 14 042
llama.cpp / Ollama	LLaMA-3-8B-Instruct Q4_K_M	60,85 %	8 544 / 14 042
llama.cpp / Ollama	LLaMA-3-8B-Instruct Q8_0	62,39 %	8 761 / 14 042
llama.cpp / Ollama	LLaMA-3.2-3B-Instruct Q4_K_M	56,89 %	7 989 / 14 042

Tabelle 4.8: MMLU-Gesamtaccuracy: TinyChatEngine und llama.cpp/Ollama (14 042 Fragen, 57 Teilgebiete, Logprob-Scoring)

Das beste Ergebnis erzielt LLaMA-3-8B-Instruct in der Q8_0-Quantisierung mit 62,39 %. Die Q4_K_M-Variante liegt mit 60,85 % nur 1,5 Prozentpunkte darunter. TinyChatEngine (AWQ INT4) erreicht 57,27 % und liegt damit 3,6 Prozentpunkte unter dem llama.cpp-Q4_K_M-Ergebnis. Dieser Befund deckt sich mit der in Tabelle 4.10 gezeigten Tendenz aus den TinyBenchmarks. Die Ursache ist dabei nicht ausschließlich im Quantisierungsformat zu suchen, sondern auch in Unterschieden der Prompt-Pipeline, wie in Abschnitt 4.4.4 diskutiert. Der veröffentlichte Referenzwert für LLaMA-3-8B-Instruct beträgt 68,4 % [Ll24]. Die Abweichung unserer Messungen von rund 6–11 Prozentpunkten ist unter anderem auch auf die abweichende Evaluierungsmethodik zurückzuführen.

4.4.2 MATH500

Tabelle 4.9 zeigt die MATH500-Accuracy der untersuchten Konfigurationen. Auch hier basiert die Evaluationsmethodik auf den Notebooks von Raschka [Ra25], wiederum zur Sicherstellung der Vergleichbarkeit mit der Arbeit von Heiming [He26]. Im Unterschied zu MMLU erfordert MATH500 die Generierung einer freien Antwort. Die Auswertung extrahiert dabei das letzte `\boxed{}`-Ergebnis aus der Modellausgabe, normalisiert es und prüft mathematische Äquivalenz per SymPy. Fehlt dieses Format, greift ein schwächerer Fallback, Modelle ohne konsistente Antwortformatierung erzielen daher systematisch niedrige Trefferquoten, auch wenn die Rechnung inhaltlich stimmt.

Die Ergebnisse zeigen ein breites Spektrum. Gemma-3-4B erzielt trotz deutlich kleinerer Modellgröße die höchste gemessene Accuracy (64,6 %) und deutet damit auf eine stärkere Eignung dieser Modellfamilie für die verwendeten mathematischen Aufgaben hin. LLaMA-3.1-8B-Instruct erreicht 35,0 %. Der Generationsunterschied zwischen LLaMA-3 und LLaMA-3.1 kann einen Teil der Lücke zu den LLaMA-3-8B-Ergebnissen (3,8 und 8,8 %) erklären. Für LLaMA-3-8B-Instruct liefern llama.cpp (3,8 %) und TinyChatEngine

Backend	Modell / Quantisierung	Accuracy	Plattform
TinyChatEngine	LLaMA-3-8B-Instruct (AWQ INT4)	8,8 %	RPi5 16 GB
llama.cpp	LLaMA-3-8B-Instruct Q4_K_M	3,8 %	Desktop (CPU)
llama.cpp	LLaMA-3.1-8B-Instruct Q4_0	35,0 %	Desktop (CPU)
llama.cpp	Gemma-3-4B-Instruct Q4_K_M	64,6 %	Desktop (GPU)

Tabelle 4.9: MATH500-Accuracy nach Backend und Modell

(8,8 %) deutlich niedrigere Werte. Beide Backends weisen dabei keine einheitliche Ausgabeformatierung der Endantwort auf, was bei striktem Zeichenkettenabgleich zu systematischen Fehlzuordnungen führen kann. MATH500 ist daher in dieser Arbeit vor allem als formatempfindlicher Hinweis auf mathematisches Reasoning zu verstehen. Für einen belastbaren absoluten Gütevergleich wären eine vereinheitlichte Antwortformatierung und eine Plattform- bzw. Backend-konsistente Wiederholung der Messungen erforderlich.

4.4.3 Qualitative Beobachtungen im Chat

Vor der eigentlichen Benchmark-Auswertung fiel bei interaktiven Tests mit TinyChatEngine auf, dass die AWQ-Variante das erwartete Gesprächsformat nicht immer stabil einhielt. Insbesondere bei LLaMA-3 kam es wiederholt dazu, dass das Modell nicht nur die Antwort des Assistenzsystems fortsetzte, sondern zusätzlich die nächste Nutzerrolle antizipierte und beide Gesprächsseiten in einer Ausgabe übernahm. Dieses Verhalten trat in den llama.cpp-Referenzläufen in dieser Form nicht auf und war für Chat-Anwendungen problematisch, da es die Konsistenz längerer Dialoge unmittelbar beeinträchtigt.

Die Beobachtung war zunächst qualitativ und entstand außerhalb der standardisierten Benchmark-Läufe. Sie war jedoch relevant genug, um die anschließende Evaluation gezielt auf den Einfluss der Prompt-Formatierung auszurichten: Wenn bereits die Gesprächsrollen nicht stabil eingehalten werden, ist plausibel, dass dies auch die Qualität formaler Benchmarks beeinflusst.

4.4.4 TinyBenchmarks

Ausgehend von den qualitativen Chat-Beobachtungen wurde die Hypothese formuliert, dass die Qualitätsunterschiede zwischen TinyChatEngine und llama.cpp nicht ausschließlich auf AWQ gegenüber GGUF zurückzuführen sind, sondern dass auch das Prompt-Protokoll einen eigenständigen Einfluss hat. Die Evaluation wurde daher mit zwei Schwerpunkten durchge-

führt: dem Einfluss der Prompt-Formatierung sowie dem Verlauf der Modellgüte über einige GGUF-Quantisierungsstufen.

Einfluss der Prompt-Formatierung Tabelle 4.10 zeigt den Einfluss der Prompt-Formatierung für `Q4_K_M` beider Primärmodelle. Bei llama.cpp werden zwei Modi verglichen: ohne Chat-Template sowie mit Chat-Template und aktiviertem `gen_prefix` (`" The answer is"`). Der `gen_prefix` war notwendig, um einen systematischen Messartefakt zu beheben: Ohne ihn bewertet lm-evaluation-harness die Antwortoptionen A–D an einer Tokenposition, an der das Instruct-Modell nahezu identische Wahrscheinlichkeiten für alle vier Optionen vergibt, was zu zufallsähnlichen MMLU-Rohwerten führt. Der `gen_prefix` öffnet einen Assistenten-Turn vor der bewerteten Position und behebt diesen Effekt.

Modell / Backend	Formatierung	MMLU raw	gpirt avg
LLaMA-3-8B / llama.cpp Q4_K_M	ohne Chat-Template	0,590	0,669
LLaMA-3-8B / llama.cpp Q4_K_M	mit Chat-Template	0,620	0,643
LLaMA-3-8B / TinyChatEngine	kein Template	0,520	0,637
LLaMA-3-8B / TinyChatEngine	Legacy-Format	0,570	0,649
LLaMA-3-8B / TinyChatEngine	Native Meta-Format	0,280	0,624
LLaMA-2-7B / llama.cpp Q4_K_M	ohne Chat-Template	0,460	0,533
LLaMA-2-7B / llama.cpp Q4_K_M	mit Chat-Template	0,320	0,486
LLaMA-2-7B / TinyChatEngine	kein Template	0,380	0,521
LLaMA-2-7B / TinyChatEngine	Legacy-Format	0,310	0,522
LLaMA-2-7B / TinyChatEngine	Native Format	0,260	0,456

Tabelle 4.10: Einfluss der Prompt-Formatierung auf TinyBenchmarks-Ergebnisse (llama.cpp jeweils Q4_K_M, TinyChatEngine AWQ-INT4)

Für LLaMA-3 liegen die `gpirt`-Werte beider llama.cpp-Modi im Bereich 0,643–0,669. Die Template-Wahl hat damit keinen dominierenden Einfluss. Die Chat-Template-Variante wird als Primärauswertung verwendet, da sie dem vorgesehenen Einsatzmodus der Instruct-Modelle entspricht.

Der direkteste Backend-Vergleich ergibt sich aus den Läufen ohne jegliche Formatierung: TinyChatEngine erreicht dort 0,637, llama.cpp 0,669. Dieser Vergleich schließt Prompt-

Formatierung als Einflussfaktor aus und zeigt, dass llama.cpp unter identischen Bedingungen höhere gpirt-Werte erzielt. Das Legacy-Format (0,649) liegt geringfügig über dem no-template-Wert. Der deutliche Einbruch im MMLU-Rohwert unter dem nativen Meta-Format (0,280) deutet darauf hin, dass MMLU-Aufgaben mit diesem Template in TinyChatEngine nicht erwartungsgemäß ausgewertet werden, während die übrigen Teilbenchmarks weniger betroffen sind.

Bei LLaMA-2 liefern kein Template und Legacy-Format für TinyChatEngine nahezu identische gpirt-Werte (0,521 und 0,522), was darauf hindeutet, dass das Legacy-Schema für LLaMA-2 in der Praxis keinen wesentlichen Einfluss auf die Ergebnisse hat. Unter Chat-Template-Bedingungen liegt TinyChatEngine Legacy (0,522) über llama.cpp Q4_K_M (0,486). Ohne Template kehrt sich das Verhältnis um (llama.cpp 0,533 vs. TinyChatEngine 0,521). Die Formatierungswahl beeinflusst den Vergleich zwischen den Backends also stärker als die Quantisierung selbst.

Einfluss der Quantisierungstiefe Tabelle 4.11 zeigt die Ergebnisse für beide Modelle über einige getestete Quantisierungsstufen (llama.cpp, Chat-Template mit gen_prefix). Als Referenz dient bf16. Δ gibt die absolute Abweichung im gpirt-Wert an.

Quantisierung	LLaMA-3-8B-Instruct		LLaMA-2-7B-Chat	
	gpirt	Δ	gpirt	Δ
bf16 (Referenz)	0,658	—	0,498	—
Q8_0	0,656	−0,002	0,494	−0,004
Q6_K	0,657	−0,001	0,499	+0,001
Q5_K_M	0,658	±0,000	0,493	−0,005
Q4_K_M	0,643	−0,015	0,486	−0,012
Q3_K_M	0,643	−0,015	0,466	−0,032
Q3_K_S	0,628	−0,030	0,478	−0,020
Q2_K	0,555	−0,103	0,458	−0,040
IQ3_S	0,293	−0,365	0,285	−0,213
TinyChatEngine (Legacy)	0,649	−0,009	0,522	+0,024
TinyChatEngine (Native)	0,624	−0,034	0,456	−0,042

Tabelle 4.11: TinyBenchmarks (gpirt avg) über einige Quantisierungsstufen: LLaMA-3-8B-Instruct und LLaMA-2-7B-Chat auf dem Raspberry Pi 5 (llama.cpp, Chat-Template mit gen_prefix, Δ gegenüber bf16)

Für LLaMA-3-8B-Instruct zeigt die Tabelle von bf16 bis Q4_K_M keine wesentliche Veränderung der gpirt-Werte. Angesichts der kleinen Stichprobengröße von TinyBenchmarks sind die Unterschiede in diesem Bereich nicht als inhaltlich bedeutsam zu werten. Ab Q3_K_S ist ein Rückgang erkennbar, Q2_K liegt klar unterhalb der höherwertigen Stufen. IQ3_S fällt mit einem gpirt-Wert von 0,293 auf Zufallsniveau und ist für den praktischen Einsatz nicht

geeignet. TinyChatEngine im Legacy-Format (0,649) liegt im Bereich der mittleren GGUF-Quantisierungsstufen. Das native Format (0,624) fällt sichtbar ab.

Bei LLaMA-2-7B-Chat liegen die `gpirt`-Werte insgesamt deutlich niedriger (bf16: 0,498 gegenüber 0,658 bei LLaMA-3). Das qualitative Bild ist ähnlich: Von `bf16` bis `Q4_K_M` keine wesentlichen Unterschiede, darunter zunehmender Verlust, `IQ3_S` erneut nicht funktionsfähig.

Insgesamt zeigen die Ergebnisse, dass TinyChatEngine nicht als isolierter AWQ-Gegenpol zu llama.cpp betrachtet werden sollte. Zwischen beiden Systemen unterscheiden sich nicht nur Quantisierung und Laufzeitverhalten, sondern auch die semantische Eingaberepräsentation.

4.5 LocalScore

Für den hardwarebezogenen Durchsatzvergleich wurden fünf Modellkonfigurationen in der Quantisierungsstufe Q4_K_M mit LocalScore gemessen: Llama 3.2 1B (tiny), DeepSeek R1 Distill Qwen 1.5B, Gemma-3-4B-IT, Llama 3.1 8B (small) und Qwen 2.5 14B (medium). Tabelle 4.12 zeigt die Ergebnisse zusammen mit ausgewählten Vergleichsplattformen. Die vollständigen LocalScore-Ausgaben aller vier Läufe sind in Abschnitt A.4 im Anhang abgebildet. Die 8B-LocalScore-Zeile ist damit nicht identisch mit den LLaMA-3-8B-Instruct-Qualitätsmessungen, liegt aber in derselben Modellgrößenklasse und wird ausschließlich für Hardware- und Durchsatzvergleiche herangezogen.

Der LocalScore sinkt mit wachsender Modellgröße stark: Das 1B-Modell erreicht 27 Punkte, Gemma-3-4B-IT erreicht 10 Punkte, das 8B-Modell 4 und das 14B-Modell 2. Mit Token-Generierungsraten von 11,06 tok/s bzw. 4,15 tok/s überschreiten das 1B-Modell und Gemma-3-4B-IT den in RQ1 gesetzten Schwellenwert von 3 tok/s für interaktive Nutzung. Das 8B-Modell (2,08 tok/s) und das 14B-Modell (1,12 tok/s) verfehlen ihn. Der Raspberry Pi 4 erreicht für das 1B-Modell lediglich 7 Punkte. Eine NVIDIA RTX 3090 erzielt für dasselbe 1B-Modell 5 612 Punkte und liegt damit um den Faktor ≈ 208 über dem Raspberry Pi 5. Für das 8B-Modell beträgt dieser Faktor noch ≈ 261 (1 044 vs. 4 Punkte). Die Ergebnisse bestätigen die in Abschnitt 2.7.1 diskutierte strukturelle Leistungslücke zwischen CPU- und GPU-basierter Inferenz.

Ein weiterer limitierender Faktor ist die TTFT, die mit der Prompt-Länge linear wächst. Abbildung 4.2 visualisiert diesen Zusammenhang für die auf dem Raspberry Pi 5 gemessenen Modellgrößen. Tabelle 4.13 ergänzt die zugehörigen Einzelwerte.

Plattform	Modell	LocalScore	TG (tok/s)	PP (tok/s)	TTFT (ms)
RPi4 4B (Cortex-A72)	Llama 3.2 1B Q4_K_M	7	2,60	7,23	204 971
RPi5 16GB (Cortex-A76)	Llama 3.2 1B Q4_K_M	27	11,06	53,96	28 595
RPi5 16GB (Cortex-A76)	Gemma-3-4B-IT Q4_K_M	10	4,15	19,24	71 611
RPi5 16GB (Cortex-A76)	Llama 3.1 8B Q4_K_M	4	2,08	7,85	178 826
RPi5 16GB (Cortex-A76)	Qwen 2.5 14B Q4_K_M	2	1,12	4,44	319 454
RTX 3090 (GPU, 24 GB)	Llama 3.2 1B Q4_K_M	5 612	354,56	14 630,76	93
RTX 3090 (GPU, 24 GB)	Llama 3.1 8B Q4_K_M	1 044	106,27	3 724,59	347
RTX 3090 (GPU, 24 GB)	Qwen 2.5 14B Q4_K_M	565	61,45	2 059,19	629

Tabelle 4.12: LocalScore-Ergebnisse und Plattformvergleich (llamafile 0.9.2, Q4_K_M)

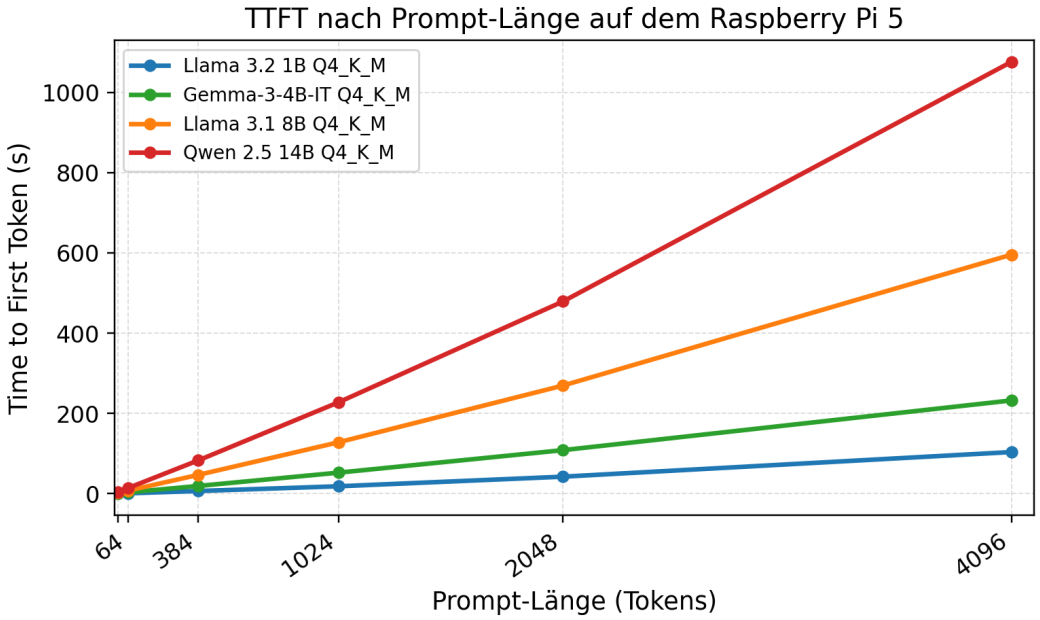

Abbildung 4.2: TTFT in Abhängigkeit von der Prompt-Länge für verschiedene Modellgrößen auf dem Raspberry Pi 5 16 GB (LocalScore, Q4_K_M, CPU-only)

Bereits bei einem Kontext von 1024 Tokens wartet der Nutzer beim 8B-Modell über zwei Minuten auf die erste Antwort. Beim 14B-Modell sind es fast vier Minuten. Bei 4096 Prompt-Tokens steigen diese Werte auf knapp zehn bzw. fast achtzehn Minuten – Latenzen, die einen interaktiven Chatbetrieb praktisch ausschließen. Auch Gemma-3-4B-IT liegt bei langen Prompts deutlich oberhalb unmittelbarer Antwortzeiten: Bei 1024 Prompt-Tokens beträgt die TTFT bereits rund 53 s, bei 4096 Prompt-Tokens knapp vier Minuten. Selbst das 1B-Modell, das in der Kurz-Kontext-Nutzung flüssig arbeitet, akkumuliert bei langen Eingaben merkliche Wartezeiten. Die Ergebnisse unterstreichen, dass auf dem Raspberry Pi 5

eine aggressive Modell- und Kontextlängenwahl keine optionale Optimierung, sondern eine Grundvoraussetzung für praxistauglichen Betrieb ist.

Prompt-Tokens	1B (s)	4B (s)	8B (s)	14B (s)
16	0,33	1,07	2,47	4,27
64	1,07	3,36	7,94	14,46
384	6,77	19,40	47,06	83,25
1024	18,59	52,55	128,27	227,93
2048	42,49	108,60	269,54	479,08
4096	103,88	232,54	595,84	1 076,40

Tabelle 4.13: TTFT in Sekunden nach Prompt-Länge (LocalScore, Q4_K_M, CPU-only)

Abbildung 4.3 zeigt ergänzend die Token-Generierungsrate in Abhängigkeit von der gesamten Kontextgröße. Als Kontextgröße wird dabei die Summe aus Prompt- und generierten Tokens aus der LocalScore-Spalte *tokens processed* verwendet. Bei mehrfach auftretenden Kontextgrößen wurde der Mittelwert der zugehörigen Testfälle gebildet.

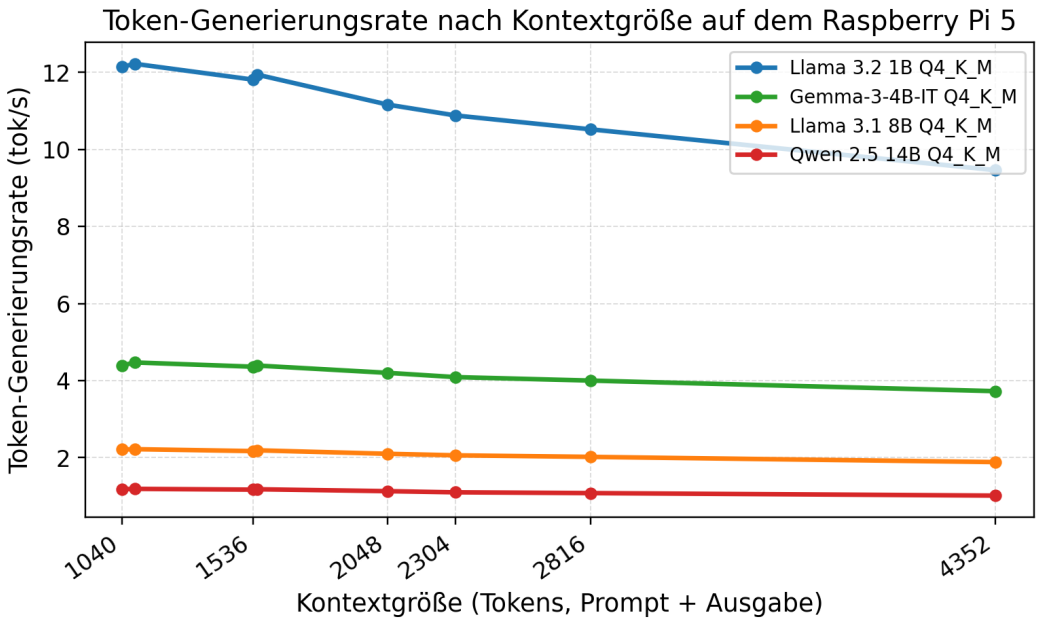

Abbildung 4.3: Token-Generierungsrate in Abhängigkeit von der gesamten Kontextgröße auf dem Raspberry Pi 5 16 GB (LocalScore, Q4_K_M, CPU-only)

5 Diskussion und Fazit

5.1 Diskussion der Ergebnisse

5.1.1 TinyChatEngine als Untersuchungsplattform

TinyChatEngine 1 ist in seiner Grundkonzeption eine in sich geschlossene Inferenzplattform ohne externe Laufzeitabhängigkeiten. Dieser Ansatz macht die Engine zu einem gut nachvollziehbaren Ausgangspunkt für eigene Erweiterungen: Der überschaubare Codeumfang und die direkte Abbildung von Modell, Inferenz und Sampling in C++ ermöglichen ein tieferes Verständnis der zugrundeliegenden Abläufe, als dies bei größeren Projekten wie llama.cpp möglich wäre.

In ihrer Ausgangskonfiguration fehlten jedoch wesentliche Voraussetzungen für eine automatisierte Evaluation. Es gab keine Möglichkeit, den Gesprächszustand zwischen Anfragen zurückzusetzen, Inferenzparameter zur Laufzeit anzupassen oder strukturierte Abfragen über eine Netzwerkschnittstelle zu stellen. Durch die im Rahmen dieser Arbeit entwickelte REST-API wurden diese Lücken gezielt geschlossen. Die resultierende TinyChatEngineAPI ermöglicht den automatisierten Betrieb von Benchmarks mit mehreren tausend Einträgen (etwa MMLU mit über 14 000 Fragen) sowie die nahtlose Integration in OpenAI-kompatible Softwarelösungen wie das lm-evaluation-harness.

Rückblickend wäre es effizienter gewesen, von Beginn an eine vollständig OpenAI-kompatible Schnittstelle zu entwickeln, anstatt zunächst eine proprietäre Lösung aufzubauen und diese nachträglich um einen kompatiblen Endpunkt zu ergänzen. Dieser Umweg verursachte zusätzlichen Entwicklungsaufwand, insbesondere bei der Erstellung von Evaluations-Notebooks. Die realisierte Infrastruktur schafft dennoch eine wertvolle Grundlage: Mit dem lm-evaluation-harness lassen sich nun beliebige Datensätze direkt gegen AWQ-quantisierte Modelle evaluieren. Für künftige Arbeiten erleichtert dies die aufgabenspezifische Untersuchung von AWQ-Modellen erheblich.

5.1.2 Vergleich TinyChatEngine und llama.cpp

Der Vergleich der beiden Inferenz-Backends fällt in nahezu allen Kategorien zugunsten von llama.cpp aus.

Funktionsumfang und Flexibilität llama.cpp unterstützt eine erheblich größere und aktuellere Modellauswahl, darunter auch neuere Architekturen wie Mixture-of-Experts-Modelle. Konfigurationsparameter lassen sich über Kommandozeilenargumente oder eine Konfigurationsdatei setzen, ohne dass ein Neubau der Binärdatei erforderlich ist. Bei TinyChatEngine 1 sind relevante Parameter teils über mehrere Quelldateien verteilt, was Anpassungen aufwändiger macht und einen Rebuild voraussetzt. Darüber hinaus bietet llama.cpp integrierte Unterstützung für Speculative Decoding und KV-Cache-Quantisierung, die in TinyChatEngine 1 nicht verfügbar sind. Das deutlich größere Entwicklerteam und die aktive Community sorgen für eine schnellere Unterstützung neuer Modellgenerationen und kontinuierliche Optimierungen.

Inferenzgeschwindigkeit Auf identischer Hardware konnte TinyChatEngine die Inferenzgeschwindigkeit von llama.cpp nicht erreichen. Die Token-Generierungsrate lag in allen neun LocalScore-Testfällen konsistent um den Faktor 1,5 bis 3,4 unter llamafile. Beim Prompt-Processing war der Unterschied mit Faktor 3 bis 4 noch ausgeprägter (vgl. Tabelle 4.2). Dies ist bemerkenswert, da TinyChatEngine durch plattformspezifische Kernel für ARM-SIMD und die AWQ-Gewichtsrepräsentation eigentlich Geschwindigkeitsvorteile versprechen sollte. Im aktuellen Zustand kann TinyChatEngine jedoch nicht mit den Optimierungen von llama.cpp mithalten.

Modellgüte Die Qualitätsbenchmarks zeigen ein differenzierteres Bild. MMLU-Accuracy und TinyBenchmarks-`gpirt`-Werte hängen stark von der Prompt-Formatierung ab, sodass ein unmittelbarer Vergleich nur unter kontrollierten Bedingungen aussagekräftig ist. Unter vergleichbaren Formatierungsvoraussetzungen erzielt llama.cpp jedoch durchgehend höhere Benchmark-Werte. Ein praktisch relevanter Qualitätsvorteil von AWQ im Chat-Gebrauch gegenüber äquivalenten GGUF-Quantisierungsstufen lässt sich aus den vorliegenden Ergebnissen nicht ableiten.

5.1.3 Praktische Einsatzfähigkeit auf dem Raspberry Pi 5

Plattformwahl Für den produktiven Einsatz von 8B-Modellen ist der Raspberry Pi 5 mit 16 GB RAM klar vorzuziehen. Der 8-GB-Variante fehlt der notwendige Headroom für einen stabilen Langzeitbetrieb. Insbesondere in Kombination mit Benchmark-Infrastruktur und Monitoring-Prozessen ist das Risiko von OOM-Ereignissen nicht vernachlässigbar. Hinzu kommt, dass das nutzbare Kontextfenster auf dem 8-GB-System für 7- und 8B-Modelle nicht für alle MMLU-Fragen mit Few-Shot-Beispielen ausreichte und damit einzelne Benchmark-Eingaben speicherbedingt nicht vollständig abgebildet werden konnten. Auf dem 16-GB-System konnten alle Benchmarks stabil und ohne speicherbedingte Einschränkungen durchgeführt werden.

Modellgröße, Interaktivität und Hintergrundverarbeitung Die Messungen zeigen eine ausgeprägte Abhängigkeit zwischen Modellgröße und Inferenzgeschwindigkeit, die je nach Einsatzszenario unterschiedlich zu gewichten ist. Das 1B-Modell (Llama 3.2 1B, Q4_K_M) erreicht mit rund 11 tok/s eine Token-Rate, die den Schwellenwert für interaktiven Dialogbetrieb (RQ1: ≥ 3 tok/s) deutlich überschreitet und damit flüssige Konversation ermöglicht. Das gilt für beide RPi5-Varianten, also auch bei nur 8 GB RAM. Die Modellgüte ist bei diesem Parameterumfang begrenzt, dennoch eignet sich das Modell für Aufgaben, bei denen das Ergebnis leicht überprüfbar ist oder Fehler keine kritischen Folgen haben: Anwendungsfälle, die zu komplex für regelbasierte Automatisierung sind, aber keine Präzision auf dem Niveau größerer Modelle erfordern. Die ergänzende Messung mit Gemma-3-4B-IT (Q4_K_M) bestätigt, dass auch ein Modell der 3–5B-Klasse den Interaktivitätsschwellenwert erreichen kann: Gemma erzielt 4,15 tok/s bei 19,24 tok/s Prompt-Processing. Damit bildet diese Modellklasse einen relevanten Zwischenbereich zwischen sehr schnellen, aber qualitativ begrenzten 1B-Modellen und langsameren 8B-Modellen.

Die gemessenen 8B-Konfigurationen unterschreiten den Interaktivitätsschwellenwert mit 2,08–2,33 tok/s (llamafile, Q4_K_M), während das 14B-Modell (Qwen 2.5 14B) im Mittel 1,12 tok/s erreicht. Für Echtzeitkonversation sind diese Raten nicht ausreichend. Als Hintergrundprozesse ohne Echtzeitanforderung sind 8B- und 14B-Modelle auf dem Raspberry Pi 5 16 GB hingegen vollständig praxistauglich. Aufgaben wie E-Mail-Klassifikation, Spam-Erkennung, Dokumenten-Routing oder die Vorqualifikation von Anfragen im First-Level-Support lassen sich asynchron verarbeiten, da Latenzen von wenigen Minuten pro Anfrage in solchen Szenarien problemlos akzeptabel sind. Für derartige Anwendungen, die inhaltlich zu anspruchsvoll für einfache Regelwerke, aber gut im Rahmen eines 8B-Modells lösbar sind, bietet der Raspberry Pi 5 16 GB ein kosteneffizientes lokales Setup ohne Abhängigkeit von Cloud-Diensten.

5.1.4 Methodische Erkenntnisse

Ein wesentlicher Querschnittsbefund dieser Arbeit liegt auf methodischer Ebene: Bereits kleine Anpassungen an Prompt-Aufbau, Few-Shot- bzw. Zero-Shot-Konfiguration, Auswertungslogik oder Inferenzparametern können die gemessenen Benchmark-Ergebnisse teils erheblich beeinflussen. Dies zeigt sich besonders deutlich beim Vergleich der Prompt-Formate für TinyChatEngine (vgl. Tabelle 4.10): Der MMLU-Rohwert unter dem nativen Meta-Format bricht auf 0,28 ein, ein Wert nahe dem Zufallsniveau, während die `gpirt`-Metrik den Einbruch abmildert. Ebenso zeigt die Gegenüberstellung mit und ohne `gen_prefix` bei llama.cpp, dass die Wahl des Auswertungspunkts im Tokenstream erheblichen Einfluss auf die Messqualität hat.

Diese Beobachtungen unterstreichen, dass Benchmark-Ergebnisse im LLM-Bereich ohne genaue Kenntnis der verwendeten Prompt-Formatierung, Few-Shot- bzw. Zero-Shot-Konfiguration, Auswertungsstrategie und Infrastruktur nur eingeschränkt vergleichbar sind. Die vollständige Dokumentation dieser Parameter ist daher nicht nur für die Reproduzierbarkeit dieser Arbeit, sondern auch für eine belastbare Aussage zur tatsächlichen Modellqualität wesentlich.

5.2 Beantwortung der Forschungsfrage

Die Forschungsfragen lassen sich auf Basis der erhobenen Daten wie folgt beantworten.

RQ1 (Interaktive Nutzbarkeit): Lokale LLM-Inferenz auf dem Raspberry Pi 5 ist praxistauglich, wobei die sinnvolle Modellgröße vom konkreten Einsatzszenario abhängt. Den Schwellenwert von ≥ 3 tok/s für interaktiven Dialogbetrieb erreicht das getestete 1B-Modell (Llama 3.2 1B, Q4_K_M) mit rund 11 tok/s auf beiden RPi5-Varianten komfortabel. Auch Gemma-3-4B-IT (Q4_K_M) überschreitet diesen Schwellenwert mit 4,15 tok/s und zeigt damit, dass interaktive Nutzung auf dem Raspberry Pi 5 nicht auf die 1B-Klasse beschränkt ist. Diese Aussage gilt jedoch nur für kurze Eingabekontexte: Die TTFT wächst linear mit der Prompt-Länge und liegt für Gemma-3-4B-IT bei 1024 Prompt-Tokens bereits bei rund 53 s, bei 4096 Tokens bei knapp 4 Minuten (vgl. Tabelle 4.13). Interaktiver Chatbetrieb mit längeren Prompts scheidet damit praktisch aus, unabhängig von der Token-Generierungsrate. 8B-Modelle (2,08–2,33 tok/s in den gemessenen Q4_K_M-Läufen) und 14B-Modelle (ca. 1,12 tok/s) sind für synchronen Chatbetrieb zu langsam, eignen sich aber gut für asynchrone Hintergrundaufgaben: E-Mail-Klassifikation, Spam-Erkennung und Dokumenten-Routing lassen sich mit diesen Token-Raten effizient und ohne Cloud-Abhängigkeit betreiben. Für interaktiven Betrieb mit Modellqualität jenseits der 3–5B-Klasse sind die gemessenen Raten aktuell nicht ausreichend. Dieser Anwendungsfall erfordert entweder leistungsfähigere

Hardware oder Modellgenerationen, die stärker auf ressourcenbeschränkte Inferenz ausgelegt sind.

RQ2 (TinyChatEngine vs. llama.cpp): Der Vergleich zeigt keinen praktisch relevanten Vorteil von TinyChatEngine 1 (AWQ) gegenüber llama.cpp (GGUF). TinyChatEngine ist als Forschungsplattform für AWQ interessant, unterscheidet sich von llama.cpp jedoch nicht nur im Gewichtsformat, sondern auch in Modellintegration, Prompt-Verarbeitung und Funktionsumfang. Bei der Inferenzgeschwindigkeit und in den betrachteten Qualitätsbenchmarks schneidet llama.cpp unter vergleichbaren Bedingungen besser ab. Für den Energieverbrauch pro Token erlauben die vorliegenden Messungen dagegen keinen eindeutigen Backend-Sieger, da TinyChatEngine zwar geringere CPU-Last und absolute Leistungsaufnahme zeigt, zugleich aber weniger Token pro Sekunde erzeugt. Ein wesentlicher Befund ist dabei, dass ein Teil der beobachteten Qualitätsunterschiede nicht auf AWQ selbst zurückzuführen ist, sondern auf Unterschiede in der Prompt-Pipeline, die in einem einfachen Backend-Vergleich leicht übersehen werden.

RQ3 (Empfohlene Konfiguration): Für den praktischen Einsatz auf dem Raspberry Pi 5 empfiehlt sich ein GGUF-basiertes Setup mit llama.cpp als Backend. Die höchste Leistungsreserve unter den modellgenau untersuchten LLaMA-3-8B-Instruct-Konfigurationen bietet die Q4_K_M-Quantisierung, die mit 2,33 tok/s die beste Token-Generierungsrate erzielt, bei einem `gpirt`-Verlust von lediglich $-0{,}015$ gegenüber `bf16`. Für Anwendungsfälle mit Priorität auf Interaktivität sind Modelle in der 1B-Klasse die schnellste Option. Llama 3.2 1B (Q4_K_M) hat sich dabei als komfortabel interaktiv nutzbar erwiesen und läuft auch auf dem 8-GB-System stabil. Wenn eine höhere Modellgüte wichtiger ist als maximale Geschwindigkeit, ist Gemma-3-4B-IT (Q4_K_M) auf dem Raspberry Pi 5 16 GB ein relevanter Kompromiss: Das Modell erreicht mit 4,15 tok/s noch interaktive Token-Raten und erzielt zugleich die höchste gemessene MATH500-Accuracy. TinyChatEngine bleibt als spezialisierte AWQ-Forschungsplattform relevant, ist für einen allgemeinen produktiven Einsatz auf dem Raspberry Pi 5 jedoch derzeit nicht die erste Wahl.

5.3 Einschränkungen und Ausblick

Der erhebliche Laufzeitaufwand der Benchmarks auf ressourcenbeschränkter Hardware begrenzte den erreichbaren Konfigurationsraum. Vollständige MMLU-Läufe erfordern auf dem Raspberry Pi 5 bei realistischen Inferenzraten mehrere Tage und wurden daher teilweise auf einem Desktop-System ausgeführt. vLLM, das offiziell ARM-Support ankündigt und als weiteres Backend evaluiert werden sollte, konnte auf dem Raspberry Pi 5 nicht erfolgreich kompiliert werden: Der Compiler-Prozess überschritt den verfügbaren Arbeitsspeicher, so-

dass eine Evaluation ausfiel und vLLM nicht in den Vergleich einbezogen werden konnte (vgl. Abschnitt 3.7).

Eine weitere Einschränkung betrifft TinyChatEngine selbst. Die Engine ist stark spezialisiert und nur begrenzt auf neue Modellfamilien übertragbar. Weitere Arbeiten könnten deshalb zwei Richtungen verfolgen: einerseits eine systematische Reproduktion der AWQ-Experimente in einer flexibleren Laufzeitumgebung, andererseits eine Ausweitung der Evaluation auf neuere kleinere Modellgenerationen, die speziell für Edge-Hardware optimiert wurden.

Ausblick Das Ökosystem für Edge-LLM-Inferenz entwickelt sich schnell. Neuere, speziell für ressourcenbeschränkte Hardware konzipierte Modellgenerationen in der Größenklasse 1–3B mit verbesserter Architektureffizienz könnten die in dieser Arbeit beobachteten Token-Raten-Grenzen deutlich verschieben, ohne die Modellgüte proportional einzubüßen. Parallel dazu gewinnen hardwareseitige Beschleuniger an Bedeutung: Externe NPU-Acceleratoren sowie integrierte Recheneinheiten in neueren ARM-SoCs könnten die Inferenzgeschwindigkeit auf eingebetteten Plattformen wesentlich verbessern, ohne den Formfaktor oder die Kosten grundlegend zu verändern. Gleiches gilt für erweiterte Vektorinstruktionen: Die gewichtsgebundenen Matrix-Vektor-Operationen der Decode-Phase stellen einen nahezu idealen Anwendungsfall für SIMD-Erweiterungen wie ARM SVE2 dar – eine Fähigkeit, die der Cortex-A76 des Raspberry Pi 5 noch nicht besitzt und die neuere ARM-Kerne unmittelbar einbringen würden. Mit der in dieser Arbeit aufgebauten Evaluationsinfrastruktur lassen sich solche Konfigurationen systematisch und reproduzierbar bewerten. Darüber hinaus eröffnet die Integration des lm-evaluation-harness die Möglichkeit, AWQ-quantisierte Modelle gezielt gegen domänenspezifische Datensätze zu evaluieren, etwa für Anwendungen in industriellen Steuerungssystemen, im Bildungsbereich oder in Offline-Assistenzsystemen, und so den tatsächlichen Nutzen von AWQ in konkreten Einsatzszenarien zu quantifizieren.

Abkürzungsverzeichnis

ADC	Analog-Digital-Wandler	20
API	Application Programming Interface	3
AWQ	Activation-aware Weight Quantization	1
BPE	Byte Pair Encoding	26
CPU	Central Processing Unit	1
CUDA	Compute Unified Device Architecture	10
FFN	Feed-Forward Network	6
FP16	16-Bit-Gleitkommazahl (Half Precision)	9
GGUF	GPT-Generated Unified Format	3
GQA	Grouped Query Attention	5
GPU	Graphics Processing Unit	1
I2C	Inter-Integrated Circuit	20
IRT	Item Response Theory	33
KV	Key-Value	5
LLM	Large Language Model	1
MHA	Multi-Head Attention	7
MQA	Multi-Query Attention	7
MMLU	Massive Multitask Language Understanding	2
NPU	Neural Processing Unit	3
NVMe	Non-Volatile Memory Express	17
OOM	Out of Memory	26
Prefill	Prompt-Verarbeitungsphase (erstes Token)	6
QKV	Query-Key-Value	19
RAM	Random Access Memory	1
REST	Representational State Transfer	3
SSD	Solid State Drive	17
SIMD	Single Instruction Multiple Data	1
SSE	Server-Sent Events	23
TTFT	Time to First Token	2

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literatur

[Ai23] Ainslie, J.; Lee-Thorp, J.; de Jong, M.; Zelasko, Y.; Sanghai, S.; Tay, Y.: GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. EMNLP 2023, 2023, URL: https://arxiv.org/abs/2305.13245.

[Fr23] Frantar, E.; Ashkboos, S.; Hoefler, T.; Alistarh, D.: GPTQ: Accurate Post-Training Quantization for Generative Pre-Trained Transformers. In: The Eleventh International Conference on Learning Representations. ICLR 2023, 2023, URL: https://arxiv.org/abs/2210.17323.

[Ga24] Gao, L.; Biderman, S.; Black, S.; Golding, L.; Hoppe, T.; Foster, C.; Phang, J.; He, H.; Thite, A.; Nabeshima, N.; Presser, S.; Leahy, C.: A Framework for Few-Shot Language Model Evaluation. arXiv preprint arXiv:2405.14782, 2024, URL: https://arxiv.org/abs/2405.14782.

[Ge25] Gemma Team, Google DeepMind: Gemma 3 Technical Report, Techn. Ber., Google DeepMind, 2025, URL: https://arxiv.org/abs/2503.19786.

[Ge26] Geekworm: NVMe SSD boot with the Raspberry Pi 5, 2026, URL: https://wiki.geekworm.com/NVMe_SSD_boot_with_the_Raspberry_Pi_5, Stand: 23. 04. 2026.

[Gl25] Gerganov, G.; llama.cpp contributors: llama.cpp, https://github.com/ggerganov/llama.cpp, Commit b8664, 2025.

[He21a] Hendrycks, D.; Burns, C.; Basart, S.; Zou, A.; Mazeika, M.; Song, D.; Steinhardt, J.: Measuring Massive Multitask Language Understanding. In: The Ninth International Conference on Learning Representations. ICLR 2021, 2021, URL: https://arxiv.org/abs/2009.03300.

[He21b] Hendrycks, D.; Burns, C.; Kadavath, S.; Arora, A.; Basart, S.; Tang, E.; Song, D.; Steinhardt, J.: Measuring Mathematical Problem Solving With the MATH Dataset. In: Thirty-Fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track. NeurIPS 2021, 2021, URL: https://arxiv.org/abs/2103.03874.

[He26] Heiming, L.: Effiziente lokale LLM-basierte Chatbots auf ressourcenbeschränkten Edge-GPUs, Bachelorarbeit, Hochschule Offenburg, 2026.

[Li24] Lin, J.; Tang, J.; Tang, H.; Yang, S.; Chen, W.-M.; Wang, W.-C.; Xiao, G.; Dang, X.; Gan, C.; Han, S.: AWQ: Activation-Aware Weight Quantization for On-Device LLM Compression and Acceleration. In: Proceedings of Machine Learning and Systems. Bd. 6, 2024, URL: https://arxiv.org/abs/2306.00978.

[Ll24] Llama Team, AI @ Meta: The Llama 3 Herd of Models, Techn. Ber., Meta AI, 2024, URL: https://arxiv.org/abs/2407.21783.

[Lo26] LocalScore: About LocalScore, https://www.localscore.ai/about, Accessed: 2026-04-27, 2026.

[MI24] MIT Han Lab: TinyChatEngine: On-Device LLM Inference Library, https://github.com/mit-han-lab/TinyChatEngine, 2024.

[Mo26] Mozilla AI: llamafile, https://github.com/mozilla-ai/llamafile, Accessed: 2026-04-27, 2026.

[Po24] Polo, F. M.; Weber, L.; Choshen, L.; Sun, Y.; Xu, G.; Yurochkin, M.: tinyBenchmarks: evaluating LLMs with fewer examples. arXiv preprint arXiv:2402.14992, 2024, URL: https://arxiv.org/abs/2402.14992.

[Qw24] Qwen Team, Alibaba Cloud: Qwen2.5 Technical Report, https://arxiv.org/abs/2412.15115, 2024.

[Ra25] Raschka, S.: Reasoning from Scratch – Code Repository, https://github.com/rasbt/reasoning-from-scratch, MMLU-Notebook: `chF/02_mmlu`; MATH500-Notebook: `ch03/01_main-chapter-code/ch03_main.ipynb`, 2025.

[Te15] Texas Instruments: INA219 Zero-Drift, Bidirectional Current/Power Monitor With I^2C Interface, Datasheet SBOS448G, revised December 2015, 2015.

[Te22] TechPowerUp: NVIDIA GeForce RTX 4090 Specs, 2022, URL: https://www.techpowerup.com/gpu-specs/geforce-rtx-4090.c3889, Stand: 21. 04. 2026.

[To23] Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S. et al.: Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv preprint arXiv:2307.09288, 2023.

[Va17] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; Polosukhin, I.: Attention Is All You Need. In: Advances in Neural Information Processing Systems. Bd. 30, 2017, URL: https://arxiv.org/abs/1706.03762.

[Xi23] Xiao, G.; Lin, J.; Seznec, M.; Wu, H.; Demouth, J.; Han, S.: SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In: Proceedings of the 40th International Conference on Machine Learning. ICML 2023, 2023, URL: https://arxiv.org/abs/2211.10438.

A Anhang

A.1 TinyChatEngineAPI– Konfigurationsdatei

Listing A.1 zeigt eine exemplarische `config.json` der TinyChatEngineAPI.

```
{
    "model": "LLaMA_3_8B_Instruct",
    "format": "INT4",
    "port": 8080,
    "threads": 4,
    "llama3_prompt_format": "native",
    "llama2_prompt_format": "legacy",
    "overwrite_model_params": false,
    "model_params": {
        "temp": 0.8,
        "top_p": 0.95,
        "top_k": 40,
        "n_predict": 128,
        "repeat_penalty": 1.1,
        "frequency_penalty": 0.0,
        "presence_penalty": 0.0,
        "seed": -1
    },
    "influxdb": {
        "enabled": false,
        "url": "http://rp4ab:8086",
        "org": "llm",
        "bucket": "llm",
        "token": "llm_admin_token_change_me"
    }
}
```

Listing A.1: Beispiel einer TinyChatEngineAPI-`config.json`

A.2 Monitoring-Stack – Docker-Compose-Konfiguration

Listing A.2 zeigt die Docker-Compose-Konfiguration des Monitoring-Stacks, der auf dem Raspberry Pi 4 betrieben wird. Der Stack umfasst InfluxDB 2 (Zeitreihendatenbank, Port 8086), Grafana (Visualisierung, Port 3000) und Telegraf (System-Metriken-Erfassung). Telegraf läuft im Host-Netzwerkmodus mit Read-only-Zugriff auf relevante Systemverzeichnisse (`/proc`, `/sys`, `/etc`).

```yaml
services:
  influxdb:
    image: influxdb:2
    container_name: influxdb
    environment:
      DOCKER_INFLUXDB_INIT_MODE: setup
      DOCKER_INFLUXDB_INIT_USERNAME: admin
      DOCKER_INFLUXDB_INIT_PASSWORD: admin_password
      DOCKER_INFLUXDB_INIT_ORG: llm
      DOCKER_INFLUXDB_INIT_BUCKET: telegraf
      DOCKER_INFLUXDB_INIT_ADMIN_TOKEN: llm_admin_token_change_me
    volumes:
      - influxdb_data:/var/lib/influxdb2
      - ./influxdb/init-buckets.sh:/docker-entrypoint-initdb.d/init-
          buckets.sh:ro
    ports:
      - "8086:8086"

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    environment:
      GF_SECURITY_ADMIN_PASSWORD: admin_password
    volumes:
      - grafana_data:/var/lib/grafana
      - ./grafana/provisioning:/etc/grafana/provisioning:ro
      - ./grafana/dashboards:/var/lib/grafana/dashboards:ro
    ports:
      - "3000:3000"

  telegraf:
    image: telegraf:alpine
    container_name: telegraf
    hostname: ${HOSTNAME}
    network_mode: host
    user: "telegraf:998"
    env_file: .env
    volumes:
      - ./telegraf/telegraf.conf:/etc/telegraf/telegraf.conf:ro
      - /:/hostfs:ro
      - /etc:/hostfs/etc:ro
      - /proc:/hostfs/proc:ro
      - /sys:/hostfs/sys:ro
      - /var/run/utmp:/var/run/utmp:ro
    environment:
      HOST_ETC: /hostfs/etc
      HOST_PROC: /hostfs/proc
      HOST_SYS: /hostfs/sys
      HOST_MOUNT_PREFIX: /hostfs
    restart: unless-stopped

```

```
51 volumes:
52   influxdb_data:
53   grafana_data:
```

Listing A.2: Docker-Compose-Konfiguration des Monitoring-Stacks

A.3 Schaltplan des Leistungsmessaufbaus

Abbildung A.1 zeigt den vollständigen Schaltplan des externen Leistungsmessaufbaus mit ESP32 und zwei INA219-Sensoren.

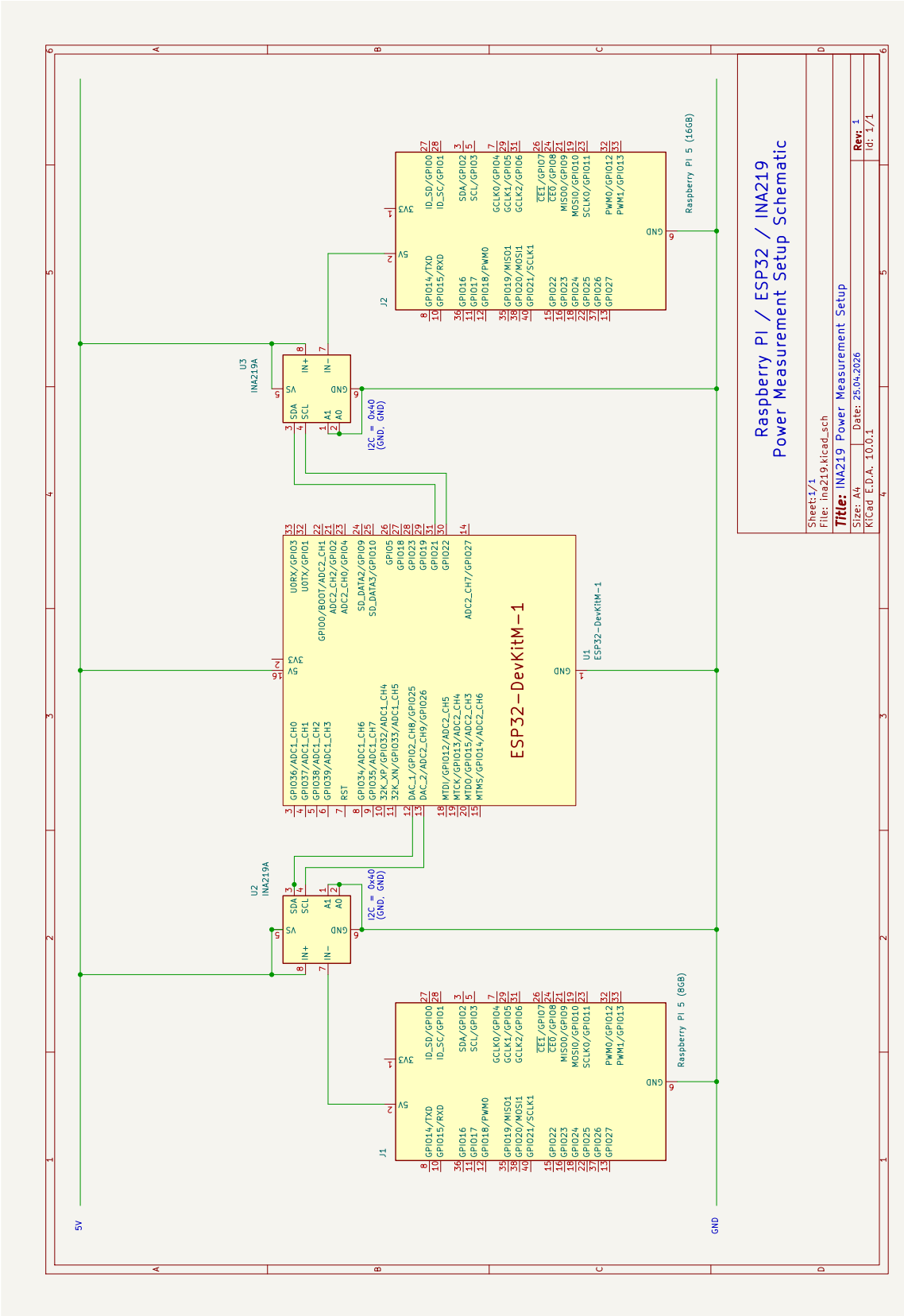

Abbildung A.1: Schaltplan des externen Leistungsmessaufbaus mit ESP32 und zwei INA219-Sensoren

A.4 LocalScore-Ergebnisse

Die folgenden fünf Abschnitte zeigen die vollständigen LocalScore-Terminalausgaben für die auf dem Raspberry Pi 5 16 GB getesteten Modellkonfigurationen (llamafile 0.9.2). Jede Ausgabe enthält die Einzelwerte aller neun Testfälle sowie den berechneten Gesamtscore. Sie ergänzen die in Abschnitt 4.5 zusammengefassten Kennzahlen.

Llama 3.2 1B Instruct – Q4_K_M

```
CPU: Raspberry Pi 5 Model B Rev 1.1 | RAM: 15.8 GiB

+-----------------------------------------------------------------------------------------------+
|                          Raspberry Pi 5 Model B Rev 1.1 - 15.8 GiB                            |
|                            Llama 3.2 1B Instruct - Q4_K - Medium                              |
+-----------------------------------------------------------------------------------------------+
|         test | run number | avg time | tokens processed | pp t/s | tg t/s    | ttft      |
| ------------ | ---------- | ---------- | ---------------- | ---------- | ---------- | ---------- |
|   pp1024+tg16 | 1/1      | 19.83 s    | 1040 / 1040      | 55.32      | 12.14      | 18.59 s    |
|  pp4096+tg256 | 1/1      | 132.35 s   | 4352 / 4352      | 39.47      | 8.96       | 103.88 s   |
|  pp2048+tg256 | 1/1      | 65.87 s    | 2304 / 2304      | 48.36      | 10.88      | 42.49 s    |
|  pp2048+tg768 | 1/1      | 115.23 s   | 2816 / 2816      | 48.48      | 10.52      | 42.33 s    |
| pp1024+tg1024 | 1/1      | 110.55 s   | 2048 / 2048      | 54.56      | 11.16      | 18.85 s    |
| pp1280+tg3072 | 1/1      | 331.54 s   | 4352 / 4352      | 55.74      | 9.96       | 23.05 s    |
|  pp384+tg1152 | 1/1      | 104.22 s   | 1536 / 1536      | 57.86      | 11.81      | 6.77 s     |
|   pp64+tg1024 | 1/1      | 84.81 s    | 1088 / 1088      | 64.01      | 12.22      | 1.07 s     |
|   pp16+tg1536 | 1/1      | 128.87 s   | 1552 / 1552      | 61.88      | 11.94      | 330.71 ms  |
+-----------------------------------------------------------------------------------------------+

   __    ____ _________   __   _____ __________ ____ ______
  / /   / __ \/ ____/ |  / /  / ___// ____/ __ \/ __ \/ ____/
 / /   / / / / /   / /| | / /   \__ \/ /   / / / / /_/ / __/
/ /___/ /_/ / /___/ ___ |/ /______/ / /___/ /_/ / _, _/ /___
/_____/\____/\____/_/ |_/_____/____/\____/\____/_/ |_/_____/

Score: 27

Token Generation:      11.06 tok/s
Prompt Processing:     53.96 tok/s
Time to First Token:   28594.67 ms

Result: https://www.localscore.ai/result/2561
```

Listing A.3: LocalScore-Terminalausgabe: Llama 3.2 1B Instruct Q4_K_M (llamafile 0.9.2, Raspberry Pi 5 16 GB)

DeepSeek R1 Distill Qwen 1.5B – Q4_K_M

```
CPU: Raspberry Pi 5 Model B Rev 1.1 | RAM: 15.8 GiB

+-----------------------------------------------------------------------------------------------------+
|                          Raspberry Pi 5 Model B Rev 1.1 - 15.8 GiB                                  |
|                        DeepSeek R1 Distill Qwen 1.5B - Q4_K - Medium                                |
+-----------------------------------------------------------------------------------------------------+
|         test | run number | avg time | tokens processed | pp t/s | tg t/s    | ttft      |
| ------------ | ---------- | ---------- | ---------------- | ---------- | ---------- | ---------- |
|   pp1024+tg16 | 1/1      | 24.79 s    | 1040 / 1040      | 43.88      | 10.97      | 23.43 s    |
|  pp4096+tg256 | 1/1      | 139.49 s   | 4352 / 4352      | 36.61      | 9.27       | 112.00 s   |
|  pp2048+tg256 | 1/1      | 74.27 s    | 2304 / 2304      | 41.39      | 10.32      | 49.57 s    |
|  pp2048+tg768 | 1/1      | 125.02 s   | 2816 / 2816      | 41.52      | 10.15      | 49.43 s    |
| pp1024+tg1024 | 1/1      | 120.69 s   | 2048 / 2048      | 44.76      | 10.47      | 22.97 s    |
| pp1280+tg3072 | 1/1      | 341.75 s   | 4352 / 4352      | 44.32      | 9.82       | 28.98 s    |
|  pp384+tg1152 | 1/1      | 113.23 s   | 1536 / 1536      | 46.69      | 10.97      | 8.31 s     |
|   pp64+tg1024 | 1/1      | 92.66 s    | 1088 / 1088      | 48.87      | 11.21      | 1.39 s     |
|   pp16+tg1536 | 1/1      | 139.22 s   | 1552 / 1552      | 45.89      | 11.06      | 432.93 ms  |
+-----------------------------------------------------------------------------------------------------+

   __    ____ _________   __   _____ __________ ____ ______
  / /   / __ \/ ____/ |  / /  / ___// ____/ __ \/ __ \/ ____/
 / /   / / / / /  / /| | / /   \__ \/ /   / / / / /_/ / __/
/ /___/ /_/ / /___/ ___ |/ /______/ / /___/ /_/ / _, _/ /___
/_____/\____/\____/_/ |_/_____/____/\____/\____/_/ |_/_____/

Score: 27

Token Generation:      10.47 tok/s
Prompt Processing:     43.77 tok/s
Time to First Token:   32946.43 ms
```

Listing A.4: LocalScore-Terminalausgabe: DeepSeek R1 Distill Qwen 1.5B Q4_K_M (llamafile 0.9.2, Raspberry Pi 5 16 GB)

Gemma-3-4B-IT – Q4_K_M

```
CPU: Raspberry Pi 5 Model B Rev 1.1 | RAM: 15.8 GiB

+-----------------------------------------------------------------------------------------------+
|                          Raspberry Pi 5 Model B Rev 1.1 - 15.8 GiB                            |
|                               Gemma 3 4b It - Q4_K - Medium                                   |
+-----------------------------------------------------------------------------------------------+
|         test | run number | avg time | tokens processed | pp t/s | tg t/s    | ttft      |
| ------------ | ---------- | -------- | ---------------- | ------ | --------- | --------- |
|   pp1024+tg16 | 1/1       | 55.98 s  | 1040 / 1040      | 19.57  | 4.39      | 52.55 s   |
|  pp4096+tg256 | 1/1       | 302.60 s | 4352 / 4352      | 17.64  | 3.64      | 232.54 s  |
|  pp2048+tg256 | 1/1       | 170.87 s | 2304 / 2304      | 18.90  | 4.09      | 108.60 s  |
|  pp2048+tg768 | 1/1       | 300.32 s | 2816 / 2816      | 18.89  | 4.00      | 108.67 s  |
| pp1024+tg1024 | 1/1       | 296.09 s | 2048 / 2048      | 19.55  | 4.20      | 52.61 s   |
| pp1280+tg3072 | 1/1       | 872.23 s | 4352 / 4352      | 19.56  | 3.81      | 65.71 s   |
|  pp384+tg1152 | 1/1       | 283.32 s | 1536 / 1536      | 20.02  | 4.36      | 19.40 s   |
|   pp64+tg1024 | 1/1       | 232.20 s | 1088 / 1088      | 20.33  | 4.47      | 3.36 s    |
|   pp16+tg1536 | 1/1       | 351.00 s | 1552 / 1552      | 18.67  | 4.39      | 1.07 s    |
+-----------------------------------------------------------------------------------------------+


    __   ____ _________   __   _____ __________ ____ ______
   / /  / __ \/ ____/ |  / /  / ___// ____/ __ \/ __ \/ ____/
  / /  / / / / /   / /| | / /   \__ \/ /   / / / / /_/ / __/
 / /___/ /_/ / /___/ ___ |/ /______/ / /___/ /_/ / _, _/ /___
/_____/\____/\____/_/ |_/_____/____/\____/\____/_/ |_/_____/

Score: 10

Token Generation:      4.15 tok/s
Prompt Processing:     19.24 tok/s
Time to First Token:   71611.47 ms
```

Listing A.5: LocalScore-Terminalausgabe: Gemma-3-4B-IT Q4_K_M (llamafile 0.9.2, Raspberry Pi 5 16 GB)

Llama 3.1 8B Instruct – Q4_K_M

```
CPU: Raspberry Pi 5 Model B Rev 1.1 | RAM: 15.8 GiB

+-----------------------------------------------------------------------------------------------+
|                          Raspberry Pi 5 Model B Rev 1.1 - 15.8 GiB                            |
|                         Meta Llama 3.1 8B Instruct - Q4_K - Medium                            |
+-----------------------------------------------------------------------------------------------+
|         test | run number | avg time | tokens processed | pp t/s | tg t/s    | ttft       |
| ------------ | ---------- | -------- | ---------------- | ------ | --------- | ---------- |
|   pp1024+tg16 | 1/1       | 135.04 s | 1040 / 1040      | 8.01   | 2.21      | 128.27 s   |
|  pp4096+tg256 | 1/1       | 734.98 s | 4352 / 4352      | 6.88   | 1.83      | 595.84 s   |
|  pp2048+tg256 | 1/1       | 393.20 s | 2304 / 2304      | 7.61   | 2.06      | 269.54 s   |
|  pp2048+tg768 | 1/1       | 650.08 s | 2816 / 2816      | 7.60   | 2.02      | 269.87 s   |
| pp1024+tg1024 | 1/1       | 614.24 s | 2048 / 2048      | 8.05   | 2.10      | 127.66 s   |
| pp1280+tg3072 | 1/1       | 1745.34 s | 4352 / 4352     | 7.99   | 1.94      | 160.78 s   |
|  pp384+tg1152 | 1/1       | 577.61 s | 1536 / 1536      | 8.24   | 2.17      | 47.06 s    |
|   pp64+tg1024 | 1/1       | 469.12 s | 1088 / 1088      | 8.49   | 2.22      | 7.94 s     |
|   pp16+tg1536 | 1/1       | 703.81 s | 1552 / 1552      | 7.75   | 2.19      | 2.47 s     |
+-----------------------------------------------------------------------------------------------+

   __   ____ _________   __   _____ __________ ____ ______
  / /  / __ \/ ____/ |  / /  / ___// ____/ __ \/ __ \/ ____/
 / /  / / / / /  / /| | / /   \__ \/ /   / / / / /_/ / __/
/ /___/ /_/ / /___/ ___ |/ /______/ / /___/ /_/ / _, _/ /___
/_____/\____/\____/_/ |_/_____/____/\____/\____/_/ |_/_____/

Score: 4

Token Generation:      2.08 tok/s
Prompt Processing:     7.85 tok/s
Time to First Token:   178825.95 ms
```

Listing A.6: LocalScore-Terminalausgabe: Llama 3.1 8B Instruct Q4_K_M (llamafile 0.9.2, Raspberry Pi 5 16 GB)

Qwen 2.5 14B Instruct – Q4_K_M

```
CPU: Raspberry Pi 5 Model B Rev 1.1 | RAM: 15.8 GiB

+-----------------------------------------------------------------------------------------------+
|                          Raspberry Pi 5 Model B Rev 1.1 - 15.8 GiB                            |
|                            Qwen2.5 14B Instruct - Q4_K - Medium                               |
+-----------------------------------------------------------------------------------------------+
|         test | run number | avg time | tokens processed | pp t/s | tg t/s    | ttft       |
| ------------ | ---------- | -------- | ---------------- | ------ | --------- | ---------- |
|   pp1024+tg16 | 1/1       | 240.56 s | 1040 / 1040      | 4.51   | 1.18      | 227.93 s   |
|  pp4096+tg256 | 1/1       | 1334.78 s | 4352 / 4352     | 3.81   | 0.99      | 1076.40 s  |
|  pp2048+tg256 | 1/1       | 710.69 s | 2304 / 2304      | 4.28   | 1.10      | 479.08 s   |
|  pp2048+tg768 | 1/1       | 1186.04 s | 2816 / 2816     | 4.29   | 1.08      | 478.29 s   |
| pp1024+tg1024 | 1/1       | 1130.70 s | 2048 / 2048     | 4.53   | 1.13      | 226.88 s   |
| pp1280+tg3072 | 1/1       | 3232.52 s | 4352 / 4352     | 4.51   | 1.04      | 284.53 s   |
|  pp384+tg1152 | 1/1       | 1067.20 s | 1536 / 1536     | 4.66   | 1.17      | 83.25 s    |
|   pp64+tg1024 | 1/1       | 872.94 s | 1088 / 1088      | 4.69   | 1.19      | 14.46 s    |
|   pp16+tg1536 | 1/1       | 1309.43 s | 1552 / 1552     | 4.64   | 1.18      | 4.27 s     |
+-----------------------------------------------------------------------------------------------+

   __   ____ _________   __   _____ __________ ____ ______
  / /  / __ \/ ____/ |  / /  / ___// ____/ __ \/ __ \/ ____/
 / /  / / / / /   / /| | / /   \__ \/ /   / / / / /_/ / __/
/ /___/ /_/ / /___/ ___ |/ /______/ / /___/ /_/ / _, _/ /___
/_____/\____/\____/_/ |_/_____/____/\____/\____/_/ |_/_____/

Score: 2

Token Generation:      1.12 tok/s
Prompt Processing:     4.44 tok/s
Time to First Token:   319454.25 ms

Result: https://www.localscore.ai/result/2563
```

Listing A.7: LocalScore-Terminalausgabe: Qwen 2.5 14B Instruct Q4_K_M (llamafile 0.9.2, Raspberry Pi 5 16 GB)

A.5 Grafana-Dashboards

Die folgenden drei Abbildungen zeigen die im Rahmen dieser Arbeit eingesetzten Grafana-Dashboards. Sie ergänzen die Beschreibungen in den Abschnitten 3.6.3, 3.4.3 und 3.8.

A.5.1 Übersichts-Dashboard

Abbildung A.2 zeigt das kombinierte Übersichts-Dashboard, das Leistungs- und Inferenzdaten einer Messsession zusammenfasst. Am oberen Rand des Dashboards befinden sich zwei Dropdowns: eines zur Auswahl des INA219-Sensors, dessen Leistungsdaten angezeigt werden, und eines zur Auswahl des Telegraf-Hosts, dessen CPU-Metriken dargestellt werden. Damit lässt sich das Dashboard ohne Anpassung für beide Raspberry-Pi-5-Systeme einsetzen. Darunter folgen im oberen Bereich drei Zeitreihen-Panels mit Inferenzmetriken, gefolgt von drei Stat-Panels, die jeweils einen einzelnen Wert hervorheben: zwei Zeitmetriken sowie den Energieverbrauch in Milliwattstunden. Im unteren Bereich zeigt ein Leistungsgraph den Stromverbrauch im Zeitverlauf mit zwei getrennten Linien für Board A (RPi5 16 GB) und Board B (RPi5 8 GB). Darunter befinden sich die CPU-Auslastung aufgeschlüsselt nach Kernen sowie die Speicherauslastung des ausgewählten Systems.

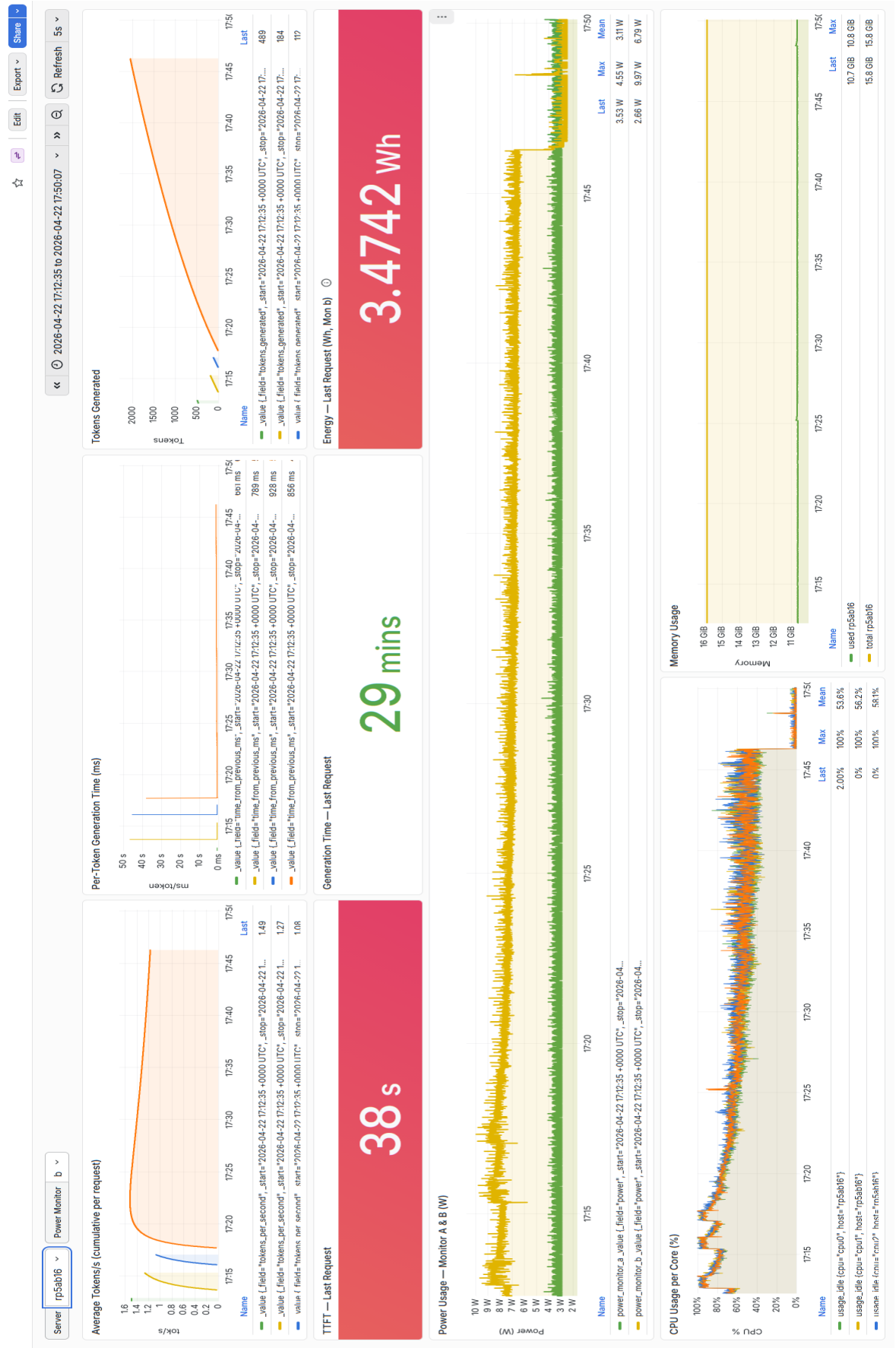

Abbildung A.2: Grafana-Übersichts-Dashboard: Stat-Panels für Laufzeit und Energieverbrauch sowie Zeitreihen der Inferenzmetriken über eine vollständige Messsitzung (eigene Aufnahme)

A.5.2 Leistungsmessungs-Dashboard

Abbildung A.3 zeigt das Leistungsmessungs-Dashboard, das die Rohdaten beider INA219-Sensoren in Echtzeit visualisiert. Die vier Stat-Panels am oberen Rand geben die aktuellen Mittelwerte der beiden Boards wieder: Board A (RPi5 16 GB) mit 2,93 W und 571 mA, Board B (RPi5 8 GB) mit 6,72 W und 1,31 A. Die Zeitreihen-Graphen darunter zeigen, jeweils in separaten Panels, Spannung, Strom, Leistung und Shunt-Spannung beider Kanäle. Gelbe und grüne Kurven kennzeichnen die beiden Sensor-Kanäle und ermöglichen so die simultane Beobachtung der Leistungsaufnahme beider Systeme in einem gemeinsamen zeitlichen Kontext. Die feine zeitliche Auflösung lässt die charakteristischen Phasen eines Inferenzlaufs erkennen: den kurzen Leistungspeak beim Modell-Laden, die erhöhte Grundlast während der Prefill-Phase und das gleichmäßigere Profil während der Token-Generierung.

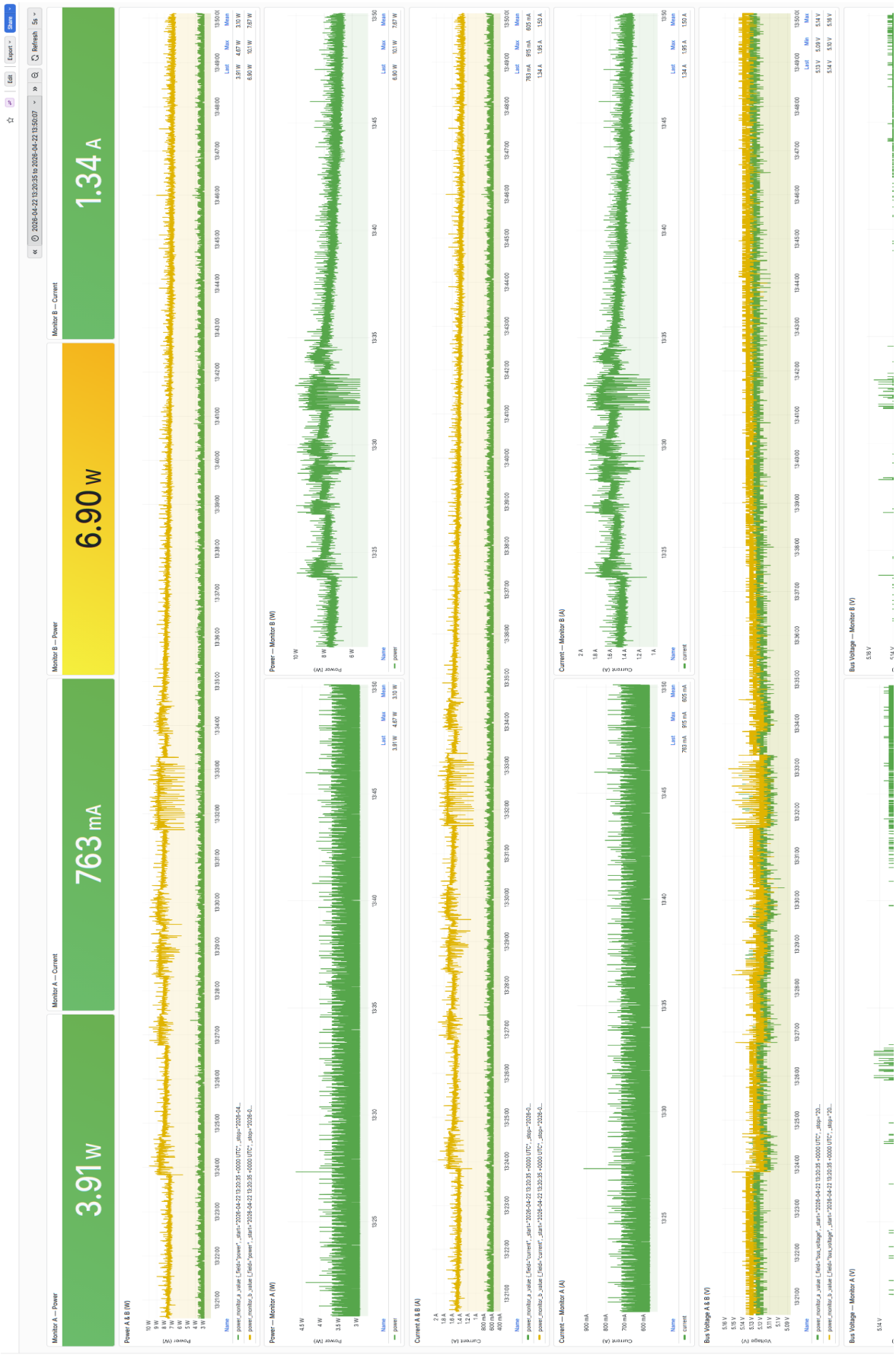

Abbildung A.3: Grafana-Leistungsmessungs-Dashboard: Spannung, Strom, Leistung und Shunt-Spannung beider INA219-Kanäle (Board A: RPi5 16 GB, Board B: RPi5 8 GB) im Zeitverlauf (eigene Aufnahme)

A.5.3 TinyChatEngine-Inferenz-Dashboard

Abbildung A.4 zeigt das TinyChatEngine-spezifische Inferenz-Dashboard, das die vom `influx_logger`-Modul in InfluxDB geschriebenen Metriken visualisiert.

Das Dashboard gliedert sich in zwei Bereiche. Der obere Bereich enthält vier Zeitreihen-Panels, die die Qualität der Token-Generierung erfassen. *Tokens per Second* (oben links) zeigt den kumulativen Durchschnitt der Token-Rate: Weil das erste Token die gesamte Prefill-Zeit einschließt, startet der Wert nahe null und steigt mit jedem weiteren generierten Token an. Bei sehr langen Requests sinkt er gegen Ende langsam wieder ab, da die Decode-Geschwindigkeit mit wachsendem KV-Cache abnimmt und die Speicherbandbreite zur dominierenden Engstelle wird. *Average Token Time* (oben rechts) zeigt denselben Sachverhalt invers als mittlere Token-Generierungszeit, und zwar in zwei Kurven in einem gemeinsamen Graphen: einmal inklusive TTFT und einmal ohne, um die reine Decode-Phase isoliert beurteilen zu können. *Per Token Generation Time* (Mitte links) stellt die Generierungszeit jedes einzelnen Tokens nicht gemittelt dar und macht so kurzzeitige Schwankungen sichtbar. *Cumulative Tokens Generated* (Mitte rechts) zeigt den kumulativen Tokenstand als Treppenkurve und dient als Fortschrittsindikator für laufende Inferenzläufe.

Der untere Bereich des Dashboards speichert und visualisiert die Inferenz-Konfigurationsparameter des jeweiligen Requests, darunter Temperatur, Top-P und Top-K, und ermöglicht so die nachträgliche Zuordnung von Messergebnissen zu den verwendeten Sampling-Einstellungen.

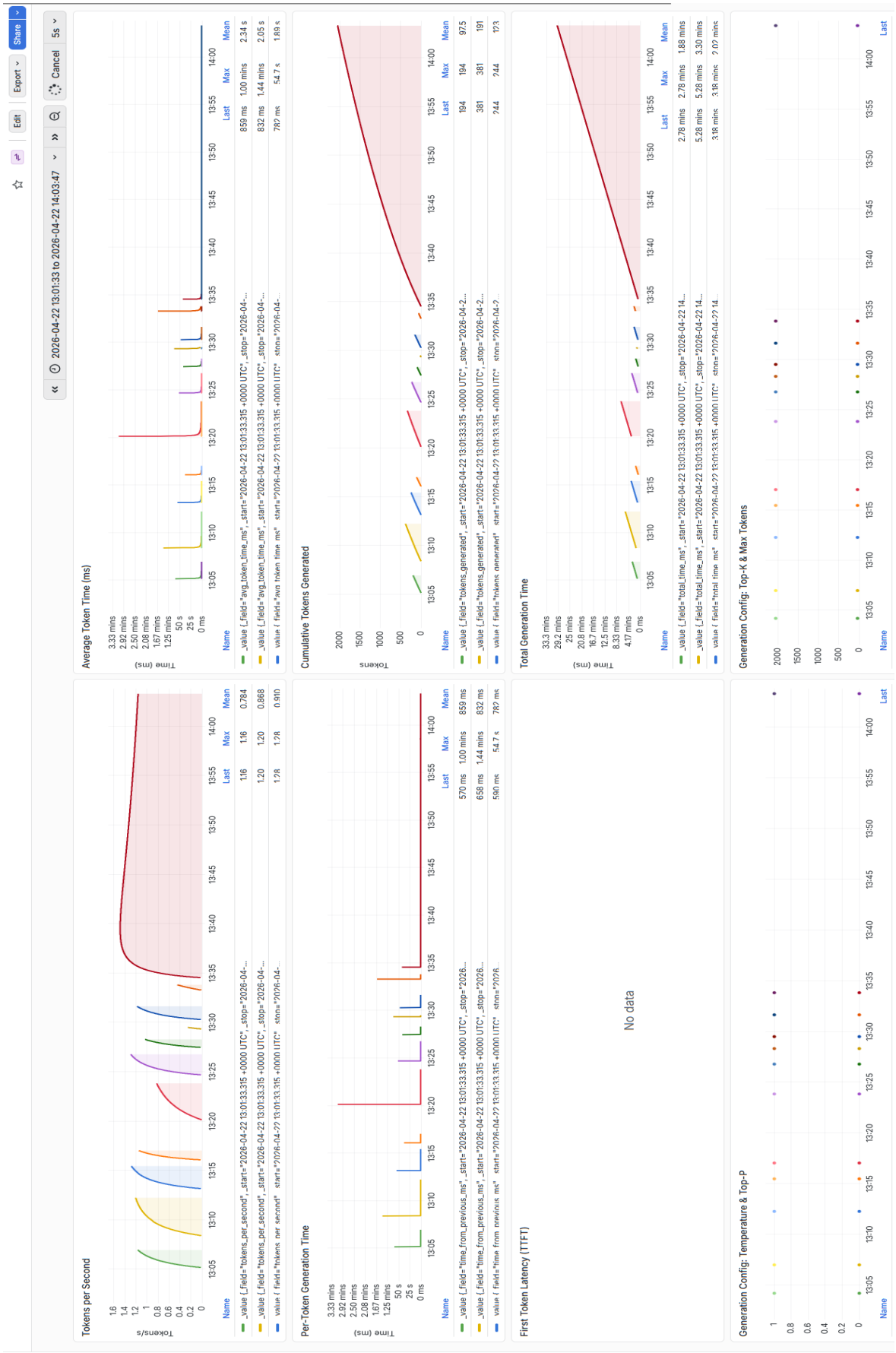

Abbildung A.4: Grafana-Dashboard für TinyChatEngine-Inferenzmetriken: Tokens per Second (kumulativer Durchschnitt), mittlere Token-Zeit mit und ohne TTFT, nicht gemittelte Token-Generierungszeit, kumulierter Tokenstand sowie Inferenz-Konfigurationsparameter (eigene Aufnahme)