electronics

MDPI

Article

Heterogeneous Low-Power Agentic Architecture for Edge Distributed Home Surveillance

Luigi Lucio Zappia, Marco Lattuada, Marco Bianchessi and Danilo Pietro Pau *

System Research and Applications, STMicroelectronics, 20864 Agrate Brianza, Italy; luigilucio.zappia@st.com (L.L.Z.); marco.lattuada@st.com (M.L.); marco.bianchessi@st.com (M.B.)
* Correspondence: danilo.pau@st.com

Abstract

Recent advancements in edge hardware enabled increasingly complex artificial intelligence workloads to be executed directly on resource-constrained devices. However, single-edge devices remain computationally limited compared with hybrid or cloud alternatives. This work presents a fully edge-based architecture implementing agentic AI on heterogeneous devices for home surveillance with a natural language interface, in which analysis tasks are parallelized across the distributed devices. The system is tested on a two-node prototype: a Raspberry Pi 5 with a Hailo-10H accelerator and an STM32MP257FDK board. Dynamic leader election allows recovery from the loss of any single node, including the leader, with graceful degradation. Qwen3-1.7B achieved 93% tool-selection accuracy over 24 tools and maintained robust summarization performance, outperforming FunctionGemma-270M. The jina-reranker-v2-base-multilingual cross-encoder performs comparably to the tested LLMs at tool selection on small pool sizes (i.e., with few tools to choose from). Hardware accelerators improved inference energy efficiency by 2.0× to 17.3×, while leader execution incurred less than 2 W of additional power. The main bottleneck is model latency rather than network overhead, and KV-cache reuse proved essential to meet the latency constraint on the STM32MP257FDK. The proposed prototype coordinates inference across heterogeneous edge devices to answer queries, with the weaker node exhibiting scalability limitations.

Keywords: distributed networking; edge artificial intelligence; tiny machine learning; edge language models; agentic frameworks; microprocessors; low power

Check for updates

Academic Editor: Fernando De la Prieta Pintado

Received: 10 June 2026
Revised: 15 July 2026
Accepted: 17 July 2026
Published: 19 July 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

In recent years, the adoption of surveillance technologies has accelerated. The market reflects this trend: the global video surveillance market was valued at USD 73.75 billion in 2024 and is projected to reach USD 147.66 billion by 2030 with a compound annual growth rate (CAGR) of 12.1% between 2025 and 2030 [1] (Refs. [1,2] are market-research reports (Grand View Research and Mordor Intelligence), while refs. [3,4] are vendor reports published by physical-security technology companies (Genetec and Axis Communications). They are not peer-reviewed studies and are cited here solely to provide market and industry context). Advancements in technology and rising security concerns across multiple sectors are key drivers of this growth [1,2].

According to an industry survey by security firm Genetec, as of early 2026, an increasing number of surveillance systems rely on hybrid (using both edge and cloud hardware) or cloud-based approaches [3] that may use company servers or serverless architectures; for

example, using services such as Amazon Web Services or Microsoft Azure. The use of cloud infrastructures reduces the initial investments required [5] and removes upfront hardware outlays [2], facilitating the diffusion of such systems. A fully cloud-based architecture uses satellite devices (e.g., sensors) connected to an external computer. These satellite devices stream raw data to a central server where complete analysis is performed. This approach is attractive because it offers low initial costs for edge users, high scalability, ease of deployment, and access to vast computational resources in the cloud [5], which seem virtually unlimited. Even though cloud adoption in surveillance is accelerating, on-premises (i.e., locally installed) video management systems and hybrid solutions still remain prevalent [3], with 63.75% of the 2025 deployments still being on-premises, mainly due to financial and healthcare operators preferring local control [2], which also meets privacy requirements. According to market research firm Mordor Intelligence, three of the biggest companies in the surveillance market are Hikvision, Dahua, and Axis Communications: all of them provide hybrid solutions merging both Artificial Intelligence (AI) on edge and cloud analysis tools [2]. Axis' State of AI in Video Surveillance report found that 62% of system integrators see AI and generative AI (GenAI) as the main trends, while end customers are more focused on analytics and turning data into practical, usable insights [4]. The reduction in the costs of edge cameras and the increasing availability of Deep Learning (DL) [6] and Tiny Machine Learning (TinyML) models have made it much easier to perform analysis on surveillance data on edge solutions.

Agentic AI and AI Agents have developed considerably in recent years and have increased in popularity thanks to the release of Large Language Models (LLMs) and Vision-Language Models (VLMs): these technologies allow for generalization beyond fixed pipelines and generative reasoning [7,8]. The most important task an LLM-based agent must perform is tool-calling. Tool-calling is a technique that allows LLMs to execute pre-written deterministic functions. The recent maturity of LLM-based agents and tool-calling frameworks has opened new possibilities for orchestrating surveillance pipelines that can reason, perform actions, and reply to user queries in natural language [7,8]. LLM-powered agentic AI running on edge represents a promising approach for camera control tasks [9].

That said, due to computational constraints [10], AI models running on edge can be both less capable and slower than AI models running on cloud devices. Rather than relying on monolithic models that process video feeds in isolation, agentic architectures can decompose complex tasks into modular sub-tasks that can be distributed across agents [8], and thus, potentially, different devices [11]. Singh [11] argues that distributed AI systems offer advantages in parallelism, robustness, and scalability. The modular nature of such systems simplifies both conceptualization and implementation of complex cognitive tasks: thus, the agentic paradigm also provides a useful abstraction to model problem-solving efforts in this domain [11]. This flexibility makes multi-agent systems particularly well suited for heterogeneous surveillance deployments, which are characterized by devices with different capabilities.

This paper proposes a fully edge-based agentic architecture for distributed home surveillance. Prior work addressed individual aspects of this setting but not their integration. Edge tool-calling methods such as TinyAgent [12] and ToolLLM [13] improved accuracy by fine-tuning LLMs, but operated on a single device and targeted only the model executing the calls rather than the overall system. Tool-retrieval studies such as ToolRet [14] found that retrieval models performed poorly on large tool corpora, but did not benchmark them on tool selection directly against LLMs. Edge video-analytics projects such as DeepCamera [15] concentrated coordination and analysis on a single node, without addressing the single point of failure (SPOF) problem. Distributed agent-placement methods such as that of Wang et al. [16] optimized agent placement but did not coordinate multi-sensor

inference under node failure. The novelty of this work lies not in its individual building blocks but in their combination within the proposed architecture and in the evaluation that characterizes its performance. The key design choices are distributed tool-calling and leader election. These decouple the LLM from the analysis it requests and provide fault tolerance: a single tool-call can be executed in parallel across several sensing devices, and the failure of any node, including the coordinator, does not stop the whole system from operating. Tool-calling accuracy and reply quality are evaluated on 24 hand-crafted home-surveillance tools over a synthetic dataset generated with Claude Opus 4.7. Power consumption and latency are measured directly on the edge devices, analyzing the efficiency gains enabled by AI accelerators. No model is fine-tuned on the evaluation tool set; this preserves the models' generalization capabilities and keeps the system modular, since in the proposed implementation, a new tool can be added by uploading only two files, with no retraining. Large-scale fine-tuning is left to future work: the ToolLLM study [13] reported that fine-tuning can also improve accuracy on out-of-distribution tools. A detailed comparison with these and other studies is provided in Section 3.

The rest of this study is organized as follows. Section 2 introduces the research question, states the requirements that any proposed solution must respect, and gives a formal definition of the problem. Section 3 reviews related work and discusses existing systems whose knowledge is helpful to introducing the proposed system. Section 4 highlights the contributions of this paper. Section 5 describes the proposed system architecture, the implementation details, the software stack, and the hardware used. Section 6 presents the experimental methodology and the results concerning the tool-calling accuracy of the proposed approaches. Section 7 presents the experimental methodology and the results concerning power consumption, latency, and scalability of the edge components. Section 8 compares metrics and offers insights into the results. Section 9 summarizes the findings and their implications for future research on agentic systems.

2. Research Question, Requirements, and Problem Statement

This section introduces the research question, states the requirements that any proposed solution must respect, and gives a formal definition of the problem. The main question is introduced in Section 2.1, and it is divided into subquestions or goals, closely tied to the requirements. The requirements are stated in Section 2.2. The problem this work addresses is formalized in Section 2.3.

2.1. Research Question and Objectives

The central research question of this work is: "Can a heterogeneous network of resource-constrained edge devices sustain real-time multi-sensor analytics and conversational querying with no cloud dependency, while also tolerating the failure of any single node, including the coordinating one?"

To address this question systematically, it is decomposed into three practical subquestions (or objectives) that drive the remainder of this paper. These objectives will determine the proposed approach and its experimental evaluation.

- Feasibility: Can generative reasoning on edge devices interpret commands with sufficient accuracy, and can distributed sensor inference scale while preserving accuracy?
- Efficiency: What are the latency and power consumption profiles of the main nodes under representative workloads?
- Resilience: Can the system maintain service continuity when any node fails?

2.2. Requirements

The system is expected to satisfy the previously mentioned objectives by fulfilling the following requirements:

- Edge-only operation: to ensure reliability, inference, coordination, and data storage shall run entirely on edge devices, without dependence on cloud infrastructure.
- Responsiveness: the system must sustain surveillance question answering (Q&A). The end-to-end latency from the reception of the user query to the reply shall not exceed 20 s.
- Fault tolerance: the system shall continue operating if any single node fails, including the coordinating node.
- Horizontal scalability: inference workloads shall be distributed across heterogeneous nodes and remain efficient as devices are added or removed.
- Generative reasoning accuracy: the conversational interface shall interpret user commands to orchestrate sensor operations correctly.
- Energy efficiency: power consumption shall remain within the operational limits of edge hardware to support sustained deployment.

These requirements reflect the constraints imposed by edge deployment, the need for uninterrupted operation, and the interaction patterns expected from conversational control of distributed sensing.

2.3. Formal Problem Description

This subsection gives a formal description of the problem addressed by this work. It does not depend on a specific implementation and formalizes the requirements stated in Section 2.2.

- Edge devices. The problem is defined over a set of M heterogeneous edge devices $D = \{d_1, \ldots, d_M\}$, differing in hardware capabilities. No cloud component is required for inference, coordination, or storage.
- User queries. A user query q is a natural language request about the state of the monitored environment. The expected result is a natural language reply y.
- Tool sets. Each device d_i can execute a tool set $T_i \subseteq T$, possibly empty. A tool $t \in T_i$ is a function that reads a sensor or runs an on-device model on sensor data. The global tool set is $T = \bigcup_{i=1}^{M} T_i$.
- Sensing outputs. The execution of a tool t produces a structured output o. The outputs collected for a query q form the set O_q.
- Conversational state. Serving a query q requires a conversational state $C = (q, h)$, where q is the pending query and h is its chat history when available. This state must be preserved while q is being served.
- Failure model. Failures are single-node and fail-stop: at most one device is unavailable at a time, and an unavailable device stops responding rather than behaving arbitrarily.
- Latency constraint. The end-to-end response time τ, measured from the reception of q to the delivery of y, must not exceed 20 s, formalizing the responsiveness requirement of Section 2.2.
- Scalability. The device set D and the global tool set $T = \bigcup_{i=1}^{M} T_i$ are open: devices and the tools they carry may be added at run time. Adding a device d_i can extend the global tool set T with its tool set T_i and requires no modification of the other devices in D nor of the coordination logic. The inference workload is distributed across the available heterogeneous devices.
- Energy. The energy per inference is measured on the tested hardware in Section 7 and used in Section 8. No fixed numerical energy bound is imposed; the only hard bound is the latency constraint.

Given the device set D, the global tool set T, and a user query q, the system must select the tools in T relevant to q, execute each on a device able to run it, collect the resulting outputs O_q, and return a correct natural language reply y within the latency constraint. Under the stated failure model, the system must continue to operate and preserve the conversational state C upon the failure of any single device.

3. State of the Art

The literature relevant to this work has developed along three main streams:

1. Network architectures and cloud limitations;
2. AI on edge and device constraints;
3. Agentic systems and tool-calling methods.

Even if pairwise intersections between these streams are vastly present in the literature (e.g., AI on Edge to address cloud limitations), the confluence of all three has received limited attention.

3.1. Network Architectures and Cloud Limitations

The limitations of cloud-only architectures for sensor networks, including latency, bandwidth consumption, and reliability under partial connectivity, were articulated in early work by Zhang et al. [17]. Shi et al. subsequently formalized Edge Computing as a paradigm in which computation is moved closer to the data source to mitigate these limitations [18]. Growing literature has refined this position, examining the trade-offs between edge and cloud deployment in terms of scalability, cost, and privacy [5,10]. An important finding is that edge or hybrid networks lower operational costs compared with fully cloud-based architectures, due to less bandwidth consumption [5,10]. Another issue that can affect cloud architectures is the Single Point Of Failure (SPOF) paradigm: in fully cloud-based systems, if the central server goes out of service, the entire network goes offline, which is unacceptable for critical infrastructures [19]. One way of mitigating this is to introduce redundancy, implying duplicate components, such that the failure of one does not stop the system. However, cloud-server redundancy entails even higher costs and tighter scalability limitations. Dependence on external servers can also lead to exposure of sensitive data [20]. Distributed architectures are a possible solution to increase reliability and fault tolerance [5,10], as they are immune to the SPOF problem [20]. While hybrid networks can be promising, Nguyen et al. [20] noted that edge-based IoT systems that still rely on centralized cloud servers inherit the SPOF vulnerability and are still affected by it, even if less critically.

3.2. AI on Edge and Device Constraints

Edge Computing has been widely recognized as a promising solution to most problems affecting cloud-based architectures [6,10,17,18]. Studies comparing edge and cloud architectures reveal that edge computing excels in reducing latency and enhancing data privacy through local processing [5]. Those studies differ on whether there is superior scalability and flexibility in using the cloud [5] or in deploying edge devices [10]. Those differences can be attributed to a high variability in the implementation of such systems (e.g., vendor lock-in, adoption of open source standards, or use of proprietary methods that may or may not grant easy scalability). Within this broader paradigm, video analytics has emerged as a particularly active application domain [6], due to both the bandwidth cost of streaming raw video and the privacy concerns over sharing the data with external servers. Xu et al. [6] provided a comprehensive survey of edge video analytics and identified four architectural tiers. Most importantly, they noted that the combined use of edge devices and dedicated "edge cloud" servers (dedicated servers physically closer to

the end-user, often acting as branches of a main, more powerful cloud, to reduce latency) has been recognized by Microsoft as the only feasible approach for meeting strict real-time requirements for video analytics at scale. Recent image-restoration research has also moved towards novel and more efficient architectures that run on consumer-grade hardware. DSwinIR [21] contains 24 million parameters and, compared with existing competitors, it requires fewer computational resources and achieves lower inference latency with state-of-the-art performance. Specifically, DSwinIR uses 15% fewer FLOPs and runs 22% faster than PromptIR, while requiring approximately 40% fewer FLOPs than DRSformer. The use of novel and relatively small architectures is promising for the future of AI on edge hardware. Historically, edge devices lacked the resources necessary to perform advanced DL analyses and coordination tasks [6], but recent advancements in edge hardware and optimization techniques promise enough computational capabilities to be able to perform most of the analysis offline, directly on device. In January 2026, Raspberry Pi released its Raspberry Pi AI HAT+ 2 accelerator module (based on the Hailo-10H chip), which can achieve 40 TOPS (Tera Operations Per Second) in int 4. This module can be mounted on a Raspberry Pi 5, running LLMs, VLMs, and other DL models on the accelerator without using resources from the CPU [22]. AI model compression techniques such as quantization, pruning, and model distillation demonstrated that advanced AI models can run on resource-constrained devices [23]. A recent study by Kordjukovs and Pau [24] calculated the minimum theoretical CPU throughput in Tera Floating-point Operations Per Second (TFLOPS) and memory bandwidth in GigaByte per second (GBps) required to sustain a target generation rate of 10 Token Per Second (TPS) for 21,039 GPT-Generated Unified Format (GGUF) LLMs checkpoint files, quantized in the format Q4-K-M (4-bit K-block medium). Other studies have explored promising architectures to run VLMs on the edge for video analysis [9]. Recent work focused on optimized architectures for fast Visual Question Answering (VQA) running efficiently on the NPUs of STM32MP257FDK boards (https://github.com/GloriaGio/vqa-stm32mp2 (accessed on 16 July 2026)).

3.3. Agentic Systems and Tool-Calling

In parallel, the rise of capable LLMs has motivated research on agentic systems, in which autonomous components coordinate to solve tasks that exceed the capability of a single model invocation. Sapkota et al. [8] distinguish between AI agents and agentic AI systems. AI agents are characterized as modular, task-specific systems enabled by LLMs and VLMs, building on the foundations of GenAI through tool-calling, prompt engineering, and enhanced reasoning. Agentic AI systems, on the other hand, represent a broader paradigm shift, incorporating multi-agent collaboration, dynamic task decomposition, persistent memory, and coordinated autonomy [8]. Studies on agent registries and discovery mechanisms propose architectures that allow large numbers of autonomous agents to locate and communicate with each other [25]. Google researchers conducted a study on the scaling behavior of multi-agent systems, analyzing how coordination strategy interacts with task structure [26]: this work shows how important it is to use an appropriate architectural design relative to the type of task to perform, to avoid error propagation [26]. Consistently, empirical results indicate that serial multi-agent pipelines tend to amplify the factual errors of individual agents, whereas parallel structures whose outputs are fused by an integrator tend to suppress such error amplification [27]. Most of this literature, however, assumes cloud-hosted models and evaluates coordination in settings where the computation is effectively unbounded. Several open-source projects, such as DeepCamera [15], integrate consumer IP cameras with a centralized private analysis server running LLMs. These efforts reduce cloud dependency in practice but retain a single coordinating node whose failure disables the entire pipeline. In the last months, agentic frameworks have also

developed to fit devices with limited resources: frameworks such as NullClaw eliminate runtime overhead by compiling directly to machine code. NullClaw uses as little as 1 MB of RAM, allowing the framework to boot in under two milliseconds (ms) on microcontrollers (MCUs) and single-board computers [28]. Alternatively, the use of libraries such as langGraph [29] and similar frameworks, implementing custom AI-based agents, allows an even lighter-weight approach to agentic LLMs, since it makes it possible to reduce the length of the context window to the minimum required by the agent to perform actions and reply to the user. Importantly, Anthropic developed the Model Context Protocol (MCP), a JSON-based communication protocol, as an open-source standard for connecting AI applications to other systems (https://modelcontextprotocol.io/docs/learn/architecture (accessed on 16 July 2026)).

Tool-calling is the most important task an LLM in an agentic setting has to perform: a large body of literature has focused on improving tool-calling capabilities in LLMs. Hao et al. [30] developed ToolkenGPT, adding each tool as a token to the vocabulary of LLMs with frozen weights. The model learns only the embeddings for that new tool and avoids full fine-tuning when new tools are added. Qin et al. [13] first fine-tuned an LLM on a dataset of 16,000 real REST APIs, then they used a depth-first search-based decision tree (DFSDT) to support the planning and reasoning ability of the model and finally they also added a neural bi-encoder to perform the retrieval of the top 5 tools with the highest cosine similarity to the request, placing only those in the context of the LLM for it to choose from. Liu et al. [31] point out how in-context tool calls often exceed the context window of LLMs (as also previously noted by Hao et al. [30]) and, since this problem is particularly relevant during multi-step tool calls, they propose to structure tool calls as a directed graph, so that the model follows edges between related tools rather than selecting them from a flat list. Wang et al. [32] use a similar approach to ToolkenGPT; however, they also decided to fine-tune the entire model: this removes the need for few-shot prompting but requires updating all the model's parameters. Lumer et al. [33] take the opposite approach, with no fine-tuning, and instead retrieve relevant tools through a three-step Retrieval-Augmented Generation (RAG) pipeline over enriched tool descriptions, improving recall across pools of up to 4000 tools. They also suggest using an LLM to rewrite the tool description if it is poorly worded or too short. Erdogan et al. [12] developed TinyAgent, fine-tuning 1.1B and 7B models for tool-calling; they also trained a DistilBERT classifier to filter which tools enter the prompt. TinyAgent uses an approach that looks similar to that of Qin et al. [13]. However, the former is focused on edge devices and uses a classifier directly instead of matching embeddings: this allows less flexibility since adding a tool requires retraining the classifier. Shi et al. [14] developed the ToolRet benchmark, showing that Information Retrieval (IR) models tend to perform poorly on tool retrieval tasks. One of the probable reasons is low lexical overlap. Kokane et al. [34] created the Toolscan benchmark to identify error patterns in tool-calling performed by LLMs. Their study also reported that LLMs' tool-calling accuracy drops when function names share tokens with other tools in the available pool. Their similarity metric is a Jaccard-style token overlap on API names: shared tokens (i.e., function words) are stripped, and the remaining overlap is scored. This was done such that near-duplicate names receive high similarity and tools with no token overlap receive low similarity. Higher name overlap correlated with a decrease in the success rate. While relevant, their approach relies solely on lexical names and does not account for descriptions or parameter schemas. Section 9 proposes how these insights could be used, addressing the limitations mentioned here, for an informed design of tool names and descriptions. Kate et al. [35] reported accuracy drops of up to 85% as the context length filled with available tools increases. They also observed up to 91% degradation in

answer retrieval as tool response length increases, varying across tested models. All tested LLMs featured at least 8 billion parameters.

3.4. Intersection: Distributed Systems for Agentic AI

Each of these three categories has matured considerably, but their intersection has received limited attention. Recently, a few studies tackled frameworks of distributed agentic AI on edge devices, showing promising results. Notably, Jabarullah et al. [36] proposed a workload distribution manager (WDM) that balances the workload between the edge network and the cloud, outperforming or rivaling existing cloud-based or edge-only alternatives. However, their approach still relies on a single device on which the WDM is hosted, called the edge-gateway, which could be a SPOF. Abdelmoniem et al. [37] dealt with the challenges of federated learning on heterogeneous edge networks but did not delve into the difficulties of having to coordinate the entire network to execute one task at inference time. Wang et al. [16] noted that the migration of LLM-based agents only requires transferring memory and configuration files rather than the whole code base. They proposed an approach to effectively migrate LLM-based agents across edge servers to reduce latency. Table 1 benchmarks the current work against the most relevant studies. These works were selected for their focus on implementation and their support for local inference to handle privacy-sensitive data.

Table 1. Comparison of representative studies against the key requirements addressed in this work. A check (✓) indicates the requirement is addressed; a cross (×) indicates it is not (or only partially) addressed. It is important to note that each of these works focuses on different use cases: none of them proposes an end-to-end edge-based distributed framework exploiting GenAI at inference time.

Study/System	Local Analysis	No SPOF	Multi-Sensor	GenAI
DeepCamera [15]	✓	×	×	✓
TinyAgent [12]	✓	×	✓	✓
Jabarullah et al. [36]	✓	✓	✓	×
Abdelmoniem et al. [37]	✓	✓	×	×
Wang et al. [16]	✓	✓	×	✓
This work	✓	✓	✓	✓

Local Analysis = supports local analysis; No SPOF = The architecture does not have a Single Point Of Failure; multi-sensor = The proposed system supports multi-sensor inference; GenAI = exploits the use of GenAI.

4. Contributions

The main contributions of this paper are:

- A fully cloudless multi-agent architecture for surveillance that addresses single points of failure through dynamic leader election.
- Proposal of two complementary approaches for distributed tool-calling on heterogeneous edge devices, compatible with varying hardware capabilities.
- An empirical evaluation of the feasibility and efficiency of a prototype of the proposed system, focusing on how tool-calling scales on small LLMs and power consumption.

The proposed architecture integrates state-of-the-art GenAI on edge with smaller DL models (possibly even TinyML models) to analyze sensor inputs directly on the sensing devices while being coordinated by an elected leader agent.

This architecture exploits the heterogeneity of the network to reduce the computational workload on a single agent: it uses different models on different edge devices to execute analysis tasks on multiple devices in parallel. This system can also be operated in a fully decentralized configuration, using a mesh network instead of a central router for IP-based Agent-to-Agent (A2A) communication. This eliminates external network dependencies

(with the exception of an optional additional cloud storage for further redundancy) and removes the central access point as an additional single point of failure, complementing the leader role already addressed through dynamic leader election.

It is important to note that, while the implementation proposed in Section 5.2 is agnostic to the underlying network topology (as long as it is IP-based), all tests were performed while connected to a single access point. The elimination of the access point as a single point of failure is therefore a property of the architecture when deployed on a mesh network. Only the failure of a single device (under the failure mode defined in Section 2.3) has been tested, so resilience has been demonstrated only against a device-level single point of failure.

5. Proposed Approach

As seen in Section 3, edge video analytics work rarely incorporates generative reasoning at the edge and does not focus on performing both reasoning and analysis simultaneously. Work on LLM-based agentic systems rarely constrains itself to resource-limited or disconnected deployment, and the edge computing literature, while supportive of decentralized coordination in principle, offers few end-to-end proposals that combine on-device generative inference, multi-agent orchestration, and graceful degradation under node failure. Taken together, these observations point to a gap in the current landscape. Cloud-based systems are mature but incompatible with strict privacy, latency, and availability requirements. Hybrid systems inherit a softened version of the same SPOF problem, and single-device edge deployments lack the computational resources to run generative reasoning, vision analysis, and multi-sensor inference concurrently. At the same time, the building blocks for a different approach have only recently matured enough to be combined: lightweight agentic runtimes, on-device quantized DL models (including LLMs and VLMs), and affordable edge accelerators.

This section is divided into the following subsections:

1. Architecture details Section 5.1;
2. Software implementation Section 5.2;
3. Hardware devices Section 5.3;
4. Model Selection for leader capabilities Section 5.4.

5.1. Architecture Details

This work proposes an architecture comprising different agents connected to each other, Peer to Peer (P2P)–equivalently A2A. On each device, there can be one or more "agents". An agent is defined by an agent-ID in the network and has its associated functions. Agents can be sensing agents or leader agents: a device can host both a sensing agent and a leader agent at the same time.

There is exactly one leader agent in the network at any time. Let $D_{\text{cand}} \subseteq D$ be the set of candidate leaders, that is, the devices connected to the network that can run a leader agent. Each device $d \in D_{\text{cand}}$ is assigned a device score $s(d)$, computed from the presence of an AI accelerator (NPU, Hailo, or dedicated GPU), the available RAM, the CPU compute in TFLOPS (Tera Floating-point Operations Per Second), and the memory bandwidth in GB/s (GigaBytes per second). The score function used is defined by Equation (2) in Section 5.2. The leader, denoted by d_L, is the candidate with the highest score, according to Equation (1).

$$d_L = \arg \max_{d \in D_{\text{cand}}} s(d) \tag{1}$$

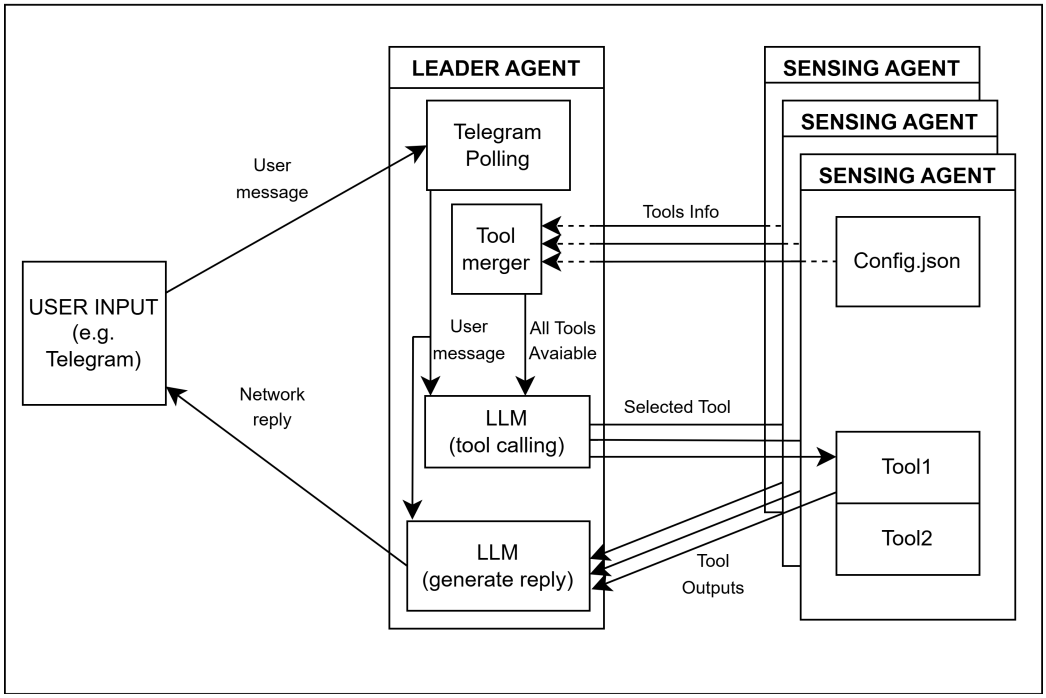
Ties are broken deterministically by agent-ID. The leader can be re-elected whenever D_{cand} changes. Concretely, leader re-election happens in two cases:

1. The leader disconnects: the current leader is removed from D_{cand}, and the remaining candidate with the highest score becomes the new leader.
2. A more powerful device joins: if a device joins D_{cand} and it has a score higher than that of the current leader, it becomes the new leader.

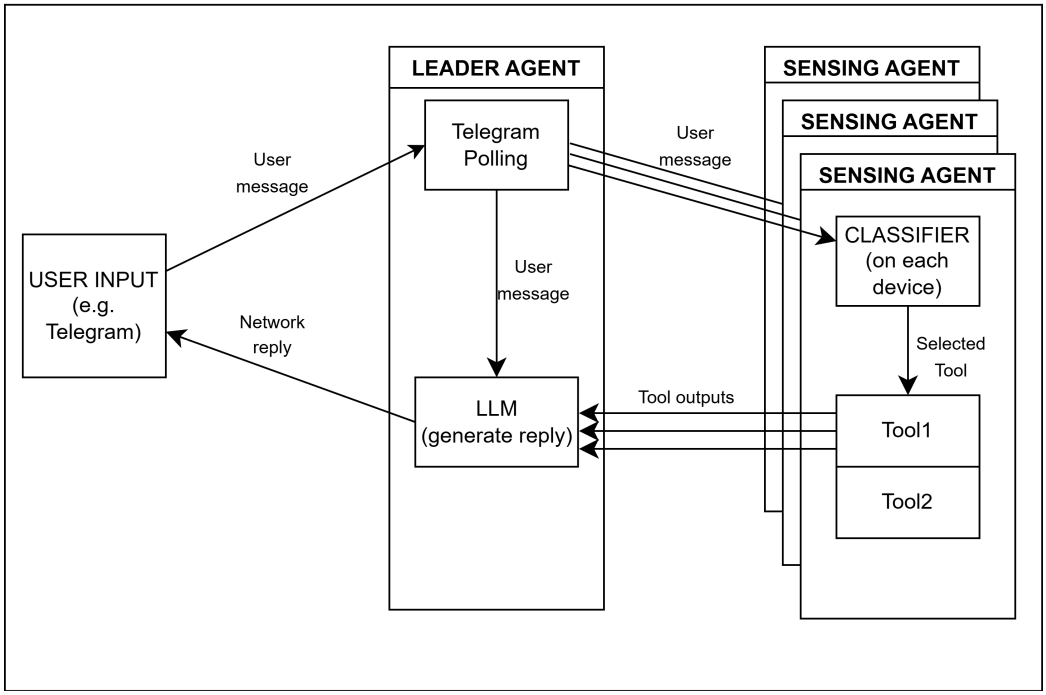
The new leader restores the conversational state C from its replica, kept on the device with the second-highest score, so that the service continues. When the new leader runs a less capable model than the previous one, operation continues with graceful degradation.

First, the program receives the user query (e.g., through a Telegram bot or another interface). Two complementary approaches are considered for interpreting the query, executing the appropriate tools on the sensing devices, and generating a natural language reply. Approach 1 is evaluated in full, with tool-calling accuracy and power-consumption measurements, whereas Approach 2 is assessed only as a preliminary proof of concept. The two approaches are illustrated in Figure 1. Approach 1 uses one LLM on the leader agent to interpret the user query and select the best tool, while Approach 2 uses multiple re-rankers (or classifiers) on the sensing agents to select the tools. In both of the approaches, the system relies on one leader agent and a set of sensing agents connected through a local IP-based communication layer. Both of them use an LLM on the leader agent to forge a final reply in natural language, given the tool results and the original user query. Those approaches can potentially run concurrently on the same network.

1. Approach 1 (Figure 1a) is based on distributed tool-calling managed by the leader. The LLM running on the leader interprets the user request and selects the tool that best matches the user query among those exposed by the sensing agents (each sensing agent can expose more than 1 tool). This approach is the one on which this work focuses the most because it is universally applicable to all sensing devices: any agent can expose one or more tools to the leader through the network interface, without requiring a local classifier. Functions called through tool-calling can also present arguments that the LLM can flexibly fill. The criteria used in this paper for the selection of the LLM, based on the hardware available, are discussed in Section 5.4. In order to avoid passing duplicate tools to the LLM (e.g., if two sensing agents can execute the same functions), a tool merger block is introduced, which eliminates duplicates. When the LLM chooses a tool, any sensing agent in the network that can execute it does so, then sends the results back to the leader.
2. Approach 2 (Figure 1b) forwards the user request directly to the sensing agents that support this approach. Those sensing agents use an onboard classifier (or IR system) to locally decide whether the query matches one of their available tools, completely bypassing the LLM's tool-calling: if the query matches a tool, it executes it. As previously mentioned, an alternative to a custom classifier can also be a re-ranker, using bi-encoders or cross-encoders: this essentially transforms tool-calling into an IR problem. The main drawback of this IR-based (or classifier-based) approach is that it cannot perform tool-calling with parameters without resorting to more advanced techniques. Tool selection on the edge devices can improve architectural scalability and flexibility, but it requires an additional model (IR or Classifier) on each sensing agent. Thus, it requires more powerful sensing devices and makes the addition of new ones not trivial. The accuracy of one IR-based approach is benchmarked in this work only as a preliminary proof of concept.



(**a**) Approach 1



(**b**) Approach 2

Figure 1. The two approaches proposed by this work are shown as block diagrams of the leader agent and sensing agent components and the message flow between them. In both approaches, the leader agent polls the user input, and the LLM on the leader agent generates the final natural language reply based on the sensing agents' tool outputs and the original user query. The final reply is sent back to the user. In Approach 1 (**a**): (i) each sensing agent forwards its available tools, stored in a local file, config.json, to the leader; (ii) the leader merges the tool lists received from the sensing agents (removes duplicates); (iii) the LLM on the leader agent selects a tool and calls it; (iv) the sensing agents that have that tool execute it and return the output to the leader. In Approach 2 (**b**): (i) the leader agent forwards the user message directly to the sensing agents; (ii) each sensing agent runs a local classifier that selects the matching tool and executes it; (iii) the sensing agent returns the tool output to the leader.

These two approaches are not intended to be mutually exclusive. This design proposes that they run concurrently: different sensing agents can support different approaches on the same network. Figure 2 presents a simplified flowchart of the system workflow, while Appendix J (Figures A6–A12) provides more detailed flowcharts. For Approach-1 devices, the leader handles request interpretation and tool aggregation. For Approach-2 devices, it forwards the user query directly to the sensing agents, which perform local classification to decide which tool to execute. Tool results are always sent back to the leader, which generates a natural language reply.

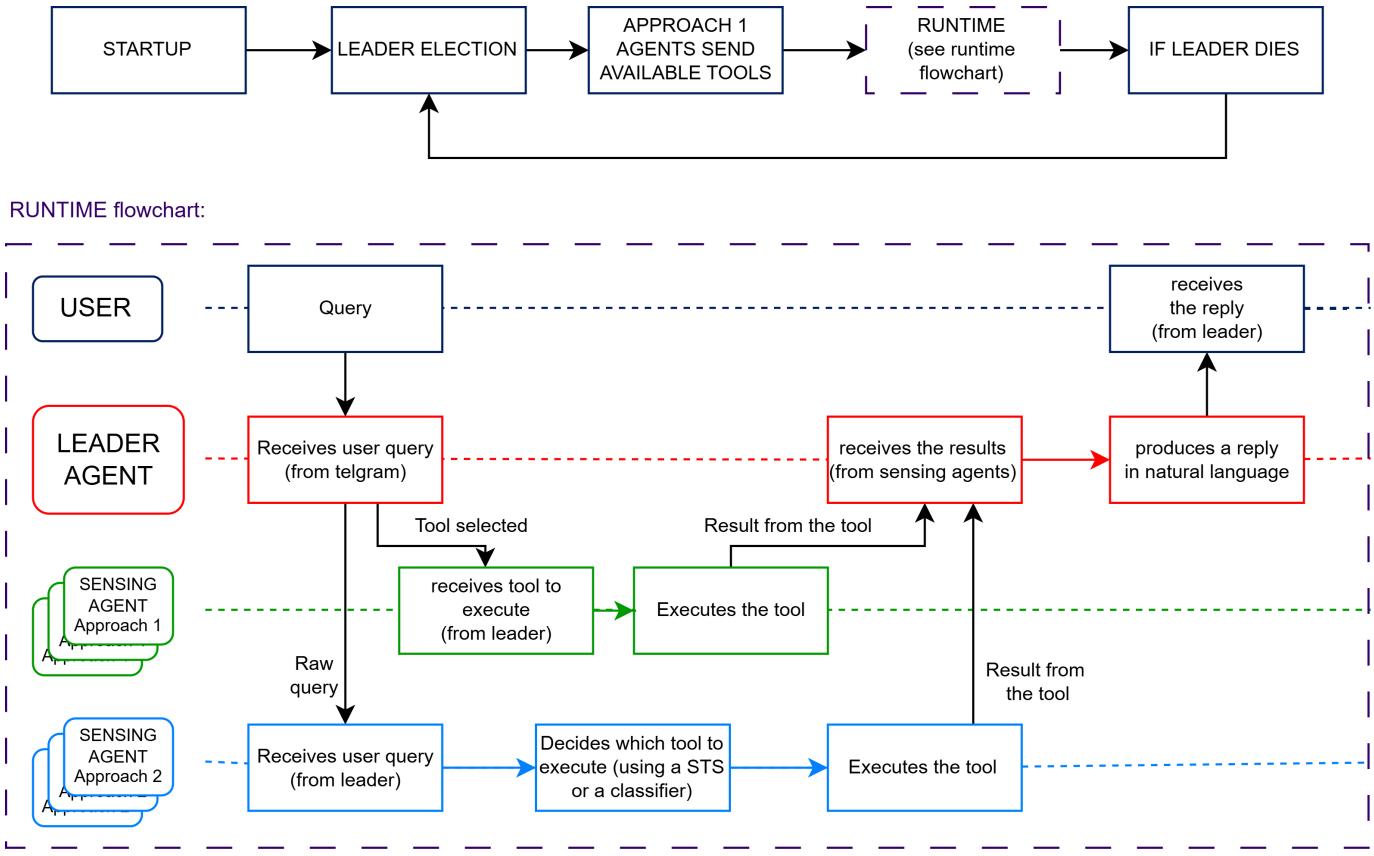


Figure 2. High-level overview of the system workflow following initialization, illustrating the concurrent execution of Approach 1 and Approach 2.

5.2. Software Implementation

This section describes the implementation of the architecture presented in Section 5.1. The code is available on https://github.com/Gigizap/heterogeneous-edge-ai-network (accessed on 16 July 2026). The software stack has been implemented in Python, and it comprises the following components:

- Device Manager (`main.py`): the main orchestrator of the device, runs on its own dedicated thread. It runs the startup code (`ElectionLogic/identity.py`), reads the device configuration, manages the leader election process (`ElectionLogic/election.py`), activates and deactivates agents on the device, and backs up data from the current leader to ensure continuous operation in the event of a failure.
- Election Logic (`ElectionLogic/election.py`): contains the modules used by main.py to manage the election processes.
- Election Logic (`ElectionLogic/score.py`): computes the score of the device; the score is a heuristic that represents computational capabilities.

- Leader Agent Logic (`LeaderLogic/run_leader.py`): handles leader orchestration and LLM inference, transforming user input into commands for the network to execute; runs on its own dedicated thread.
- Leader Agent Logic (`LeaderLogic/backup_manager.py`): manages the replication of the conversational state on the second most powerful device with leader capabilities (e.g., on the second most powerful candidate).
- Sensing Agent Logic (`SensingLogic/run_client.py`): capabilities are defined by a JSON file listing the available tools and the mapping between tool names and the functions to execute; AI models run locally on the device, and the sensing agent replies to commands issued by the leader; runs on its own dedicated thread.
- Communication Layer: manages IP-based communication and works plug-and-play with a central router (thanks to the Dynamic Host Configuration Protocol–DHCP). It can also operate over mesh networks or direct LAN connections, though these modes require static IP assignment. Inspired by MCP, the payload is in JSON; the payloads implemented are visible in Appendix A in Table A1. Runs asynchronously via `asyncio`, enabling non-blocking message handling across the network. Each agent has its own "transport" object that it uses to send and receive messages, as seen in Figure 3.

When setting up a new device, an agent-ID is required to the user: the device listens for other devices and prevents the user from inputting an ID already in use. Each sensing agent can execute some functions, called tools or skills, and return the results to the leader. The execution of a skill can involve the call of a local DL model or just the reading of sensor data (e.g., temperature), depending on the sensing agent configuration.

Each device that can run a leader agent computes a scalar device-score $s(d)$, used to elect the active leader agent in the network. The score is a heuristic: a weighted sum of three hardware terms, each normalised against reference values representative of a mid-range PC ($\mathrm{REF}_{\mathrm{TFLOPS}} = 1.0$, $\mathrm{REF}_{\mathrm{BW}} = 25\,\mathrm{GB/s}$ and $\mathrm{REF}_{\mathrm{RAM}} = 8\,\mathrm{GB}$), and scaled by an accelerator multiplier. The score of a device d is

$$s(d) = \mathrm{round}\left[1000\, a \left(w_B \frac{B}{\mathrm{REF}_{\mathrm{BW}}} + w_C \frac{C}{\mathrm{REF}_{\mathrm{TFLOPS}}} + w_R \frac{R}{\mathrm{REF}_{\mathrm{RAM}}} \right)\right], \tag{2}$$

where B is the memory bandwidth (GB/s), C is the CPU compute (TFLOPS) and R is the available RAM (GB). These three quantities correspond to the hardware parameters reported by Kordjukovs et al. [24] to determine the feasibility of on-device language-model inference. The weights are $w_B = 0.45$, $w_C = 0.35$, and $w_R = 0.20$, with $w_B + w_C + w_R = 1$; bandwidth receives the largest weight as it is the primary bottleneck for the target workload. The term a is the accelerator multiplier: $a = 1.4$ when the device has an AI accelerator (NPU, Hailo, or dedicated GPU) and $a = 1$ otherwise. The factor 1000 scales the result to integer scores. The scoring function serves only to rank leader-capable devices in the proposed implementation, and it is reported solely for reproducibility. This study did not verify whether it generalizes to other devices than the tested ones, and no such claim is made. Conversation history backed up between the leader and the second most powerful device, described in Section 5.1, is retained with no defined deletion policy.

Each sensing agent compatible with approach 1 exposes its skills as an OpenAI-compatible JSON for tool-calling to the leader. Sensing agents compatible with approach 2 instead do not forward their tool list to the leader because the tool selection is performed by the sensing agent itself. The proposed system was designed such that the function name exposed to the LLM (for approach 1) or to the IR system (approach 2) can differ from the actual function name in the codebase, providing flexibility in how tool calls are implemented.

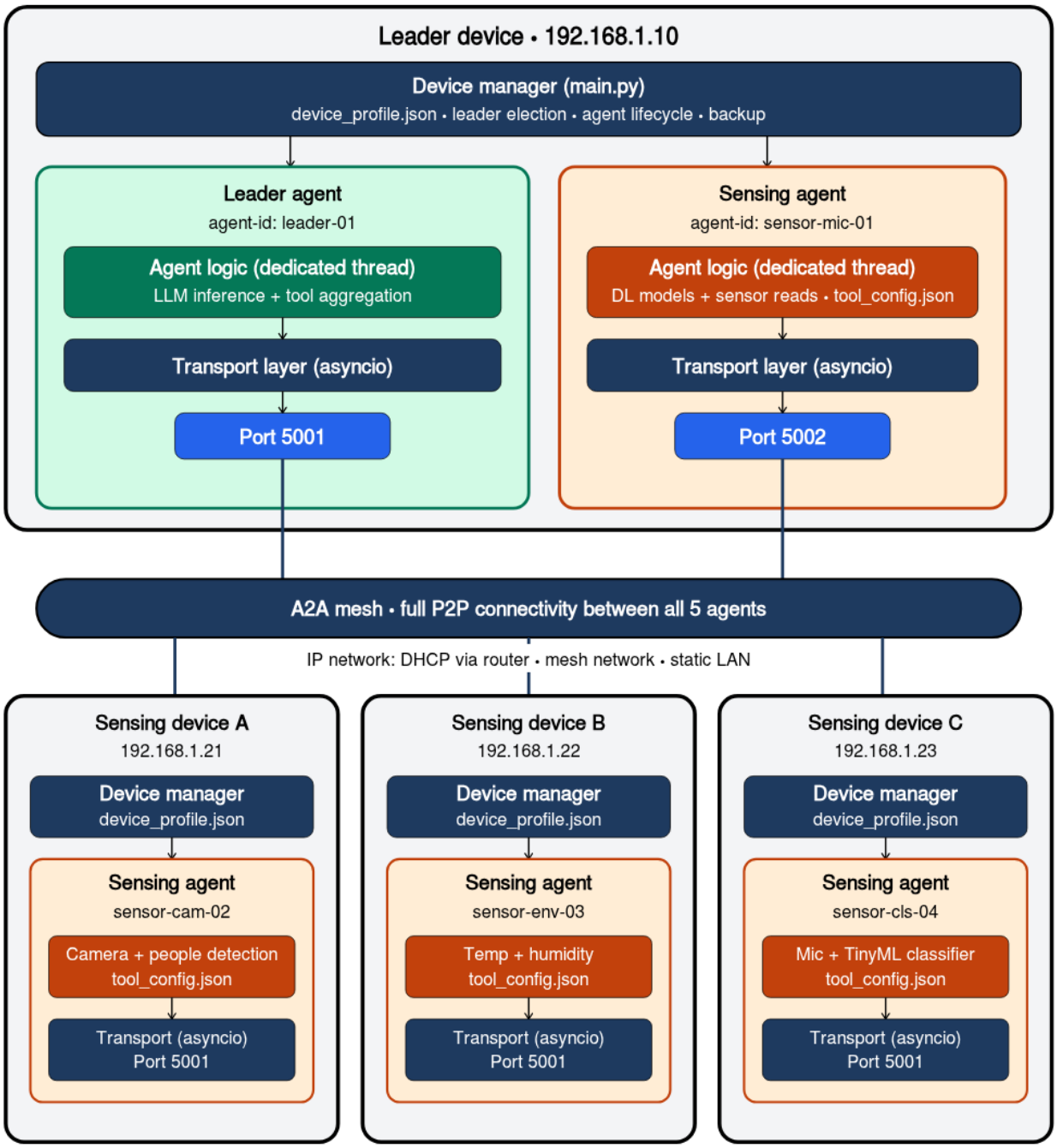


Figure 3. This picture shows the main components of each agent and the agent-to-agent connections. The Device manager component loads the agent-present from a device_profile.json file. Sensing Agent logic defines the LLM-facing tools in a tool_config.json file; this file also points at the Python code containing the actual implementation of said tools. Each agent relies on a transport layer to manage network connections; each transport layer owns its port. Leader Agent logic runs the LLM, manages tool aggregation, and distributes tool-calling. More than 1 agent can live on the same device, on different ports. Every agent stores all the other agent-IDs together with the respective IP and port. The modular architecture makes the addition of sensing agents trivial.

On the communication layer, a custom IP-based protocol enables discovery, broadcast, direct communication, and result forwarding among agents. The A2A connections are represented in Figure 3. The physical device connections are represented in Figure 4. Each Agent holds in memory a list of all the other peers in the network in its "transport" object.

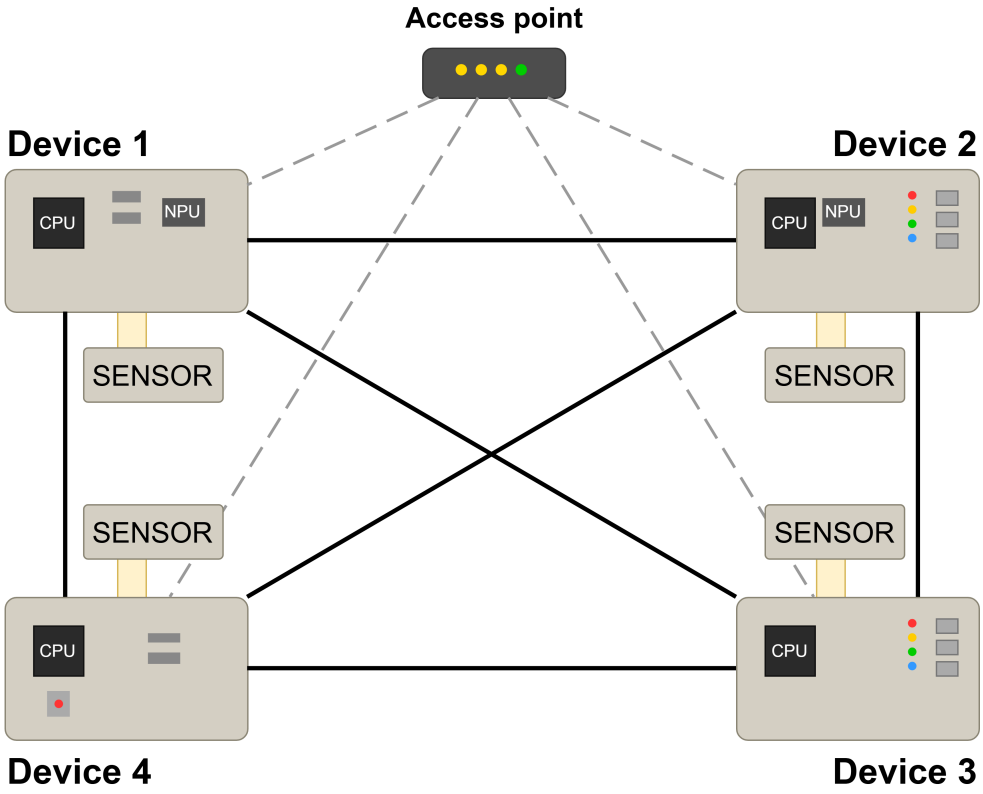


Figure 4. Connections between devices in the network. Any IP-capable device running the software stack can participate. Devices discover one another within a shared broadcast domain and establish direct peer-to-peer (P2P) connections. IP assignment is handled by an optional DHCP/infrastructure layer; in a mesh deployment, this is replaced by static or MAC-derived addressing. In the depicted example, 4 generic devices are equipped with generic sensors. Devices 1 and 2 are equipped with an NPU, while devices 3 and 4 only have CPUs.

The current implementation delegates most security guarantees to the network perimeter rather than to the agent-to-agent protocol itself. In the tested deployment, the access point admits only devices provisioned in advance by the network administrator, and a firewall rule blocks all traffic except connections to the Telegram Bot API. This mitigates the risk of an external attacker reaching the agents directly since no port on any device is reachable from outside the local network. These controls do not extend inside the perimeter: the communication layer performs no per-message authentication or authorization, transmits JSON payloads unencrypted, and does not restrict which tools an agent may invoke on another. Consequently, any device admitted to the network is implicitly trusted, with no mechanism to detect a compromised or malicious agent once it has joined. This trust model is considered sufficient for the single-household deployment evaluated here, where the set of provisioned devices is small and administrator-controlled. Extending it into a complete authentication, authorization, and encryption framework is outside the scope of this work, whose objective is to validate the feasibility of the proposed architecture rather than to deliver a production-ready security system. A mesh-network deployment, discussed as an alternative in Section 4, would require additional work to reproduce an equivalent admission-control point, since it lacks a single access point at which perimeter filtering could be applied. Limitations and possible remedies are discussed in Section 9.5.

This design supports mixed hardware deployments, graceful degradation, and, potentially, concurrent execution of the two approaches within the same network. That said, this work focuses primarily on evaluating Approach 1 because it requires less powerful sensing devices and thus can also be integrated with very lightweight sensing devices.

This paper conducts tool-calling benchmarks and power-consumption measurements for Approach 1, while for Approach 2, only a single IR model is benchmarked as a proof of concept to validate the architecture's flexibility.

5.3. Hardware Devices

The tested hardware layer comprises the following components:

- A Raspberry Pi 5, with 16 GB of RAM.
- A Raspberry Pi AI HAT + 2, based on a Hailo-10H accelerator, with 8 GB of dedicated RAM (hereafter referred to as "Hailo") attached to the Raspberry Pi 5.
- An STM32MP257FDK board with 4 GB of RAM.
- A Raspberry Pi Camera Module 2 (hereafter referred to as "PiCamera") attached to the Raspberry Pi 5.
- A B-CAMS-IMX camera module attached to the STM32MP257FDK.

This means the tested setup has two heterogeneous nodes:

1. Device 1: Raspberry Pi 5 + Hailo + PiCamera.
2. Device 2: STM32MP257FDK + B-CAMS-IMX camera module.

Respectively, they are Device 1 and Device 2 shown in Figure 5.

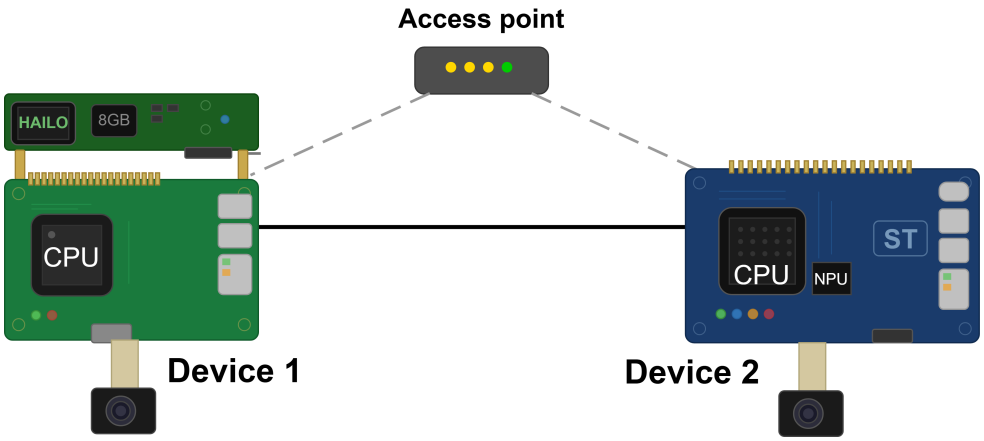


Figure 5. Representation of the two tested devices in this study. (i) Device 1 is a Raspberry Pi 5 equipped with a Raspberry AI HAT +2. (ii) Device 2 is a STM32MP257FDK board. The devices are fitted with camera modules.

Each device executes one or more agents according to their respective capabilities. The Raspberry Pi 5, equipped with the Hailo accelerator, acts as the most powerful node (on the tested setup): it runs LLMs on the Hailo or, through `llama.cpp`, on its CPU. It can also serve as a sensing device, running YOLO models or a VLM to analyze its video feed. The STM32MP257FDK serves mainly as a sensing agent, executing optimized DL models for image classification on its NPU for local analysis and sensing; it can also take over as the leader if the most powerful node fails, running small LLMs through `llama.cpp` on its CPU.

5.4. LLM Selection for Leader Capabilities

This subsection motivates the choice of `functiongemma:270m` for the STM32MP257FDK board and qwen3:1.7b for the Raspberry Pi 5 + Hailo node.

A device in the network has leader capabilities if its hardware is able to run LLMs while maintaining real-time question-answering (Q&A) and it is configured to do so. The LLM acting as the backbone for the leader agent is chosen by filtering first for hardware compatibility, then for function-calling accuracy. The procedure is applied independently to each leader-capable device, since the hardware-compatibility filter depends on the inference backend available on the target device.

5.4.1. Hardware Compatibility Filter

Since the STM32MP257FDK board runs inference on its CPU while the Raspberry Pi 5 node can offload to the Hailo accelerator, the two platforms are filtered against different criteria: an empirical throughput threshold for the STM32MP257FDK, and accelerator compatibility for the Hailo.

Candidate Models on STM32MP257FDK

An initial set of 25 models was benchmarked for inference speed; the results are reported in Table 2. To preserve interactive Q&A, the candidate set of LLMs for this board, $\mathcal{M}^{*}_{\text{STM32}}$, is restricted to models exceeding 5 TPS.

$$\mathcal{M}^{*}_{\text{STM32}} = \{\, \texttt{gemma3:270m}, \texttt{smollm2:135m}, \texttt{stories110m}, \texttt{stories42M}, \texttt{stories15M}, \texttt{stories260K} \,\}.$$

Note that `gemma3:270m` shares its architecture with `functiongemma:270m`, so the two models exhibit the same inference speed; the throughput measured for `gemma3:270m` therefore applies to the function-calling variant.

Table 2. Benchmarking on STM32MP257FDK (STM32MP257FDK, OpenLinux on Dual Arm A35 1.5 GHz, 4 GB RAM).

Framework	SLM	Eval Rate [TPS]
llama.c	stories260K	889.95
llama.c	stories15M	46.21
llama.c	stories42M	18.17
llama.c	stories110m	7.2
ollama 0.12.11	smollm2:135m	5.75
ollama 0.12.11	gemma3:270m	5.5
ollama 0.12.11	sam860/LFM2:350m	4.69
llama.cpp-6589	LFM2-350M-ENJP	3.99
llama.cpp-6589	LFM2-350M-Math	3.74
ollama 0.12.11	qwen:0.5B	3
ollama 0.12.11	sam860/LFM2:700m	2.34
llama.cpp	BitNet-25 2B I2_S 2bpw ternary pp512	1.68
ollama 0.12.11	llama3.2:1b	1.47
ollama 0.12.11	gemma3:1b	1.38
ollama 0.12.11	granite4:350m	1.24
llama.cpp-6589	LFM2-1.2B-Extract	1.25
llama.cpp	BitNet-25 2B I2_S 2bpw ternary tg128	1.05
ollama 0.12.11	deepseek-r1:1.5b	1.01
ollama 0.12.11	qwen3-vl:2b , smolvlm2:2.2b	0.69–0.86
ollama 0.12.11	exaone-deep:2.4b	0.63
ollama 0.12.11	granite3.3:2b	0.62
ollama 0.12.11	tomasmcm/openelm:3b-intruct-q5_K_M	0.44
ollama 0.12.11	phi4-mini:3.8b	0.42
ollama 0.12.11	phi3:3.8b	0.32
llama-cpp	LFM2.5-Audio-1.5B-Q8_0.gguf (ollama 1.51)	0.19–0.22

Models above the dotted line ran at more than 5 TPS in this preliminary test.

Candidate Models on Hailo and Raspberry Pi 5

For the Raspberry Pi 5 node, the candidate set is bounded not by throughput but by accelerator support: only models with a compatible Hailo compilation (.hef files) are admissible. All such models also run on the CPU through `llama.cpp`, so the selected model is, by construction, deployable on both backends, allowing the two to be evaluated on the

same model in the subsequent on-target analysis (Section 7). This first selection returns the set $\mathcal{M}^*_{\text{Hailo}}$ calculated as follows:

$$\mathcal{M}^*_{\text{Hailo}} = \{\texttt{DeepSeek-R1-Distill:1.5b}, \texttt{Llama3.2:1b}, \texttt{Qwen2.5-Instruct:1.5b}, \texttt{Qwen2.5-Coder:1.5b}, \texttt{Qwen2:1.5b}, \texttt{Qwen3:1.7b}\}.$$

5.4.2. Score-Based Selection

The selection below is applied independently to each device $d \in \{\text{STM32}, \text{Hailo}\}$, operating on the corresponding candidate set $\mathcal{M}^*_d$. In what follows, $\mathcal{M}^*$ denotes $\mathcal{M}^*_d$ for the device under consideration. Let σ_m denote the Live (AST) Single Simple Tool-Calling score of model m on the Berkeley Function Calling Leaderboard (BFCL) [38]. Single Simple Tool-Calling is chosen because small models degrade significantly on multi-tool and nested calling tasks, making it the most relevant indicator of reliable function-calling capability under the hardware constraints. Live AST is more representative of accuracy on unseen data.

Since the BFCL leaderboard does not cover every candidate model, models without a score are discarded. `functiongemma:270m`'s BFCL score is sourced from its Ollama model page. This yields a subset of the original candidate models $\hat{\mathcal{M}}$, defined as follows:

$$\hat{\mathcal{M}} = \{m \in \mathcal{M}^* \mid \sigma_m \text{ is available}\}.$$

The selected model m^* is the one in $\hat{\mathcal{M}}$ with the highest score:

$$m^* = \arg\max_{m \in \hat{\mathcal{M}}} \sigma_m.$$

The score σ_m serves as a selection metric because it is a general indicator of function-calling capability. It is not necessarily a domain-specific predictor; the selected model is subsequently evaluated on the target scenario in Section 6.

6. Model Analysis

This section focuses on evaluating tool-calling accuracy of both approaches. It calculates the accuracy of the 2 proposed approaches using a set of 24 tools that are representative of the capabilities expected from an agent operating in a smart-home environment. Section 6.1 describes the setup and the methodology used to ensure reproducible results, and Section 6.2 states the results.

6.1. Model Analysis Setup

Accuracy benchmarks were run on a computer equipped with a GPU 3080 laptop with 16GB of VRAM, 32 GB of RAM DDR4, Intel i9 12900H, using the same model files deployed on edge. The code used for computing the benchmarks, the dataset, and all the model's outputs are available in the project GitHub repository under the "BENCHMARK CODE/ACCURACY" folder at https://github.com/Gigizap/heterogeneous-edge-ai-network (accessed on 16 July 2026). LLMs were run on llama-cpp-python; two chat_handlers were created for the LLMs tested to ensure the correct use of the model template and coherent tool parsing. More details on the chat handlers are available in Appendix F. The LLMs were evaluated under a single, fixed decoding policy. Greedy decoding is used throughout (temperature = 0.0) with a fixed random seed and a repetition penalty of 1.1, the latter mitigating the degenerate looping that greedy decoding can induce

in hybrid-reasoning models. A context window of 4096 tokens is used for every model. For the Qwen3-1.7B model, both reasoning and nonreasoning modes are tested.

In order to benchmark the proposed system on the relevant domain (surveillance-related smart-home devices), a set of 24 relevant tools was handcrafted. The set of tools has been realized based on the video analysis capabilities of the available Deep Learning models on STM32MP257FDK boards (those models have been downloaded from the package X_LINUX_AI_TOOL, this is the link at the X Linux Ai Tool suite documentation: https://wiki.st.com/stm32mpu/wiki/X-LINUX-AI_tool_suite (accessed on 16 July 2026)) available on OpenSTLinux. It was further augmented with functions related to audio analysis, VLM capabilities, and relevant smart-home sensors in order to test tool-calling capabilities in such a scenario.

For a query q, let $t^*(q) \in T \cup \{\varnothing\}$ denote the ground-truth tool ($t^*(q) = \varnothing$ if abstention is expected) and $\theta^*(q)$ its ground-truth parameters. Let $\hat{t}(q)$, $\hat{\theta}(q)$ and $\hat{y}(q)$ denote, respectively, the tool, parameters and natural language reply predicted by the model under test, and let $\mathcal{Q}$ be the set of evaluated queries. The three reported metrics are defined as follows.

Selection accuracy is the fraction of queries for which the predicted tool name matches the ground truth, ignoring parameters:

$$\mathrm{Acc}_{\mathrm{sel}} = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \mathbb{1}\big[\hat{t}(q) = t^*(q)\big] \tag{3}$$

Tool-call accuracy is the fraction of queries for which both the predicted tool name and all predicted parameters match the ground truth:

$$\mathrm{Acc}_{\mathrm{call}} = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \mathbb{1}\big[\hat{t}(q) = t^*(q) \wedge \hat{\theta}(q) = \theta^*(q)\big] \tag{4}$$

Summarization grade is the average soft label from 1 to 10 assigned by GLM-4.7-flash, given the tool outputs O_q and the predicted reply:

$$\bar{g} = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} G\big(q, O_q, \hat{y}(q)\big) \tag{5}$$

where $G(\cdot) \in \{1, \ldots, 10\}$, following the grading procedure detailed in Appendix D.

The evaluation is organized into three benchmarks:

- Benchmark A measures single-tool dispatch over the full tool set, with the candidate pool sampled at sizes of 4, 8, 12, 16, 20, and 24 tools; for each query, the ground-truth tool (when one exists) is guaranteed to be present in the pool, and the remaining slots are filled by uniform random sampling under the fixed seed. Both selection accuracy (Equation (3)) and tool-call accuracy (Equation (4)) are reported, separately for the subset of queries that expect a tool versus those that expect abstention.
- Benchmark B restricts the possible tools to zero-parameter tools and compares the language models against the Jina reranker (jinaai/jina-reranker-v2-base-multilingual (Repository of the reranker: https://huggingface.co/jinaai/jina-reranker-v2-base-multilingual (Last Accessed on 9 June 2026))) on selection accuracy alone (Equation (3)), with pools of 4, 8, 12, and 16 tools.
- Benchmark C measures summarization from tool results: each model is shown 1, 5, 10, 15, and 20 function replies. The summarization grade (Equation (5)) and mean generated-token counts are recorded, using a soft label from 1 to 10 produced by GLM-4.7-flash rating the reply given the function replies. The grading procedure is shown in detail in Appendix D.

Tool-call generation is capped at 256 tokens for dispatch and 1024 tokens for summarization. Parameter matching is order-independent over keys and applies light normalization (numeric coercion and case-insensitive, whitespace-trimmed string comparison). All pool assignments and reply samples are cached to disk and reused across runs to hold the evaluation set fixed across models.

The tools cover face enrollment and recognition, object detection and counting, scene description, OCR, motion and presence monitoring, audio classification and transcription, and device diagnostics. The complete list is provided in Appendix B. Due to the findings of Shi et al. from the ToolRet benchmark [14] discussed in Section 3, the tool descriptions are expanded with 3 query examples to enhance lexical overlap. The model chosen to test approach 2 is the best model according to the ToolRet leaderboard as of 28 May 2026: jinaai/jina-reranker-v2-base-multilingual 278 million parameters (later referred to as 'Jina reranker').

The LLMs tested are chosen according to the available hardware, following the criteria defined in Section 5.4. This was done to grant an end-to-end evaluation of the proposed architecture.

The setup is described further in the following two sections:

1. Section 6.1.1 describes how the dataset containing the user queries was created. Those are the queries given on which the models were tested on tool-calling (given the aforementioned tools).
2. Section 6.1.2 describes how the three benchmarks (A, B, C) were conducted.

6.1.1. Dataset Creation

The dataset has 500 queries, 480 of which are associated with one of the 24 handcrafted tools as ground truth, and 20 have no tool associated. The sentences were generated by Claude Opus 4.7 [28 May 2026] with adaptive thinking enabled.

To generate the first 480 sentences, Claude Opus 4.7 was given the 24 handcrafted tools in an OpenAI-compatible format and instructed to write 20 queries for each tool such that every query has exactly one ground-truth tool associated and together with the correct parameters required. The model was instructed to keep them relatively short and natural (it was explicitly specified to make some of them slightly imprecise, as a real user would phrase them). The full generation prompt is provided in Appendix C. Another 20 queries were generated to have none of the 24 tools as the ground truth (such that no tools are available in the pool to satisfy those 20 queries). This was also done to calculate "abstention accuracy".

The queries are synthetic (LLM-generated), not sampled from real user logs, so they may not reflect the full diversity of real-world phrasing, error patterns, or task distributions. Synthetic queries may instead be biased towards the generating model's own stylistic and lexical conventions. Additionally, the "slightly imprecise" instruction was specified in the prompt rather than empirically derived from real user imprecision. These limitations are discussed, along with possible mitigation strategies, in Section 9.

6.1.2. Benchmark Design

Tool Pool Construction

Each test question is paired with a tool pool of size n, constructed deterministically as follows. If the question has a ground-truth tool, that tool is always included such that the correct choice is available; the remaining $n-1$ slots are filled by sampling without replacement from the other tools between the available tools (mentioned as "tool universe" in the following pseudocode). The resulting subset is then shuffled to prevent position bias. For questions where no tool-call is expected, the pool is a plain random sample of

size n. All pools are pre-generated with a fixed random seed before the benchmark loop, guaranteeing that every LLM and the re-ranker sees the identical tool set for a given (pool size, question) pair.

Two different pool files are generated because benchmarks A and B operate on different tool universes: Benchmark A uses $\mathcal{U}_{24}$ (all 24 tools) with pool sizes $\{4, 8, 12, 16, 20, 24\}$ evaluated on the full question set; Benchmark B uses $\mathcal{U}_{16}$ (the 16 zero-parameter tools only) with pool sizes $\{4, 8, 12, 16\}$ evaluated only on questions whose ground-truth tool is in $\mathcal{U}_{16}$.

Algorithm 1 summarizes this pool construction procedure: for every pool size and question, it inserts the ground-truth tool (when one exists), fills the rest by sampling distractors, shuffles, and saves the pool.

Algorithm 1 Tool-pool construction

Input: Universe $\mathcal{U}$, dataset $\mathcal{D}$, pool sizes $\mathcal{N}$, seed s
$rng \leftarrow \texttt{Random}(s)$
for $n \in \mathcal{N}$ **do**
  for $q \in \mathcal{D}$ **do**
    if $q.gt \neq \texttt{None}$ **then**
      $required \leftarrow \{t \in \mathcal{U} \mid t.name = q.gt.name\}$
      $others \leftarrow \mathcal{U} \setminus required$
      $pool \leftarrow required \cup rng.\textsc{Sample}(others,\ n-1)$
    else
      $pool \leftarrow rng.\textsc{Sample}(\mathcal{U},\ n)$
    end if
    $rng.\textsc{Shuffle}(pool)$
    $\textsc{SavePool}(n,\ q.id,\ pool)$
  end for
end for

It is important to note that, since the benchmarks A and B use different tool universes, they also use different question subsets from the dataset. Thus, the LLM accuracy numbers from Benchmark A and Benchmark B are not directly comparable. Within Benchmark B, however, every LLM is evaluated on the same questions and the same pools that are fed to the re-ranker. This makes the LLM-vs-reranker comparison rigorous. Within Benchmark A the comparison between LLMs is exhaustive, since it covers all the available tools.

Benchmark A: LLM Tool-Calling on the Full Tool Set

This first benchmark focuses on evaluating Approach 1 on the complete set of 24 tools. Each LLM is prompted with the user query and the current tool pool; it must emit exactly one tool call. The predicted tool name and arguments are parsed from the response and compared against the ground truth. Two metrics are reported:

1. selection accuracy: fraction of questions where the predicted tool name matches the ground truth.
2. tool-call accuracy: fraction of questions where both the tool name and all required parameters match (case-insensitive, numeric tolerance ± 0.01)

Both metrics are measured at each pool size $\{4, 8, 12, 16, 20, 24\}$ to reveal how accuracy degrades as the number of distractor tools grows.

Algorithm 2 details this loop: each model dispatches one tool-call per question and pool, then selection and tool-call accuracies are recorded per pool size.

Algorithm 2 Benchmark A: LLM tool-calling on the full tool set

Input: Models $\mathcal{M}$, dataset $\mathcal{D}$, pools $\mathcal{P}$, sizes $\{4, 8, 12, 16, 20, 24\}$
for $model \in \mathcal{M}$ **do**
  for $n \in \{4, 8, 12, 16, 20, 24\}$ **do**
    $sel_ok \leftarrow 0, \quad tc_ok \leftarrow 0$
    for $q \in \mathcal{D}$ **do**
      $pool \leftarrow \mathcal{P}[n][q.id]$
      $pred_name, pred_args \leftarrow model.\textsc{Dispatch}(q.prompt, pool)$
      if $pred_name = q.gt.name$ **then**
        $sel_ok \leftarrow sel_ok + 1$
        if $\textsc{ParamsMatch}(q.gt.args, pred_args)$ **then**
          $tc_ok \leftarrow tc_ok + 1$
        end if
      end if
    end for
    $\textsc{Record}(model, n, sel_ok/|\mathcal{D}|, tc_ok/|\mathcal{D}|)$
  end for
end for

Benchmark B: LLMs vs. Jina Reranker on Tools with No Parameters

The second benchmark reframes tool selection as an IR problem and compares this against the generative approach. Only the 16 zero-parameter tools are considered, and only questions whose ground-truth tool belongs to this subset are evaluated (320 queries). This restriction is necessary because the re-ranker cannot produce arguments: its only output is a ranked list of tool names; thus, the comparison is limited to the regime where both approaches are deployable.

For each query, the re-ranker scores every `(query, tool_name + description)` pair in the pool. The tool with the highest relevance score is taken as the prediction. The `model.rerank()` interface is used with `max_length=1024`: this safely handles longer tool descriptions via automatic chunking. Selection accuracy is measured at each pool size $\{4, 8, 12, 16\}$ for both the Jina reranker and the LLMs.

Algorithm 3 shows the re-ranker variant: each tool is scored as a (query, name + description) document, and the top-ranked tool is taken as the prediction.

Benchmark B—Reranker (Approach 2):

Algorithm 3 Benchmark B: reranker (Approach 2) on zero-parameter tools

Input: Questions $\mathcal{Q}_{16}$, pools $\mathcal{P}$, sizes $\{4, 8, 12, 16\}$
$reranker \leftarrow \textsc{LoadReranker}(\texttt{jina-reranker-v2-base-multilingual})$
for $n \in \{4, 8, 12, 16\}$ **do**
  $ok \leftarrow 0$
  for $q \in \mathcal{Q}_{16}$ **do**
    $pool \leftarrow \mathcal{P}[n][q.id]$
    $docs \leftarrow [t.name \,\|\, \texttt{": "} \,\|\, t.desc \textbf{ for } t \in pool]$
    $result \leftarrow reranker.\textsc{Rerank}(q.prompt, docs, top_n = 1)$
    $predicted \leftarrow pool[result[0].index]$
    if $predicted.name = q.gt.name$ **then**
      $ok \leftarrow ok + 1$
    end if
  end for
  $\textsc{Record}(\texttt{reranker}, n, ok/|\mathcal{Q}_{16}|)$
end for

The same LLMs from Benchmark A are evaluated on the identical pools and question subsets, using the same dispatch prompt. Also note that, for LLMs, only selection accuracy is recorded (since all tools in $\mathcal{U}_{16}$ take no arguments).

Algorithm 4 gives the matching LLM variant on the identical pools, recording only selection accuracy.

Benchmark B—LLM (Approach 1) on zero-param tools:

Algorithm 4 Benchmark B: LLM (Approach 1) on zero-parameter tools

Input: Models $\mathcal{M}$, questions $\mathcal{Q}_{16}$, pools $\mathcal{P}$, sizes $\{4, 8, 12, 16\}$
for $model \in \mathcal{M}$ **do**
  for $n \in \{4, 8, 12, 16\}$ **do**
    $ok \leftarrow 0$
    for $q \in \mathcal{Q}_{16}$ **do**
      $pool \leftarrow \mathcal{P}[n][q.id]$
      $pred_name, _ \leftarrow model.\textsc{Dispatch}(q.prompt, pool)$
      if $pred_name = q.gt.name$ **then**
        $ok \leftarrow ok + 1$
      end if
    end for
    $\textsc{Record}(model, n, ok/|\mathcal{Q}_{16}|)$
  end for
end for

Since the re-ranker and every LLM share the same pools, the same questions, and the same pool sizes, their selection accuracies are directly comparable. Running both across the (pool size $\times$ model) grid reveals whether tool selection with no parameters, given relatively small pool sizes, is essentially a retrieval problem or whether LLMs contribute with additional capability. This also shows whether the re-ranker degrades under distractor pressure in the same way the LLM does.

Benchmark C: Answer Generation from Tool Results

The third benchmark evaluates the final stage of the agentic loop: the production of the final reply in natural language. Assuming that a correct tool-call has already been issued, this benchmark evaluates how well each model turns the results from the tool calls into a useful natural language reply to the user's query. Only test queries with a ground-truth tool are considered. For each such question, the model is fed a fixed four-message context: a system prompt instructing it to summarize the returned data while mentioning all the important information and the available sensors, the raw user query, an assistant turn containing the (ground-truth) tool-call, and a tool turn carrying the function's reply. The model then produces its final answer, which is recorded together with its raw output and the number of generated tokens.

Since a single tool may return a variable amount of data (depending on how many devices support that tool in the network), each question is associated with a fixed bank of 20 candidate function replies. The number N of replies shown to the model is varied across $\{1, 5, 10, 15, 20\}$. The shown subset of replies is constructed deterministically: as with the benchmarks A and B, the reply samples are pre-generated with a fixed random seed before the benchmark loop. This guarantees that every tested model sees the identical data for a given (question, N) pair.

All models are evaluated with the same summarization system prompt so that differences reflect model behavior rather than prompt engineering.

Algorithm 5 outlines the generation procedure: each model is fed the fixed four-message context with the deterministically sampled replies and produces the final answer.

Benchmark C—How Tested LLMs do Answer Generation:

Algorithm 5 Benchmark C: answer generation from tool results

Input: Models $\mathcal{M}$, dataset $\mathcal{D}$, reply samples $\mathcal{S}$, counts $\{1, 5, 10, 15, 20\}$
for $model \in \mathcal{M}$ **do**
  for $N \in \{1, 5, 10, 15, 20\}$ **do**
    for $q \in \mathcal{D}$ **do**
      $idxs \leftarrow \mathcal{S}[N][q.id]$
      $seen \leftarrow [\, q.replies[i] \textbf{ for } i \in idxs \,]$
      $msgs \leftarrow [\textsc{Sys}(P_C), \textsc{User}(q.prompt), \textsc{Asst}(q.gt), \textsc{Tool}(seen)]$
      $reply, tokens \leftarrow model.\textsc{Generate}(msgs)$
      $\textsc{Record}(model, N, q.id, reply, tokens)$
    end for
  end for
end for

Since summary quality cannot be scored by exact matching, Benchmark C is evaluated with an LLM-as-judge.

The LLM chosen for this task is GLM-4.7-Flash. A local model was selected instead of a cloud-based one to enhance reproducibility. Since a coherent structured output is important for the task, GLM-4.7-Flash was chosen as it is one of the strongest open-source models below 30 billion parameters on coding (software-engineering) benchmarks. GLM-4.7-Flash achieves 59.2 on SWE-bench verified, against 34.0 for GPT-OSS-20B and 22.0 for Qwen3-30B-A3B-Thinking, while also sustaining over 50 tokens per second on the aforementioned computer used for testing.

For each record the judge is shown the user query, the tool-call, the returned data (tool results), and the model's final reply, then it grades the reply from 1 to 10 on relevance, faithfulness, use of evidence, and clarity, responding with only {`"grade"`: ⟨1–10⟩} at temperature 0. The integer grade is recovered and accepted if it is within $[1, 10]$. Records that fail to parse are assigned a placeholder and set aside for manual annotation. The judge's raw output is logged for every record. The protocol is applied uniformly across all tested LLM configurations and reply counts N. Grades are directly comparable across the (model $\times$ N) grid. The full judge system prompt and the grade-extraction procedure (including the handling of unparseable judge outputs) are reported in Appendix D.

This LLM-as-a-judge approach is validated in Appendix E on a proxy task. The validation focuses on the judge's relative ordering of reply quality rather than absolute score calibration, since, in this benchmark, the grades are used primarily to track how reply quality trends as the number of tool results increases. Results show that the judge's rankings correlate positively with expert human annotations, albeit below the level of inter-expert agreement (Kendall $\tau_b = 0.37$ vs. 0.58). Plots are reported in Figure A1.

6.2. Model Analysis Results

The results of the benchmarks A, B, and C are shown in their respective sections:

1. Section 6.2.1, Benchmark A: tool-calling of Approach 1. LLMs comparison on a maximum of 24 tools: selection accuracy (correct tool name) and tool-call accuracy (correct tool name and correct tool parameters) are considered.
2. Section 6.2.2, Benchmark B: tool-calling of Approach 2 compared with Approach 1. LLMs and Jina reranker were compared only on selection accuracy on a maximum of 16 tools.
3. Section 6.2.3, Benchmark C: assuming a correct tool-call, the final answer generation is evaluated given 1, 5, 10, 15, 20 network replies.

6.2.1. Benchmark A

Figure 6 shows tool-calling accuracy while varying the available tools for the tested configurations. This includes tools with parameters.

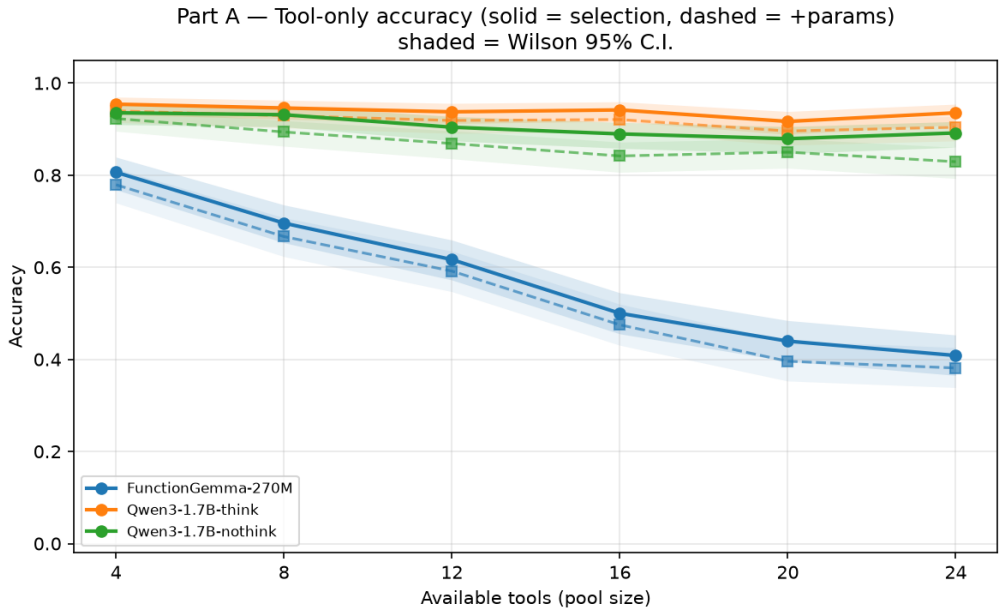


Figure 6. Benchmark A: selection (solid) and selection+params (dashed) accuracy on the tool-bearing subset, vs. pool size.

Figure 7 shows the tokens generated by the LLMs (Y axis) during the tool-calling attempts while varying the pool size (X axis).

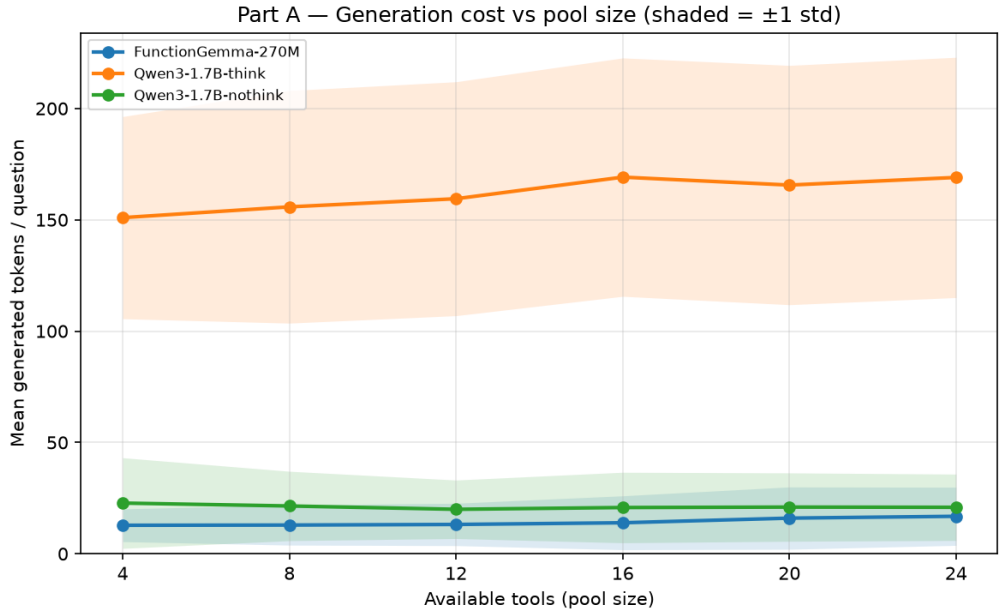


Figure 7. Benchmark A: mean generated tokens per question, vs. pool size.

Table 3 shows Part A performance, reporting selection and tool-calling accuracy for pools of size $n = 4$ and $n = 24$, abstention accuracy, and mean token count.

Table 3. Part A: tool-calling on the full 24-tool set. Selection (Sel) and tool-calling (TC, selection+params) accuracy on the tool-bearing subset at the smallest and largest pools ($n = 4, 24$), abstention accuracy on the null-tool subset at $n = 24$, and mean generated tokens per question (pooled over all pools). Accuracies are fractions shown as mean ± half-width of the Wilson 95% C.I.; tokens are mean ± 1 std.

Configuration	Sel@4 (Frac.)	Sel@24 (Frac.)	TC@4 (Frac.)	TC@24 (Frac.)	Null@24 (Frac.)	Tokens/q (Tok.)
FunctionGemma-270M	0.806 ± 0.035	0.408 ± 0.044	0.779 ± 0.037	0.381 ± 0.043	0.800 ± 0.168	14.3 ± 11.2
Qwen3-1.7B-think	0.954 ± 0.019	0.935 ± 0.022	0.940 ± 0.022	0.904 ± 0.026	1.000 ± 0.081	161.7 ± 52.4
Qwen3-1.7B-nothink	0.935 ± 0.022	0.892 ± 0.028	0.923 ± 0.024	0.829 ± 0.034	0.950 ± 0.114	21.1 ± 16.0

6.2.2. Benchmark B

The following figures and tables show tool-calling accuracy while varying the available tools, excluding the tools with parameters. This section shows the selection accuracy of the retriever and of the LLMs in performing this task.

In Figure 8 and in Table 4, it is clearly visible how functiongemma:270m accuracy degrades as the number of tools increases, while the other configurations are less affected.

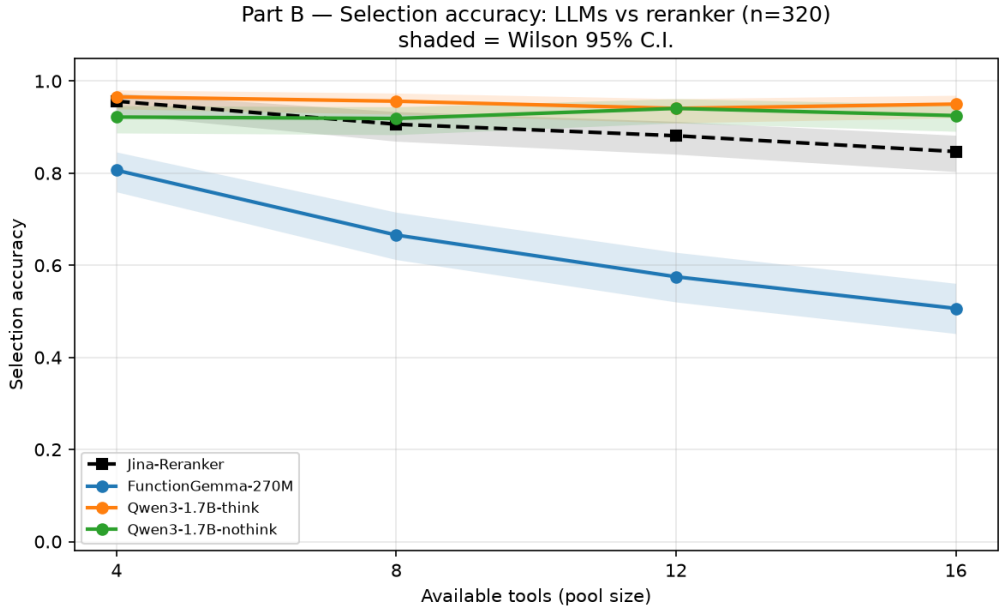


Figure 8. Benchmark B: selection accuracy of each LLM configuration and the Jina reranker, vs. pool size.

Figure 9 makes it visually evident that the number of tokens generated by Qwen3:1.7b thinking is almost an order of magnitude bigger than the non-thinking variant (according to Table 3, the average ratio between the token count of the thinking and the non-thinking variant is 161.7:21.1, approximately 7.7×).

Table 4. Part B: selection accuracy on the 16 zero-parameter tools (320 queries), for each LLM configuration and the Jina reranker, at every pool size. Values are fractions shown as mean ± half-width of the Wilson 95% C.I.

Method	$n = 4$ (Frac.)	$n = 8$ (Frac.)	$n = 12$ (Frac.)	$n = 16$ (Frac.)
FunctionGemma-270M	0.806 ± 0.043	0.666 ± 0.051	0.575 ± 0.054	0.506 ± 0.054
Qwen3-1.7B-think	0.966 ± 0.021	0.956 ± 0.023	0.941 ± 0.026	0.950 ± 0.024
Qwen3-1.7B-nothink	0.922 ± 0.030	0.919 ± 0.030	0.941 ± 0.026	0.925 ± 0.029
Jina-Reranker	0.956 ± 0.023	0.906 ± 0.032	0.881 ± 0.036	0.847 ± 0.039

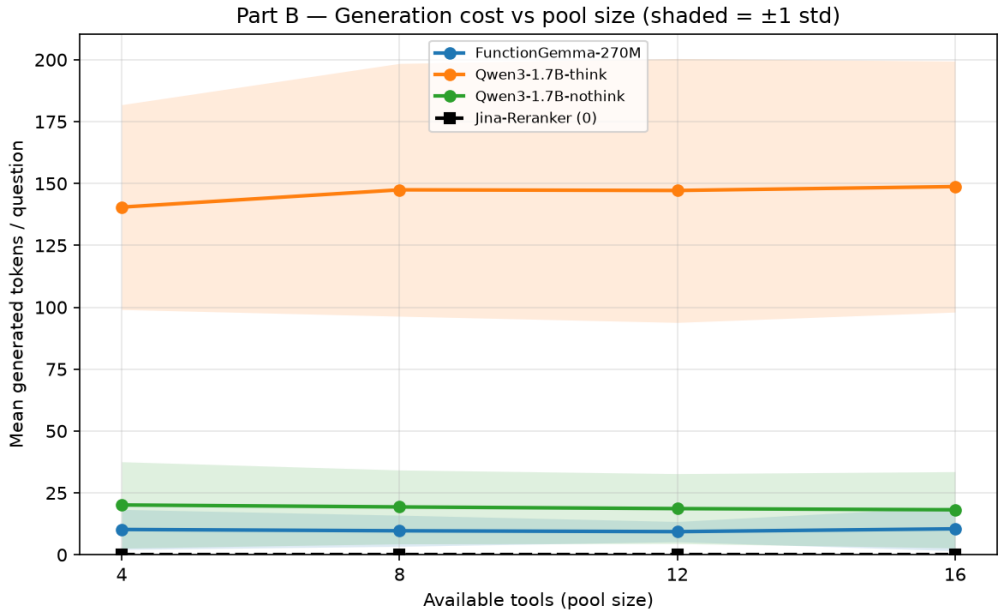


Figure 9. Benchmark B: mean generated tokens per question (the reranker generates none), vs. pool size.

6.2.3. Benchmark C

Grades assigned by the LLM judge (glm-4.7-flash:30b) are plotted in Figure 10. Note how for qwen3:1.7b thinking, the mean and the median of the grade basically coincide. Grades do degrade as the context length and number of available function-replies increase. This degradation is greater for functiongemma, less so for qwen3.

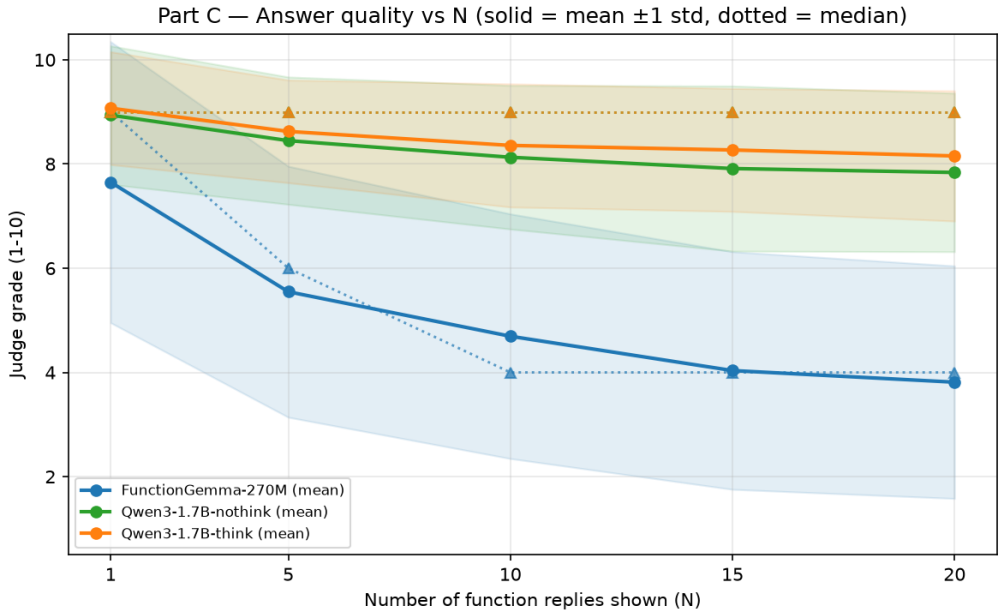


Figure 10. Benchmark C: grade assigned to the final reply summarizing the tool replies as the number of function results *N* shown to the model grows.

The number of tokens generated peaks at 15 function replies for functiongemma:270m, as visible in Figure 11. A possible explanation will be discussed in Section 8.

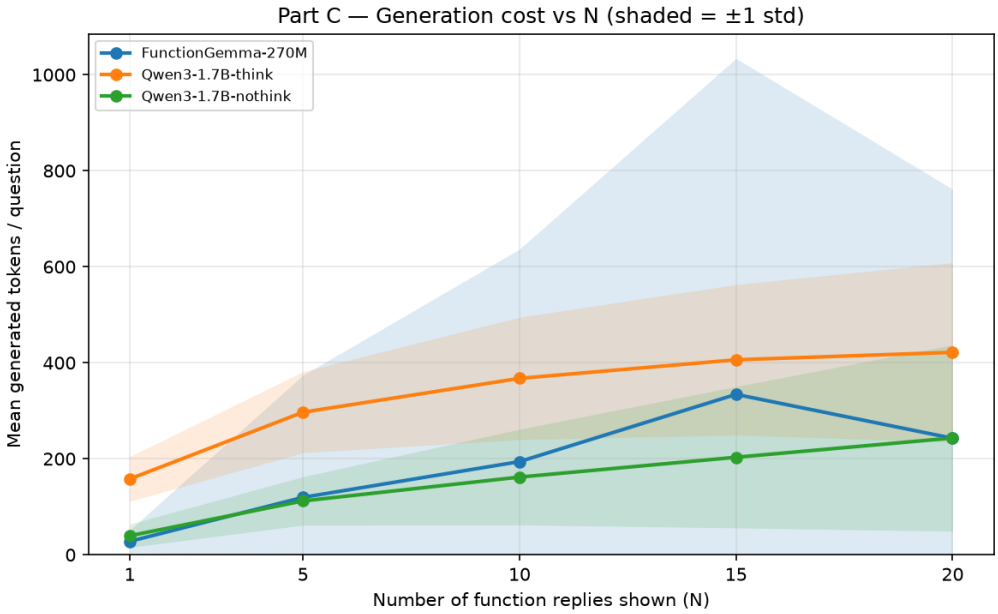


Figure 11. Benchmark C: mean generated tokens per question as the number of function replies N shown to the model grows.

Table 5 aggregates the main numerical data plotted in Figures 10 and 11.

Table 5. Part C: answer-generation quality and cost vs. number of function replies N. Grade is the LLM-judge score (1–10), tokens are mean generated tokens per question; both spreads are ± 1 std. Token distributions are right-skewed; ± std is a spread measure, not a symmetric interval.

Configuration	N (Count)	Median (1–10)	Mean ± Std (1–10)	Tokens/q (Tok.)
FunctionGemma-270M	1	9.0	7.65 ± 2.70	27.9 ± 23.4
	5	6.0	5.55 ± 2.41	119.6 ± 254.3
	10	4.0	4.69 ± 2.35	194.0 ± 442.4
	15	4.0	4.04 ± 2.28	334.1 ± 699.6
	20	4.0	3.81 ± 2.23	242.7 ± 518.9
Qwen3-1.7B-think	1	9.0	9.07 ± 1.08	157.9 ± 46.3
	5	9.0	8.62 ± 0.99	296.9 ± 84.4
	10	9.0	8.36 ± 1.18	367.4 ± 127.2
	15	9.0	8.27 ± 1.18	406.1 ± 156.5
	20	9.0	8.15 ± 1.25	421.7 ± 186.7
Qwen3-1.7B-nothink	1	9.0	8.94 ± 1.33	39.6 ± 24.1
	5	9.0	8.45 ± 1.22	112.1 ± 50.3
	10	9.0	8.13 ± 1.38	161.8 ± 99.5
	15	9.0	7.91 ± 1.59	203.1 ± 147.1
	20	9.0	7.84 ± 1.52	243.2 ± 194.0

7. On Target Analysis

This section concerns measurements of the energy consumption and latency of the devices in the network. The benchmark targets the two edge devices (i.e., the edge nodes of the proposed architecture) introduced in Section 5.3. Both devices can host sensing agents as well as leader agents.

One metric of interest is the energy cost of the leader role, calculated by measuring the power consumption delta between leader mode and sensing mode on the same device. This metric serves as a proxy for the computational overhead of the leader role and is directly

applicable to energy-constrained deployments. The testbed uses mains-powered devices, but the delta can also generalize to battery-powered deployments where an unexpected leader election directly impacts energy budgeting.

Another metric of interest is the delta between the leader capabilities of the Raspberry Pi 5 when the leader LLM is executed by the CPU and when it is executed instead on the Hailo.

7.1. On Target Analysis Setup

Power was measured using a FNIRSI FNB-C2 USB-C power meter. A USB-C cable connected the power supply to the FNB-C2 input, and the FNB-C2 was directly connected to each board's power input, placing the meter in-line on the supply path. The meter's HID/USB data port was connected to a separate logging computer (running the UsbMeter software), allowing voltage and current readings to be streamed and recorded without interfering with the power path itself. Instantaneous power was computed as the product of simultaneously sampled voltage and current, $P = V \cdot I$.

Per manufacturer specification, the FNB-C2 uses a 20-bit ADC with a resolution of $1\,\mu V$, $1\,\mu A$, and $1\,\mu W$, and a rated accuracy of $\pm(0.02\,\% + 2\,\text{digits})$ for voltage and $\pm(0.05\,\% + 2\,\text{digits})$ for current and power. The voltage and current readings of the FNB-C2 were verified against a laboratory reference instrument, as reported in Appendix G. Voltage and current were logged at 5 samples per second (SPS) over a 20-min (1200 s) window of active use for each configuration, at room temperature. Each configuration was measured independently so that the operating-system-only baseline could be subtracted to obtain the net dynamic consumption attributable to each workload. For visualization the raw power trace was smoothed with a centered rolling-mean window of 15 samples (3 s). Reported summary statistics (mean and standard deviation of voltage, current, and power per configuration) were computed from the raw, unsmoothed samples. The configurations tested are shown in Table 6.

Table 6. Hardware and software configurations under test. Each configuration is measured over a 20-min window of active use. Detection uses YOLOv8n; language-model workloads use functiongemma:270m (STM32MP257FDK) and Qwen3:1.7b (Raspberry Pi 5).

Platform	Configuration	Duration
STM32MP257FDK	Baseline (OS only, no application) YOLOv8n:320 on NPU YOLOv8n:320 on CPU functiongemma:270m on CPU YOLOv8n:320 NPU + functiongemma:270m CPU YOLOv8n:320 CPU + functiongemma:270m CPU	20 min
RPi 5 + Hailo-10H	Baseline (OS only, Hailo idle) YOLOv8n:640 on CPU YOLOv8n:640 on Hailo Qwen3:1.7b on CPU Qwen3:1.7b on Hailo Qwen3:1.7b CPU + YOLOv8n:640 Hailo YOLOv8n:640 CPU + Qwen3:1.7b Hailo	20 min

7.2. Results of on Target Analysis

This section is divided in Power consumption Section 7.2.1, Latency and scalability Section 7.2.2 and Failover time Section 7.2.3.

7.2.1. Power Consumption

Tables 7–9 show numerical data related to the aforementioned configurations; respectively, the tables show the mean and the standard deviation of the current, voltage, and power during the benchmark.

Table 7. Current statistics per configuration.

Platform	Configuration	Mean [A]	Std Dev [A]
RPi 5 + Hailo-10H	no work	1.125	0.054
	YOLOv8n:640 cpu only	1.788	0.047
	YOLOv8n:640 hailo only	1.351	0.034
	qwen3 cpu only	1.461	0.046
	qwen3 hailo only	1.364	0.019
	YOLOv8n:640 cpu + qwen3 hailo	2.034	0.087
	YOLOv8n:640 hailo + qwen3 cpu	1.665	0.038
STM32MP257FDK	no work	0.889	0.011
	YOLOv8n:320 cpu only	1.105	0.013
	YOLOv8n:320 npu only	1.137	0.012
	functiongemma cpu only	0.999	0.012
	YOLOv8n:320 cpu + functiongemma cpu	1.118	0.013
	YOLOv8n:320 npu + functiongemma cpu	1.138	0.014

Table 8. Voltage statistics per configuration.

Platform	Configuration	Mean [V]	Std Dev [V]
RPi 5 + Hailo-10H	no work	5.118	0.058
	YOLOv8n:640 cpu only	5.125	0.001
	YOLOv8n:640 hailo only	5.121	0.000
	qwen3 cpu only	5.123	0.001
	qwen3 hailo only	5.122	0.000
	YOLOv8n:640 cpu + qwen3 hailo	5.126	0.001
	YOLOv8n:640 hailo + qwen3 cpu	5.122	0.055
STM32MP257FDK	no work	4.839	0.003
	YOLOv8n:320 cpu only	4.785	0.003
	YOLOv8n:320 npu only	4.777	0.003
	functiongemma cpu only	4.812	0.003
	YOLOv8n:320 cpu + functiongemma cpu	4.781	0.003
	YOLOv8n:320 npu + functiongemma cpu	4.775	0.004

Table 9. Power statistics per configuration.

Platform	Configuration	Mean [W]	Std Dev [W]
RPi 5 + Hailo-10H	no work	5.761	0.286
	YOLOv8n:640 cpu only	9.167	0.241
	YOLOv8n:640 hailo only	6.918	0.177
	qwen3 cpu only	7.490	0.238
	qwen3 hailo only	6.987	0.099
	YOLOv8n:640 cpu + qwen3 hailo	10.429	0.451
	YOLOv8n:640 hailo + qwen3 cpu	8.532	0.219
STM32MP257FDK	no work	4.303	0.052
	YOLOv8n:320 cpu only	5.287	0.058
	YOLOv8n:320 npu only	5.434	0.056
	functiongemma cpu only	4.808	0.056
	YOLOv8n:320 cpu + functiongemma cpu	5.349	0.061
	YOLOv8n:320 npu + functiongemma cpu	5.436	0.065

As shown in Figure 12, the Hailo accelerator significantly reduces power consumption compared with CPU-only inference, especially for the YOLOv8n:640 inference. Furthermore, Figure 13 illustrates the distinct power profiles of the STM configurations. Detailed current, voltage, and power statistics are provided in Tables 7–9.

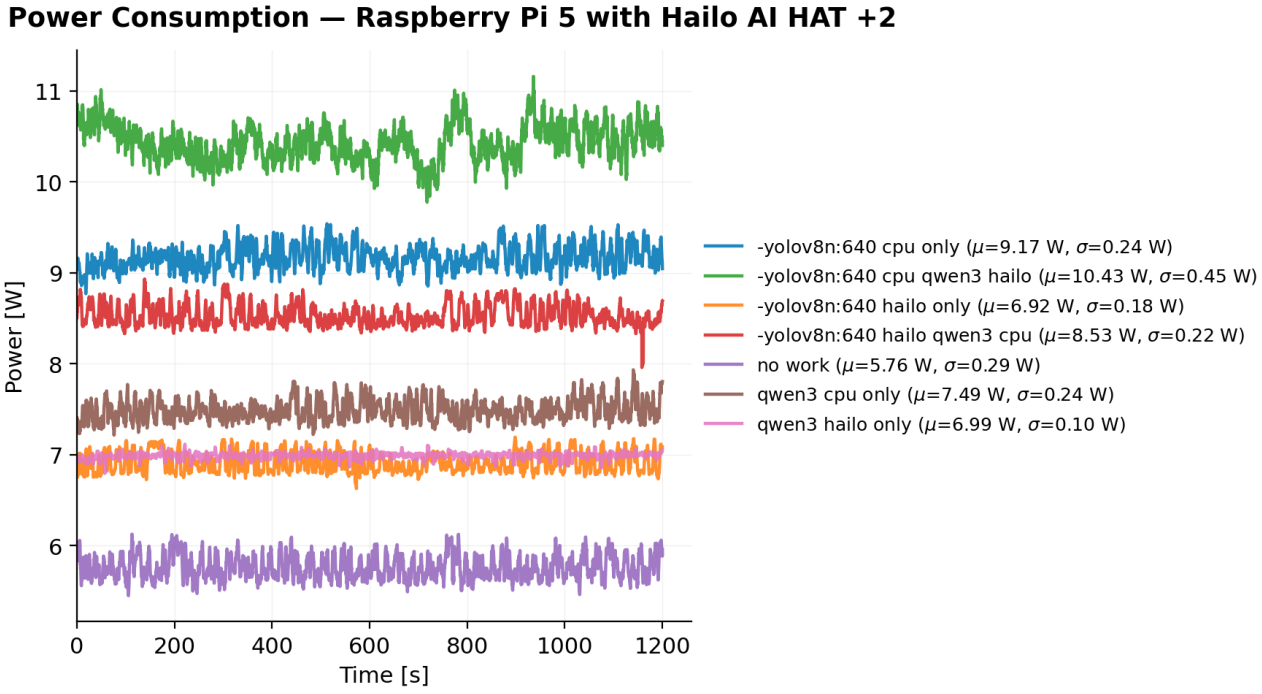


Figure 12. Power consumption comparison of Raspberry Pi 5 under different inference modes.

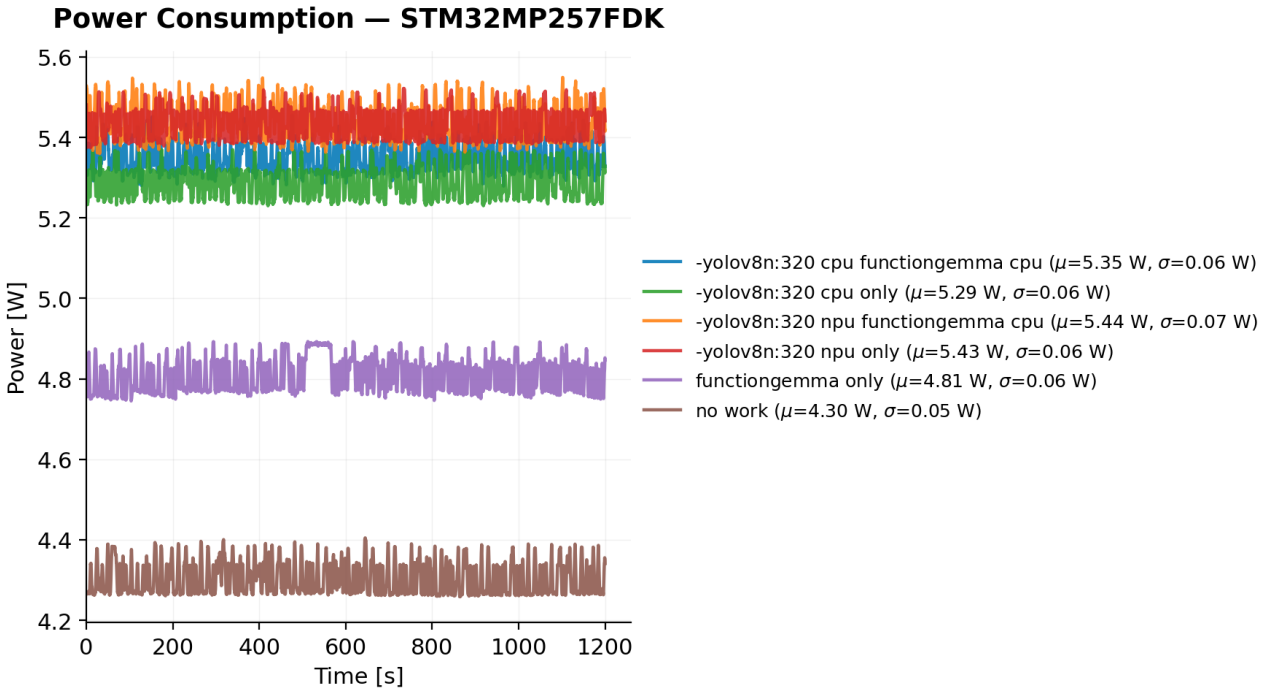


Figure 13. Power consumption comparison of STM32MP257FDK board across various operational configurations.

7.2.2. Latency and Scalability

This subsubsection characterizes the latency of the leader agent and its scaling behavior. The end-to-end latency is the time from the reception of a user query to the delivery of

the natural language reply. Serving a query involves two language-model invocations on the leader: a tool-dispatch stage, which reads the user query together with the JSON descriptions of the exposed tools and emits one tool call, and a final-reply stage, which turns the returned tool results into a natural language answer. For each stage, the time to first token (TTFT), which corresponds to the prefill of the prompt, is reported separately, as it dominates the stage time.

The configurations for this test were decided by picking the fastest leader-LLM backend measured during the continuous inference that was run to measure the power consumption in Section 7.2.1. These configurations should minimize the latency across the test sentences while remaining representative of a deployed configuration.

On the Raspberry Pi 5, Qwen3:1.7b reaches its highest throughput on the Hailo accelerator at 4.88 TPS, against 3.43 TPS on the CPU; YOLOv8n:640 likewise runs faster on the Hailo at 19.95 FPS, compared with 3.39 FPS on the CPU. On the STM32MP257FDK, functiongemma:270m is only deployable on the CPU, reaching 4.28 TPS, while YOLOv8n:320 attains 18.24 FPS on the NPU rather than 1.65 FPS on the CPU as reported in Table 10.

Table 10. Measured performance (FPS and TPS) per configuration during continuous model inference. FPS refers to the YOLOv8n detection workload and TPS to the language-model workload; a dash (—) indicates the metric does not apply to that configuration.

Platform	Configuration	Avg FPS	Avg TPS
STM32MP257FDK	YOLOv8n:320 cpu only	1.65	—
	YOLOv8n:320 npu only	18.24	—
	functiongemma cpu only	—	4.28
	YOLOv8n:320 cpu + functiongemma cpu	0.83	1.24
	YOLOv8n:320 npu + functiongemma cpu	11.20	1.24
RPi 5 + Hailo-10H	qwen3 cpu only	—	3.43
	qwen3 hailo only	—	4.88
	YOLOv8n:640 cpu only	3.39	—
	YOLOv8n:640 hailo only	19.95	—
	YOLOv8n:640 hailo + qwen3 cpu	19.01	2.61
	YOLOv8n:640 cpu + qwen3 hailo	3.16	4.83

Ten test sentences, whose ground truth is the tool exposed by the tested sensing agent, are analyzed by the Leader Agent that calls the appropriate tool.

Two leader backends were measured: `Qwen3:1.7b` on the Hailo-10H of the Raspberry Pi 5 and `functiongemma:270m` on the CPU of the STM32MP257FDK. After the model load time, latency was measured along two axes: the number of exposed tools and the number of aggregated replies. The interpretation of these measurements is given in Section 8.3.

Model Load Time

Before serving any query, the leader loads its language model. This cost is paid once, at agent start-up or immediately after a leader election, and is not part of the per-query latency. Table 11 reports the load time over ten consecutive loads per model. Both backends show a one-time cold start on the first load that is markedly slower than the warm loads, which cluster near the minimum (3.89 s for `functiongemma:270m`, 6.75 s for `Qwen3:1.7b`), while the cold start reaches the maximum (9.57 s and 31.03 s, respectively). The mean and standard deviation are computed over all ten loads and therefore include the cold-start outlier.

Table 11. Model load time (time to load the language model on the leader), over ten consecutive loads per model. The minimum is a warm load, the maximum a cold start.

Model	Device (Backend)	Mean [s]	Std [s]	Min [s]	Max [s]
`functiongemma:270m`	STM32MP257FDK (CPU)	4.47	1.79	3.89	9.57
`Qwen3:1.7b`	Raspberry Pi 5 (Hailo)	9.27	7.65	6.75	31.03

Agent Scaling and Network Overhead

The cost of scaling the number of agents was measured first, in power and in network traffic. Table 12 and Figure 14 report the aggregate power consumption of the STM32MP257FDK as the number of co-located agents grows from 1 to 100. The mean power stays close to 4.3 W throughout the whole range.

Table 12. Power statistics per number of agents.

Section	Min [W]	Mean [W]	Max [W]	Std Dev [W]
1 agent	4.1740	4.2936	4.4969	0.0332
10 agents	4.1761	4.2956	4.3877	0.0294
20 agents	4.1765	4.2991	4.3821	0.0264
50 agents	4.1837	4.3021	4.4377	0.0263
100 agents	4.1839	4.3086	4.5129	0.0302

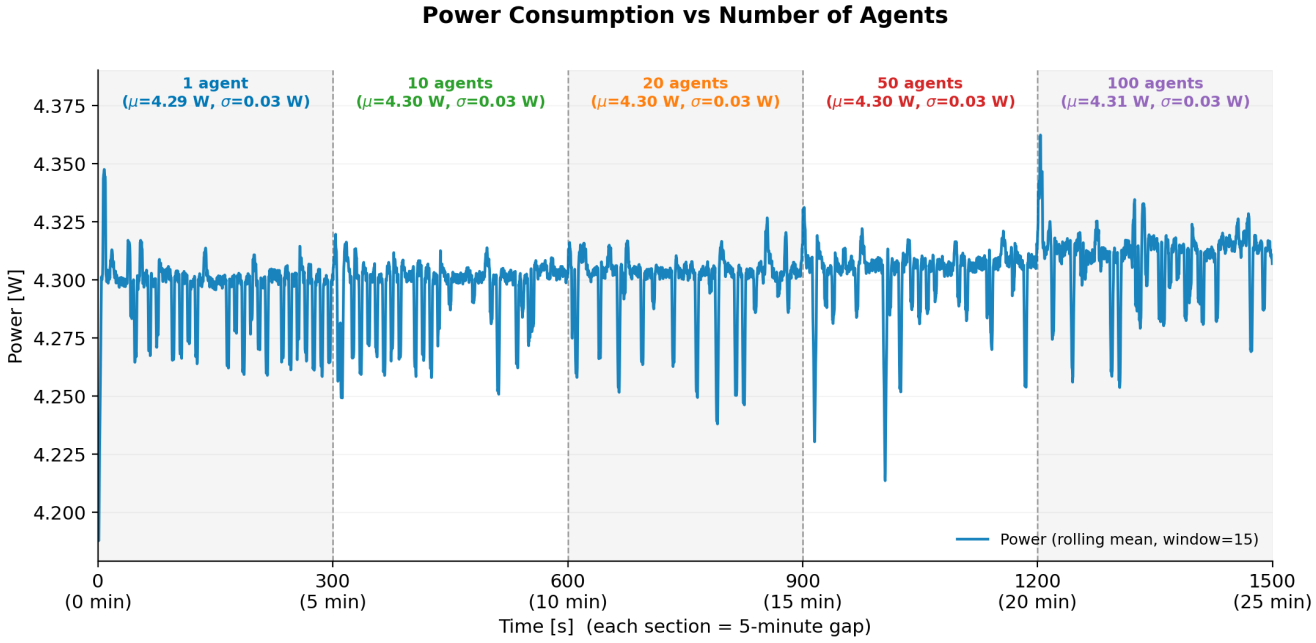


Figure 14. Aggregate power consumption of the STM32MP257FDK as the number of co-located agents grows from 1 to 100, measured on the board itself. The draw stays close to 4.3 W across the whole range.

The base network traffic was tracked under the same scaling, with all sensing agents simulated on a single machine: a mock leader ran only the discovery and tool-aggregation overlay (no model, election, or dispatch) while the fleet grew from 1 to 100. Table 13 reports the leader's base traffic as bytes in and out per minute (application-level JSON payload only, in KiB/min). The received load grows linearly with the fleet and is dominated by discovery: each agent re-announces every two seconds, so the HELLO beacons add about 1.8 KiB/min per agent, reaching 184.8 KiB/min in and 10.5 KiB/min out at 100 agents. The

two quantities that drive the leader's language-model processing, the number of exposed tools and the number of aggregated replies, are studied next.

Table 13. Leader base network load as the number of sensing agents grows, all agents simulated on a single machine. Rates are bytes in and out per minute, averaged over the one-minute hold at each fleet size (window length in the second column); the idle row is the leader alone after the fleet stopped. Application-level JSON payload only (KiB/min), excluding TCP/IP/Ethernet and handshake overhead.

Agents	Window [s]	Msgs/min in	Msgs/min out	In [KiB/min]	Out [KiB/min]
0 (idle)	431	30	30	1.7	1.7
1	60	76	40	5.9	2.6
10	60	351	43	22.5	2.8
20	60	642	62	36.8	4.5
50	60	1578	81	96.2	6.2
100	60	2833	130	184.8	10.5

Latency as the Number of Tools Grows

The two tested configurations are the following:

- Configuration 1: the STM32MP257FDK is the leader, running functiongemma:270m on its CPU, while the Raspberry Pi 5 is the sensing device, executing the called tool with YOLOv8n:640 on the Hailo.
- Configuration 2: the Raspberry Pi 5 is the leader, running Qwen3:1.7b on the Hailo, while the STM32MP257FDK is the sensing device, executing the called tool with YOLOv8n:320 on the CPU.

In this experiment, the leader was presented with a growing pool of tools (1, 6, 12, 18, and 24) while a single sensing agent returned one result per query; ten queries whose ground-truth tool is `detect_people` were issued at each pool size. Table 14 reports `Qwen3:1.7b` on the Hailo and Table 15 `functiongemma:270m` on the STM32 CPU, both with the full prompt and no key-value (KV) cache reuse. The dispatch prompt embeds the full JSON schema of every exposed tool, so it lengthens by about 47 tokens per tool for Qwen3 (223 to 1318 tokens) and about 29 for functiongemma (97 to 755 tokens). The standard pipeline swaps the system prompt between the two stages (a tool-listing dispatcher prompt and a separate reply prompt), so each stage prefills independently: in the KV use pipeline, the system prompt is instead kept constant. Figure 15 plots the resulting per-query TTFT against the prompt length.

Table 14. Leader latency of `Qwen3:1.7b` on the Hailo-10H as the number of exposed tools grows (sensing agent on the STM32MP257FDK, YOLOv8n on the CPU). Values are mean ± standard deviation. The three-stage columns sum to the end-to-end column, which is the mean over the completed queries of their per-query total. Ten queries were issued at each pool size. Input token counts are reported as single values because they are fixed by the benchmark configuration. Standard deviations below 0.005 s are reported as 0.00 (Hailo-hosted models tend to be highly consistent across runs).

Tools	Tokens in	Query→Tool [s]	Tool→Result [s]	Result→Reply [s]	E2E [s]
1	223	4.98 ± 0.00	0.64 ± 0.03	4.83 ± 0.52	10.44 ± 0.52
6	458	6.21 ± 0.00	0.62 ± 0.02	4.98 ± 0.50	11.81 ± 0.50
12	742	8.06 ± 0.00	0.64 ± 0.04	4.77 ± 0.51	13.47 ± 0.51
18	1030	9.96 ± 0.13	0.61 ± 0.22	4.98 ± 0.81	15.55 ± 0.73
24	1318	11.77 ± 0.00	0.63 ± 0.01	4.80 ± 0.54	17.19 ± 0.53

Table 15. Leader latency of `functiongemma:270m` on the STM32MP257FDK CPU (full prompt, no KV cache reuse) as the number of exposed tools grows (sensing agent on the Raspberry Pi 5, YOLOv8n on the Hailo NPU). Values are mean ± standard deviation. The three-stage columns sum to the end-to-end column, the mean over the completed queries of their per-query total. Ten queries were issued at each pool size. Input token counts are reported as single values because they are fixed by the benchmark configuration.

Tools	Tokens in	Query→Tool [s]	Tool→Result [s]	Result→Reply [s]	E2E [s]
1	97	9.12 ± 0.12	0.05 ± 0.00	17.62 ± 1.31	26.79 ± 1.40
6	237	20.02 ± 0.12	0.05 ± 0.01	17.62 ± 1.30	37.69 ± 1.37
12	407	36.94 ± 0.15	0.05 ± 0.00	17.49 ± 1.33	54.49 ± 1.44
18	581	52.82 ± 0.16	0.07 ± 0.05	17.60 ± 1.31	70.50 ± 1.41
24	755	69.42 ± 0.16	0.05 ± 0.00	17.50 ± 1.33	86.96 ± 1.45

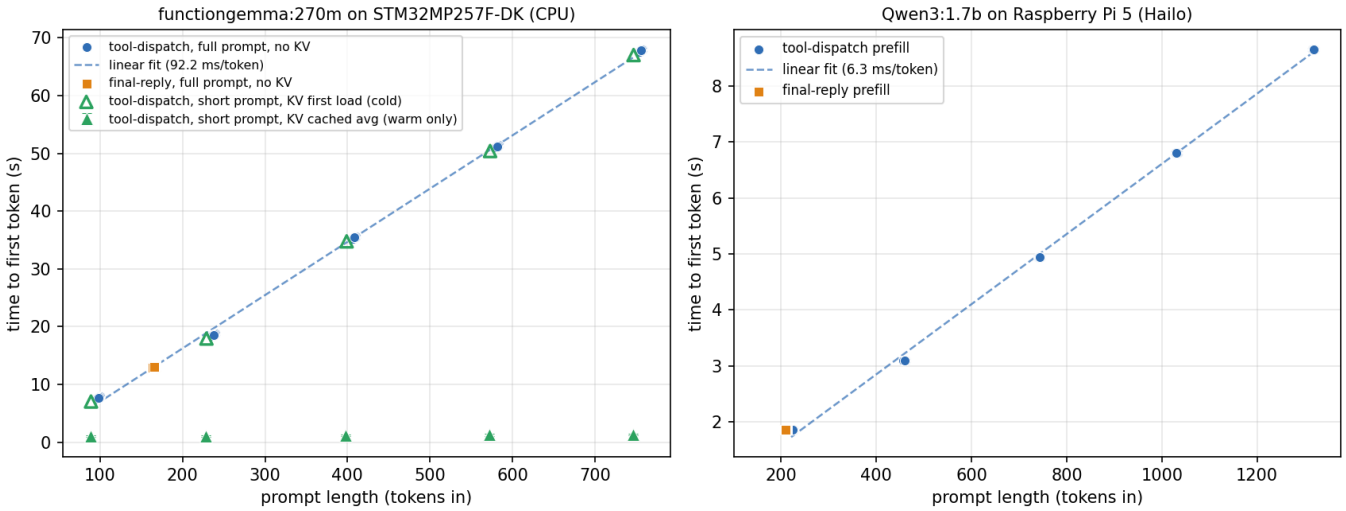


Figure 15. TTFT (prefill time) against prompt length for the two leader backends. Filled circles are the tool-dispatch stage with the full prompt and no KV reuse; squares are the final-reply stage. On the STM32 panel, hollow triangles are the cold first-load dispatch prefill of the short-prompt KV configuration (one per pool, first query), and filled triangles are the same stage averaged only over the warm cached queries of each pool, with error bars for the standard deviation. The KV cache is reset at each change in the number of tools. Dashed lines are least-squares fits through the full-prompt dispatch points.

A third configuration repeats the `functiongemma:270m` measurement with a shorter, fixed dispatch prompt (of the form "You are a model that can do function calling with the following functions") and KV-cache reuse across the ten queries of each pool. The KV cache is reset whenever the number of tools changes, so the tool descriptions are prefilled once per pool: the first query pays the full prefill, and the following nine reuse the cache, re-prefilling only the short user message. Table 16 reports the same stage breakdown for this configuration, over the nine warm (cached) queries of each pool; the one-time cold first query of each pool is excluded here and reported in the TTFT comparison of Table 17.

Table 17 collects the dispatch TTFT (the prompt prefill) of the three configurations: `Qwen3:1.7b` on the Hailo, `functiongemma:270m` with the full prompt and no reuse, and `functiongemma:270m` with the short prompt and KV reuse. The cold first-load takes the same amount of time as the no-KV version. Table 18 then compares the end-to-end latency of the three, using the steady-state (cached) value for the KV configuration.

Table 16. Leader latency of `functiongemma:270m` on the STM32 CPU with the short prompt and KV-cache reuse (cache reset when the tool count changes). Unlike the other runs, this run dispatched to a sensing agent co-located on the same board, so tool→result (about 0.59 s) is higher than in the no-KV run. This inflates the end-to-end latency by about 0.55 s, since querying a sensing agent on the Hailo is faster than querying one hosted on the STM32MP257 itself. Improvements due to KV caching are valid independently of the hosted sensing agent. Values are mean ± standard deviation over the nine warm (cached) queries of each pool; the cold first query is excluded (see Table 17). The three-stage columns sum to the end-to-end column.

Tools	Tokens in	Query→Tool [s]	Tool→Result [s]	Result→Reply [s]	E2E [s]
1	88	2.45 ± 0.13	0.59 ± 0.00	6.74 ± 2.13	9.78 ± 2.14
6	228	2.48 ± 0.13	0.59 ± 0.00	7.26 ± 1.97	10.33 ± 1.99
12	398	2.68 ± 0.14	0.59 ± 0.00	7.50 ± 1.86	10.76 ± 1.84
18	572	2.88 ± 0.15	0.59 ± 0.00	6.46 ± 1.30	9.93 ± 1.28
24	746	2.99 ± 0.20	0.59 ± 0.00	7.51 ± 1.33	11.09 ± 1.34

Table 17. Dispatch TTFT [s] (prompt prefill, mean ± standard deviation) as the number of exposed tools grows, for the three configurations. For the short-prompt KV configuration, the cold first-load takes the same amount of time as the no-KV version.

Tools	Qwen3, Hailo [s]	Functiongemma, No KV [s]	Functiongemma KV, Warm [s]
1	1.86 ± 0.00	7.67 ± 0.12	1.01 ± 0.13
6	3.09 ± 0.00	18.56 ± 0.12	1.02 ± 0.13
12	4.94 ± 0.00	35.40 ± 0.14	1.14 ± 0.14
18	6.80 ± 0.00	51.20 ± 0.16	1.25 ± 0.15
24	8.65 ± 0.00	67.79 ± 0.16	1.27 ± 0.15

Table 18. End-to-end reply latency [s] (mean ± standard deviation) binned by the number of exposed tools, for the three configurations. The KV-cache column reports the steady-state (cached) latency.

Tools	Qwen3:1.7b, No KV [s]	Functiongemma, No KV [s]	Functiongemma, KV Cache [s]
1	10.44 ± 0.52	26.79 ± 1.40	9.78 ± 2.14
6	11.81 ± 0.50	37.69 ± 1.37	10.33 ± 1.99
12	13.47 ± 0.51	54.49 ± 1.44	10.76 ± 1.84
18	15.55 ± 0.73	70.50 ± 1.41	9.93 ± 1.28
24	17.19 ± 0.53	86.96 ± 1.45	11.09 ± 1.34

Latency as the Number of Replies Grows

This experiment exposes a single tool and varies the number of sensing agents that return a result (1, 5, 10, 15, 20), so the leader aggregates a growing set of results into one reply; ten queries are issued at each setting. Three configurations are compared: `Qwen3:1.7b` on the Hailo, and `functiongemma:270m` on the STM32 CPU with and without KV-cache reuse of the fixed prompt prefix (labeled KV reuse and reset, the reply-axis analogs of the KV and no-KV tool-scaling configurations). As in the tool-scaling test, the end-to-end time is decomposed into stages measured per query and averaged. The sensing turnaround tool→result is omitted here: the result set is assembled in-process (the harness builds the exact result the dispatcher would return, without the network hop, since the network latency is not the quantity under study), so tool→result only times that in-process construction and is negligible (under 0.1 ms), leaving query→tool + result→reply = E2E. In all three configurations, the dispatch stage (query→tool) is essentially invariant (a single tool), whereas the answer-stage prompt lengthens with the number of aggregated results and drives the latency. Tables 19–21 give the per-stage breakdown for `Qwen3:1.7b` on the Hailo, `functiongemma:270m` without KV reuse, and `functiongemma:270m` with KV reuse;

for Qwen3 the answer prompt grows from 200 to 553 tokens and the end-to-end time from 8.64 s to 11.09 s. Table 22 compares the answer-stage TTFT of the three configurations, and Figure 16 plots it against the answer prompt length.

Table 19. Leader latency of `Qwen3:1.7b` on the Hailo as the number of aggregated replies grows (single tool). Values are mean ± standard deviation over ten queries. The two-stage columns sum to E2E; tool→result (the in-process build of the reply set, no network hop) is negligible (< 0.1 ms) and omitted.

Replies	Answer Tokens in	Answer Tokens out	Query→Tool [s]	Result→Reply [s]	E2E [s]
1	200	8.8	4.96 ± 0.00	3.68 ± 0.29	8.64 ± 0.29
5	272	21.6	4.96 ± 0.00	6.32 ± 2.04	11.28 ± 2.04
10	363	11.1	4.96 ± 0.00	4.77 ± 1.18	9.73 ± 1.18
15	458	13.9	4.96 ± 0.00	5.97 ± 1.52	10.93 ± 1.52
20	553	11.7	4.96 ± 0.00	6.13 ± 1.05	11.09 ± 1.05

Table 20. Leader latency of `functiongemma:270m` on the STM32 CPU without KV reuse (`llm.reset()` before every generation, so each query is an independent cold prefill) as the number of aggregated replies grows (single tool). Values are mean ± standard deviation over ten queries; tool→result is omitted as above, so query→tool + result→reply = E2E.

Replies	Answer Tokens in	Answer Tokens out	Query→Tool [s]	Result→Reply [s]	E2E [s]
1	161	17.2	9.11 ± 0.12	16.23 ± 0.99	25.34 ± 1.06
5	233	30.8	9.12 ± 0.12	24.80 ± 4.50	33.91 ± 4.52
10	324	44.9	9.11 ± 0.12	38.08 ± 6.94	47.19 ± 7.02
15	419	106.6	9.11 ± 0.13	60.93 ± 38.10	70.04 ± 38.08
20	514	71.8	9.10 ± 0.12	61.38 ± 24.10	70.48 ± 24.11

Table 21. Leader latency of `functiongemma:270m` on the STM32 CPU with KV-cache reuse (the cache is never reset across the run) as the number of aggregated replies grows (single tool). Values are mean ± standard deviation over ten queries; tool→result is omitted as above. Only the run's first query, in the 1-reply row, pays a cold prefill, which raises that row's query→tool mean and spread; all later queries reuse the cache.

Replies	Answer Tokens in	Answer Tokens out	Query→Tool [s]	Result→Reply [s]	E2E [s]
1	125	14.9	3.06 ± 1.91	5.54 ± 1.63	8.60 ± 2.86
5	197	53.2	2.46 ± 0.12	19.26 ± 6.83	21.72 ± 6.83
10	288	53.7	2.45 ± 0.12	28.59 ± 8.61	31.05 ± 8.66
15	383	102.9	2.46 ± 0.12	48.00 ± 19.41	50.46 ± 19.41
20	478	155.4	2.46 ± 0.12	69.00 ± 30.66	71.46 ± 30.69

Table 22. Answer-stage TTFT [s] (mean ± standard deviation over ten queries) as the number of aggregated replies grows, for the three configurations. functiongemma is measured without KV reuse (reset) and with KV reuse of the fixed prompt prefix.

Replies	Qwen3:1.7b, Hailo [s]	Functiongemma, Reset [s]	Functiongemma, KV Reuse [s]
1	1.85 ± 0.00	12.65 ± 0.12	2.46 ± 0.01
5	1.86 ± 0.00	18.24 ± 0.12	8.04 ± 0.02
10	2.47 ± 0.00	28.21 ± 0.12	16.83 ± 0.02
15	3.09 ± 0.00	36.42 ± 0.13	25.07 ± 0.04
20	3.71 ± 0.00	44.78 ± 0.18	33.28 ± 0.02

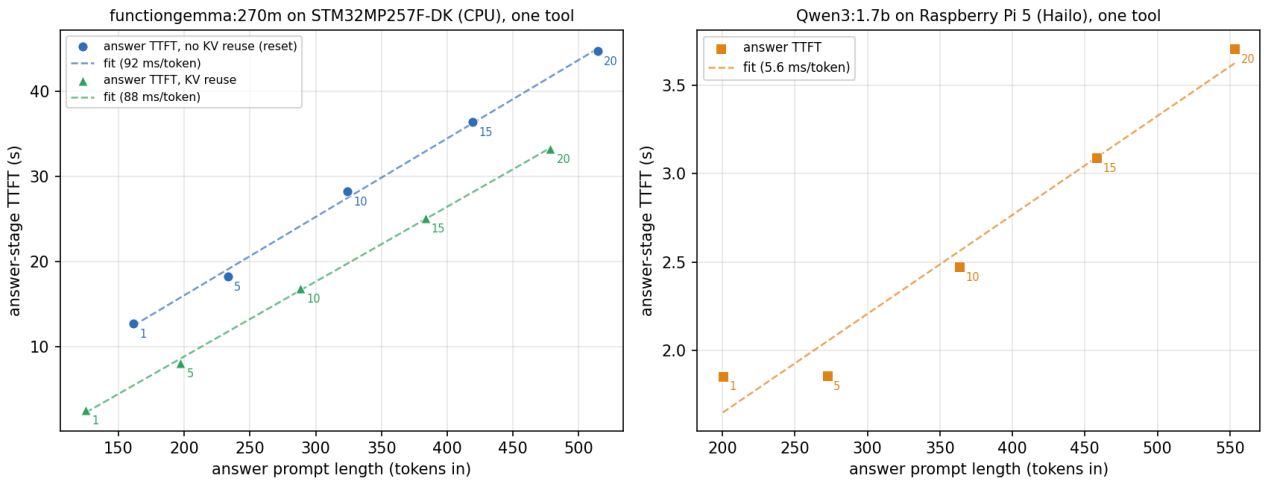


Figure 16. Answer-stage TTFT (final-reply prefill) against the answer prompt length for a single tool and a growing number of aggregated replies. Left: functiongemma on the STM32 CPU, without KV reuse (reset, circles) and with KV reuse (triangles). Right: Qwen3:1.7b on the Hailo. The number of aggregated replies is annotated on each point; dashed lines are least-squares fits. On the STM32, the two lines are nearly parallel: KV reuse lowers the intercept by a fixed amount but leaves the per-token slope unchanged.

7.2.3. Failover Time

This subsection characterizes the cost of a leader failure: the probability that an in-flight query is lost, and the time the network needs to return to service under a new leader. It also explains how the failover of a sensing device impacts the answer.

Query Loss Probability

The conversational state is replicated to the device with the second highest score (Section 5.1) as soon as the user query is received. The backup transfer between the two tested devices, measured over the same ten test queries, takes $t_b = 8.865$ ms on average (standard deviation 2.364 ms). The relatively large standard deviation shows the transfer time can vary, but for the tested queries, it remained within the same order of magnitude. This transfer window is the only interval during which a leader failure loses the pending query: under the assumed failure model, defined in Section 2.3, failure at any other point either finds the state already replicated or occurs while no query is pending. A new query could, in principle, be lost if the user sends it right after a leader failure, during the election window. This is heavily implementation-dependent: in the proposed implementation, it does not happen because Telegram queues undelivered queries. Simply checking whether the query has been delivered to the leader before discarding it would prevent a new query from being lost during the election window. That said, if a leader fails during the reply to a query, assuming a failure instant is uniformly distributed over the end-to-end duration t_{E2E} of the query itself, the probability that it also causes the loss of that query is defined by Equation (6).

$$P_{\mathrm{loss}} = \frac{t_b}{t_{\mathrm{E2E}}}. \tag{6}$$

Using the mean end-to-end latencies measured with one tool and one reply (Table 18), Equation (6) gives the following probabilities:

- $P_{\mathrm{loss}} \approx 8.5 \times 10^{-4}$ for `Qwen3:1.7b` on the Hailo ($t_{\mathrm{E2E}} = 10.44$ s),
- $P_{\mathrm{loss}} \approx 3.3 \times 10^{-4}$ for `functiongemma:270m` without KV cache ($t_{\mathrm{E2E}} = 26.79$ s), and
- $P_{\mathrm{loss}} \approx 9.1 \times 10^{-4}$ for `functiongemma:270m` with a warm KV cache ($t_{\mathrm{E2E}} = 9.78$ s).

In all three configurations, fewer than one leader failure in a thousand (occurring mid-query) would also lose the query being served. Note that slower configurations exhibit a lower loss probability only because the fixed-size backup window is diluted over a longer serving time.

Recovery Pipeline

Recovering from a leader failure involves two independent stages and is heavily dependent on time-related hyperparameters. The following calculations refer to the realized implementation.

In the first stage, the failure must be detected: the leader proves liveness by broadcasting a HELLO message every 2 s, and every node's watchdog declares a peer dead after 15 s of silence (i.e., after at least seven skipped heartbeats). Detection therefore takes between 15 and 17 s from the leader's last HELLO. In the second stage, the surviving leader-capable nodes broadcast their candidacies and wait a fixed 2 s election window, which lets the mutual score announcements converge on all nodes before each independently resolves the same total order (score, then agent-ID); the best-scoring survivor then claims leadership. The election itself is thus essentially the 2 s window, and the full detection-plus-election time is 17 to 19 s. Detection dominates: the 15 s timeout could in principle be reduced to as little as the 2 s heartbeat period, and further still with a faster heartbeat rate (the measurements showed that network traffic is not a bottleneck). However, in this implementation, the detection timer is kept at 15 s because the HailoRT bindings hold the Python (version 3.11.9) GIL for the whole duration of model loading and generation (Appendix H), suppressing the heartbeats of a healthy but busy leader; the conservative timeout is what makes detection stable under this behavior. This is a limitation of the tested Hailo software stack rather than of the architecture: on devices serving the model through `llama.cpp`, the timeout could be lowered considerably by running the Hailo stack on a different process.

Total Time Lost on a Failure

Practically, the service interruption experienced by a user (whose query is being served when the leader dies) does not depend only on the two aforementioned stages. The actual service interruption is the sum of four terms:

1. the time already spent by the old leader on the query before crashing, which is arbitrary as it depends on when the failure occurs relative to the time needed for the completion of the reply;
2. the detection and election time of 17 to 19 s discussed above;
3. the model load time on the new leader, reported in Table 11 (which could be removed from the critical path by keeping the model of the backup leader warm, at the cost of some additional power consumption);
4. the time to re-serve the query, which depends on the number of exposed tools and aggregated replies as characterized in Section 7.2.2.

Failures of Sensing Agents

Assuming the single fail-stop failure model defined in Section 2.3, the loss of any sensing agent, even if it is the backup agent, cannot cause query loss: this is trivial because as soon as the query is backed up on the backup agent, it means it is stored on two devices (the backup leader and the leader itself). If a sensing agent, hosted on a device acting as the backup leader, fails, the backup role is simply reassigned to the next candidate with the highest score (if there were at least 3 leader-capable devices online): the new backup device receives the conversational state from the leader.

Sensing agents' failure can lose the conversational state (potentially causing query loss) only assuming a non-single-node failure model. The mechanisms that deal with fault tolerance in the proposed implementation are not strictly dependent on the single-node failure model and allow for more flexibility and additional redundancy. Failure detection runs one independent watchdog per peer, so any number of concurrent fail-stop failures (from any agent) is caught in the same way in which leader failure is detected in Section Recovery Pipeline. The following characterization of the behavior beyond the stated follows from the per-peer and per-call structure of the implemented mechanisms. Only single-failure scenarios were tested (according to the declared failure model).

The election operates independently on the set of online candidates, so the simultaneous loss of several leader-capable devices (that are possibly hosting sensing agents) only shrinks that set: if there are no leader-capable devices, the candidate set D_{cand} is empty, and no election is possible. As long as one leader-capable device remains, the leader can fail, and the system will recover as described in Section Recovery Pipeline. If the leader fails and there is no other leader capable of taking over, the network goes offline.

If both the leader and the backup holder fail during the reply to a query, then a query loss occurs. However, even if outside of the considered failure model, the implementation supports replicating the state to the N highest-scoring leader-capable devices (LeaderLogic/backup_manager.py) for further redundancy. The measured backup transfer t_b and the per-agent traffic of Table 13 indicate a small marginal cost per replica. After an election, the new leader restores the state as described in Section Recovery Pipeline. In that case, the conversational state is lost only if the leader and all current replica holders fail before the state has been re-replicated to a new holder. N can be equal to the number of all the leader-capable devices; then the query is lost only if all the leader-capable devices fail. Multi-replica operation is implemented but was not evaluated experimentally, since the evaluation follows the single-failure model defined in the problem statement.

The simultaneous failure of any number of sensing-only devices, not holding the conversational state, never compromises service availability and cannot cause query loss. A detected failure triggers the removal logic depicted in Figure A9: a tool is removed from the leader's tool list only when no surviving agent exposes it. An undetected failure (i.e., a device is offline but the elapsed time is not yet sufficient for it to be considered lost) is absorbed by the dispatch logic. After issuing a tool call, the leader waits at most 2 s for the results and returns early once every agent exposing the called tool has replied. Each missing result is replaced by an explicit non-response marker for the corresponding agent (e.g., "AgentID did not reply"), and the markers are forwarded to the language model together with the actual tool results, which summarize the results in the reply. A query served while any number of sensing agents have failed undetected therefore completes with at most 2 s of additional latency.

The failure-bearing boundary of the architecture is therefore structural rather than numerical: it depends on which roles fail, not on how many devices fail. Under the fail-stop model, it can be stated as follows.

- Availability: the system serves queries as long as at least one leader-capable device remains connected. The minimal configuration is a single leader-capable device hosting both a leader agent and a sensing agent, which degrades the service to single-sensor operation. If every leader-capable device fails, the conversational service is offline, since sensing agents alone cannot serve queries.
- Analysis capability: a tool remains available as long as at least one connected agent exposes it, so the redundancy of each capability equals the number of agents exposing the corresponding tool. The failure of a sensing agent trivially removes its own sensor

from the network: the physical coverage of that sensor is lost even if the corresponding tool remains callable on other devices.

- Conversational state: if the replication factor is set equal to the number of leader-capable devices, the pending query and the chat history survive any combination of failures that leaves the leader or at least one replica holder alive. They are lost only when the leader and all replica holders fail before re-replication.
- Recovery: recovery from any tolerated combination of failures follows the detection, election, model-load, and re-serve pipeline quantified in Section Recovery Pipeline, since every stage operates on the surviving peer set.

Correlated failures caused by network partitions do not follow the fail-stop model and are not addressed here; together with adversarial behavior (Section 9.5), they are left to future work.

8. Discussion

This section interprets the experimental results. The objective of this section is to make explicit what the measurements show about deploying agentic tool-calling on heterogeneous edge devices. The discussion follows the structure of the experiments: accuracy of the agentic pipeline (Section 8.1, covering Benchmarks A, B and C), power consumption (Section 8.2) and latency (Section 8.3).

8.1. Accuracy

8.1.1. Benchmark A: Degradation with Pool Size Is Strongly Model-Dependent

Accuracy drops caused by distractor tools, documented in the literature for models of at least 8 billion parameters [35], are confirmed here at sub-2-billion scale, and their severity varies drastically with the model. Growing the pool from 4 to 24 tools halves the selection accuracy of functiongemma:270m (0.81 $\rightarrow$ 0.41), while Qwen3:1.7b loses 2 points with thinking enabled (0.95 $\rightarrow$ 0.93) and 4 points without (0.93 $\rightarrow$ 0.89), as reported in Table 3. Abstention remains comparatively robust for all configurations (0.80 to 1.00 at 24 tools). The practical consequence for the proposed architecture is that the number of tools exposed by the network is a first-class design parameter whenever the leader runs on a microprocessor-class device (such as the STM32MP257FDK). The model functiongemma:270m is dependable only on small pools, whereas Qwen3:1.7b absorbs the full 24-tool set with marginal loss.

8.1.2. Thinking Tokens Do Not Translate into Proportional Accuracy

Across all three benchmarks, the reasoning mode of Qwen3:1.7b produces large token overheads for small accuracy returns. In Benchmark A the thinking variant generates 161.7 tokens per question against 21.1 ($\approx$7.7$\times$) to gain 0.04 in selection and 0.07 in tool-call accuracy at 24 tools; in Benchmark B the two variants are essentially equivalent (0.950 vs. 0.925 at 16 tools); in Benchmark C the mean grade improves by at most 0.36 points while generating between 1.7$\times$ and 4$\times$ more tokens (Table 5). On target, at the 4.88 TPS measured on the Hailo, the $\sim$140 additional tokens per question observed in Benchmark A would add roughly 29 s to the reply, alone exceeding the 20 s responsiveness requirement of Section 2.2. The non-thinking configuration is therefore the appropriate operating point for this architecture: reasoning tokens buy robustness margins that the latency budget cannot afford.

8.1.3. Benchmark B: Zero-Parameter Tool Selection Is Largely a Retrieval Problem

The Jina reranker (278 million parameters) outperforms the similarly sized functiongemma:270m at every pool size: e.g., 0.847 against 0.506 at 16 tools, as seen in Figure 8.

It matches the six-times-larger Qwen3:1.7b at the smallest pool (0.956 vs. 0.966 thinking, above the non-thinking 0.922) and trails it by at most 0.10 at 16 tools (0.847 vs. 0.950). It degrades under distractor pressure (0.956 $\rightarrow$ 0.847), but far more slowly than the equally sized LLM. This validates Approach 2 as a credible complement: when the exposed tools take no arguments, and the sensing device can host a $\sim$300M encoder, dispatching can be moved off the leader entirely, reserving the LLM for the final natural language reply.

8.1.4. Benchmark C: Aggregation Is the True Bottleneck of the Smallest Model

Given the 480 tool-bearing test queries × 5 size of reply-subsets (1, 5, 10, 15, 20) × 3 configurations (`functiongemma:270m`, `qwen3:1.7b No thinking`, `qwen3:1.7b Thinking`), a total of 7200 natural language replies were evaluated by the LLM judge. Out of those 7200, only 20 were marked for human annotation when the judge LLM failed: they were all graded 1 out of 10 by a human annotator since those 20 replies were characterized by `functiongemma:270m` getting stuck in a loop, generating nonsensical tokens. Those tokens also filled up the context window of the LLM judge. Of the twenty evaluated replies, where N is the size of the reply-subsets, one corresponded to $N = 5$, five to $N = 10$, ten to $N = 15$, and four to $N = 20$. This, together with the high standard deviation of the mean generated tokens at $N = 15$, confirms functiongemma getting stuck in a loop as an explanation for the spike of generated tokens at a reply size of 15 visible in Figure 11. Further studies could find out the reason why it mainly happens at 15. Beyond this failure mode, grades also degrade with the number of function replies for every configuration, but with very different slopes: the mean grade of functiongemma:270m halves from 7.65 to 3.81 (median from 9 to 4) between $N = 1$ and $N = 20$, while both Qwen3:1.7b variants lose about one point (8.94 $\rightarrow$ 7.84 and 9.07 $\rightarrow$ 8.15) and keep a median of 9 throughout. Therefore, having both more exposed tools and more replying devices degrades a microprocessor-class leader independently: deployments relying on functiongemma:270m should bound not only the tool pool but also the number of replies aggregated per query, for example through hierarchical summarization.

8.1.5. The Accuracy Results Are Not an Artifact of the Query-Generating LLM

Since every test query was generated by a single model (Claude Opus 4.7), an ablation was performed, reported in Appendix I. It confirmed that the reported accuracies are a property of the benchmarked models rather than of that generator's phrasing. The queries were regenerated through two additional LLMs (ChatGPT5.6-sol [12 July 2026] and GLM5.2 [12 July 2026]), maintaining the same instructions (same system prompt, same tools, parameters, abstention cases, fixed-seed pools, and tool replies). This means that only the surface phrasing differs.

The benchmark is essentially unchanged: the mean accuracy range across the three generators is 2.5 percentage points in Benchmark A and 3.1 in Benchmark B, the benchmarked-model ranking is preserved in both, and a paired item-level McNemar test is significant in only 1 of the 54 (3 tested models $\times$ 3 generators $\times$ 6 pool sizes) tests of Benchmark A and 1 of the 48 (4 tested models $\times$ 3 generators $\times$ 4 pool sizes) of Benchmark B. A variance decomposition attributes at most 0.2% of the per-cell accuracy variance to which LLM phrased the query, against at least 76% to which the model is benchmarked. One measurable phrasing effect appears on the Benchmark C summarization grades (a significant shift in 12 of 45 cells, with GLM5.2 graded marginally higher), but remains far below the model-to-model differences. The full ablation is reported in Appendix I. This shows that the main findings reported in this paper do not depend on the LLM used to generate the queries. Perturbing the queries with realistic surface noise (typos, abbreviations, dropped function

words) lowered selection accuracy by 5.7 percentage points on average and preserved the model ranking (Appendix I.5).

8.2. Power Consumption Discussion

Three lessons emerge from Tables 7–9. First, the leader role itself is energetically cheap. Subtracting the idle baseline, hosting functiongemma:270m costs about 0.50 W on the STM32MP257FDK (4.81 vs. 4.30 W mean) and hosting Qwen3:1.7b costs 1.23 W on the Hailo (6.99 vs. 5.76 W) or 1.73 W on the Raspberry Pi 5 CPU. The energy gap introduced by a leader election is thus below 2 W in the worst-measured case: failover decisions in the proposed architecture can be driven by accuracy and latency considerations rather than by power.

Second, accelerators improve energy efficiency by far more than their raw power draw suggests. To fairly compare the energy consumption of different hardware backends, we isolate the net dynamic power actually consumed by the workload by subtracting the platform's idle power baseline (P_{idle}) from the mean power under load (P_{mean}). Dividing this by the throughput yields the net energy per inference, as shown in Equation (7).

$$E_{\text{unit}} = \frac{P_{\text{mean}} - P_{\text{idle}}}{\text{Throughput}} \quad [\text{J/frame}] \text{ or } [\text{J/token}] \tag{7}$$

Applying this metric using the power statistics from Table 9 reveals the profound impact of dedicated accelerators on energy efficiency (Tables 23 and 24). On the Raspberry Pi 5, moving YOLOv8n:640 from the CPU to the Hailo accelerator simultaneously lowers mean power and raises throughput, reducing the energy per frame from 1.00 to 0.058 J/frame: a 17.3× improvement. For Qwen3:1.7b, the Hailo delivers 42% higher throughput at 0.5 W less, halving the energy per token from 0.50 to 0.25 J/token (2.0× gain). The most counterintuitive result appears on the STM32MP257FDK: the NPU draws marginally more power than the CPU (5.43 vs. 5.29 W total), yet because it provides 11× the frame rate, it drops the energy per frame from 0.60 to 0.062 J/frame: a 9.6× reduction, again an order-of-magnitude gain.

Table 23. Net energy per inference across hardware backends. Net power is derived by subtracting the platform's idle baseline (P_{idle}) from the mean power under load.

Platform	Workload	Net Power [W]	Throughput	Net Energy
RPi 5 + Hailo-10H	YOLOv8n:640 (CPU)	$9.167 - 5.761 = 3.406$	3.39 FPS	1.00 J/frame
	YOLOv8n:640 (Hailo)	$6.918 - 5.761 = 1.157$	19.95 FPS	0.058 J/frame
	Qwen3:1.7b (CPU)	$7.490 - 5.761 = 1.729$	3.43 TPS	0.50 J/token
	Qwen3:1.7b (Hailo)	$6.987 - 5.761 = 1.226$	4.88 TPS	0.25 J/token
STM32MP257FDK	YOLOv8n:320 (CPU)	$5.287 - 4.303 = 0.984$	1.65 FPS	0.60 J/frame
	YOLOv8n:320 (NPU)	$5.434 - 4.303 = 1.131$	18.24 FPS	0.062 J/frame

Table 24. Energy efficiency ratios when moving from CPU to dedicated accelerators.

Model	J/Inference (CPU)	J/Inference (Accelerator)	Ratio
Raspberry Pi 5 (YOLOv8n:640)	1.00 J/frame	0.058 J/frame	$\sim 17.3\times$
Raspberry Pi 5 (Qwen3:1.7b)	0.50 J/token	0.25 J/token	$\sim 2.0\times$
STM32MP257FDK (YOLOv8n:320)	0.60 J/frame	0.062 J/frame	$\sim 9.6\times$

Third, on the microprocessor-class device, the power consumption hits a limit at 5.44 W (mean) as YOLOv8n:320 consumes almost as much as the same YOLOv8n:320 together with functiongemma:270 m. Contention manifests as throughput loss rather than additional watts. Adding functiongemma:270m to NPU detection leaves mean power unchanged (5.44 vs. 5.43 W) while the LLM drops from 4.28 to 1.24 TPS and detection

from 18.24 to 11.20 FPS (Table 10). Power measurements alone are therefore insufficient for provisioning such nodes and must be read together with the sustained FPS and TPS.

On the Raspberry Pi 5, the two dual-role configurations expose a clear latency-versus-power trade-off. Hosting detection on the Hailo and the LLM on the CPU is the cheapest combination (8.53 vs. 10.43 W), because CPU detection is by far the most power-hungry workload (3.41 W net against 1.73 W for the LLM); hosting the LLM on the Hailo instead preserves nearly its standalone speed (4.83 vs. 4.88 TPS, where the CPU-bound alternative falls to 2.61 TPS) at a cost of almost 2 W. In both cases, the accelerator insulates whichever workload it hosts. The most energy-efficient dual-role node is detection-on-accelerator; the most responsive leader is LLM-on-accelerator. The choice depends on whether the node is battery- or latency-constrained.

8.3. Latency and Scalability Discussion

The latency measurements of Section 7.2.2 show that the pipeline is prefill-bound: the TTFT of both stages grows linearly with the prompt length (Figure 15). Thus, the two quantities that the architecture exposes to the leader, the number of tools and the number of replies, directly set the response time. This finding is analyzed in Section 8.3.1; other aspects related to latency and scalability limitations are discussed in Section 8.3.2.

8.3.1. Why TTFT Rises Steeply on the STM32MP257FDK

Patel et al. [39] characterize LLM inference as consisting of a compute-intensive prompt-processing phase and a memory-intensive token-generation phase. For a dense decoder-only model with P parameters, parameter-dependent matrix multiplications require approximately $2P$ FLOPs per processed token [40]. Therefore, prefilling a prompt containing L tokens requires approximately $2PL$ FLOPs, excluding the additional quadratic attention cost. Across the measured prompt-length range, TTFT grows approximately linearly, indicating that the parameter-dependent term dominates the observed latency. Since every exposed tool contributes its complete JSON schema to the prompt, increasing the tool-pool size directly increases the required prefill computation; the same is true for increasing the number of replies. Each tool adds approximately 29 tokens, producing a corresponding linear increase in latency.

The measured slopes quantify this effect. On the STM32MP257FDK, `functiongemma:270m` requires approximately 92 ms per prompt token, whereas on the Hailo-10H, `Qwen3:1.7b` requires approximately 6.1 ms per prompt token (similar values were computed for both the dispatch and the answer stages). The STM32MP257FDK generates at 4.28 TPS, corresponding to approximately 234 ms per generated token. Thus, the cost of prefilling one token is $92/234 \approx 39\%$ of the cost of generating one token. The Hailo-10H generates at 4.88 TPS, corresponding to approximately 205 ms per generated token, so the cost of prefilling one token is $6.1/205 \approx 3\%$ of the cost of generating one token. The relative prefill cost is therefore approximately $13\times$ higher on the STM32MP257FDK. This comparison is affected by the differences of the tested platforms, including their different models, tokenizers, hardware, and inference backends. The tests show that, on the STM32MP257FDK, the compute-intensive prefill phase contributes more substantially to latency than the memory-intensive decoding phase.

The STM32MP257FDK includes an NPU that does not support LLM execution. Language-model inference therefore ran through `llama-cpp-python` on the two in-order Cortex-A35 cores operating at 1.5 GHz, which also execute the operating system and coordination software. By contrast, the Hailo configuration ran `Qwen3:1.7b` through a model-specific backend on a dedicated inference accelerator rated at up to 40 TOPS in INT4.

The combination of limited CPU throughput and the lack of supported LLM offloading explains why dispatch TTFT rises much more steeply on the STM32MP257FDK.

Additionally, changing the system prompt between the dispatch and answer stages required the tool schema to be prefilled again for every query. Retaining a single system prompt enabled KV-cache reuse and eliminated this repeated prefill cost. The fixed tool-schema prefix is prefilled once and retained, so each subsequent query processes only the uncached query-specific tokens. Caching prevents the same schema tokens from being recomputed for every request. Consequently, warm dispatch TTFT no longer grows substantially with the size of the tool pool, as shown in Table 17.

For the tested STM32MP257FDK configuration, this reuse is required to satisfy the 20 s latency constraint specified in Section 2.2. The same mechanism suggests that a backup leader could load its model and prefill the current tool-schema prefix before an election, thereby mitigating the latency caused by the required initial prefill stage, although doing so would require a different implementation and incur additional power consumption. Caching, however, does not reduce the cost of processing a new token. Thus, the first query after a cache reset still incurs the complete cold-prefill latency.

8.3.2. Scalability Limitations

On the Hailo, `Qwen3:1.7b` remains within the 20 s responsiveness requirement across the whole range, reaching 17.19 s at the full 24-tool pool (Table 18) with a single reply. On the STM32MP257FDK running `functiongemma:270m` without KV-cache reuse, the full-prompt configuration violates the 20 s bound already with a single tool (26.79 s) and reaches 86.96 s at 24; reusing the KV cache of the fixed prompt prefix removes the repeated tool prefill and keeps the end-to-end latency flat at about 10–11 s across every pool size, below the requirement and comparable to the Hailo-accelerated leader.

On the reply axis (Table 22), the same conclusion holds with one caveat: the cached prefix does not cover the tool results themselves (because they change between queries), so the answer-stage TTFT still grows with the number of aggregated replies. Thus, on the STM32MP257FDK, the end-to-end latency exceeded the 20 s bound already at five aggregated replies: such leaders therefore require the reply count per query to be bounded, consistently with the accuracy findings of Benchmark C. Connection-wise, scaling the number of agents is cheap: power stays almost flat up to 100 co-located agents (Table 12) and the leader's base traffic grows linearly at about 1.8 KiB/min per agent (Table 13), so discovery overhead is not a bottleneck at household scale.

8.4. Failover

The failover analysis of Section 7.2.3 shows that resilience is dominated by detection rather than by the election itself, and that the conservative 15 s liveness timeout is a consequence of the GIL limitation of the tested Hailo bindings (Appendix H) rather than of the architecture. The failure of a sensing agent mid-query can add at most 2 s of delay.

9. Conclusions and Future Works

This work investigated whether a heterogeneous network of resource-constrained edge devices can support real-time multi-sensor analytics and conversational interaction without relying on cloud infrastructure while remaining resilient to single-node failures.

The use of hardware accelerators increased inference energy efficiency between 2.0 and 17.3 times across the tested workloads, as reported in Table 24, while leader execution introduced less than 2 W of additional power consumption in every measured configuration, as reported in Section 7. The results also show that STM32MP257FDK boards are more inclined to throughput degradation caused by resource contention rather than increase in

power consumption: mean power remained within 0.01 W between the configuration running only detection on the NPU and the configuration running the combined detection on the NPU and LLM on CPU, while the detection frame rate dropped from 18.24 to 11.20 FPS.

From the perspective of generative reasoning, Qwen3-1.7B achieved the highest tool-selection accuracy among the tested language models and provided robust summarization capabilities. The comparison between small LLMs and the jina-reranker-v2-base-multilingual suggests that, for a limited number of tools, tool selection behaves largely as an information retrieval problem. This observation indicates that lightweight retrieval models may complement or partially replace LLM-based dispatch in some edge scenarios.

Latency tests showed that preserving the KV cache across calls significantly reduces TTFT.

Regarding resilience, the failover analysis showed that the probability of losing an in-flight query upon a leader failure is below one in a thousand in every tested configuration, since the conversational state is exposed to loss only during the $\sim$9 ms backup transfer performed at the reception of each query. The recovery time is dominated by failure detection (15 s timeout plus up to 2 s of watchdog tick). The detection timeout is kept conservative because the HailoRT Python bindings hold the GIL during model loading and generation, suppressing the heartbeats of a busy but healthy leader; on backends such as `llama.cpp`, this timeout could be lowered considerably, and future work could remove the limitation entirely by isolating accelerator inference in a separate process. Overall, a leader failure translates into a bounded service interruption rather than a loss of service or of conversational state.

The results demonstrate the feasibility of the proposed architecture on a small two-node prototype, comprising a Raspberry Pi 5 with a Hailo-10H accelerator and an STM32MP257FDK board. Within this setting, heterogeneous devices cooperated to serve user queries entirely on edge while preserving conversational state under single-node failure. These findings are evidence of feasibility for the tested prototype and workload under the tested conditions: larger fleets, other hardware platforms, real (non-synthetic) user traffic, multi-tool queries, and adversarial conditions were not evaluated.

This study also has several limitations. The hardware evaluation was performed on a small number of devices, and the leader-selection score was introduced only as a practical heuristic: the specific weights and multiplier were set heuristically and are not claimed as an optimized solution. Future work could include a sensitivity analysis of the weights and validation of the score across a broader range of devices to assess its generalisability. Furthermore, the benchmarks focused solely on single-tool dispatch.

Another limitation is that the evaluation dataset is synthetic, generated by a single model (Claude Opus 4.7), so it may inherit that model's stylistic and lexical conventions. To test whether this affects the results, the dataset was regenerated through two additional LLMs, and the findings resulting from the benchmark were found to be essentially invariant to the query generator (Appendix I); this ablation isolates phrasing bias but does not replace validation against real user logs, which is left to future work.

Future work will investigate larger and more heterogeneous networks, dynamic workload balancing (at present, only available RAM is retrieved dynamically), support for parameterized retrieval-based tool selection, and mesh-network deployments without external infrastructure. Additional research is also needed to study how tool naming, descriptions, and semantic similarity influence tool-calling accuracy, potentially enabling automated optimization of tool definitions for edge agentic systems. The purpose of this study was solely to propose the architecture and verify its feasibility. Considerable effort was devoted to measuring how LLM inference (required for approach 1) affects the power consumption of the tested devices. That said, some topics remain unexplored and could benefit from future work: some of them are proposed below.

9.1. Constrained Decoding Increases Accuracy

Preliminary tests on functiongemma:270m using llama-cpp-python and a custom handler enforcing constrained decoding showed an accuracy increase of 7 percentage points at the 24-tool pool size. This configuration was not used for the benchmarks because enforcing the grammar (i.e., artificially setting the undesired logits to zero) increased the inference time by approximately 7× on the computer used for benchmarking. Further studies could focus on faster implementations of constrained decoding, which might be particularly useful for small models. Results are reported in Table 25.

Table 25. FunctionGemma handler comparison on the full tool set. Selection (Sel) and tool-call (TC, selection+params) accuracy on the tool-bearing subset; cold-start latency on the first sentence (Cold), total wall-clock time (Total), and mean generated tokens per sentence (Tok). Cached_grammar_handler saves the grammar on the first sentence instead of recomputing it for each query; caching led to no improvement in inference time.

Handler	Sel (Accuracy)	TC (Accuracy)	Cold_start (s)	Total (s)	Tokens
grammar_handler	0.47	0.44	0.61	374.40	11.20
cached_grammar_handler	0.47	0.44	0.63	378.20	11.20
simple_handler	0.39	0.37	0.13	50.50	17.30

9.2. Fine-Tuning Improves Accuracy

A large body of literature has shown that fine-tuning the model increases accuracy [12,14,32]. Fine-tuning LLMs and IR could considerably improve the measured accuracy. Future work could provide a detailed analysis of selected LLMs to reach higher performance through various fine-tuning techniques.

9.3. In-Depth Analysis Also on Approach 2

As seen from the first benchmarks on approach 2, the results are promising. Further studies might analyze retrievers in similar settings, comparing them with similarly sized LLMs. It is important to note that, due to current constraints of small LLMs, approach 1 focuses on single tool-calls only, since small models generally exhibit fast accuracy degradation in multi-tool calls. Approach 2 might execute multi-tool calls more reliably instead. This can be the case even when the re-ranker chooses only the first result for every device, since different devices with different sets of tools can autonomously pick different tools. A large number of tools (running on various devices) that do not require passing in arguments might benefit from approach 2 if the devices are powerful enough to run an IR model. A smaller number of tools that require custom arguments, being executed on a less capable device, must instead rely on approach 1. A comparison of how the two approaches scale could be insightful. Another comparison could be done using a DistilBERT classifier (similar to the one developed by Erdogan et al. [12]).

9.4. Tool Similarity as a Reference Metric

As previously discussed in Section 3, Kokane et al. [34] found that LLMs' tool calls tend to degrade in accuracy when dealing with pools of tools with a high pairwise Jaccard similarity. This finding is informative, but their approach has several limitations. It considers only names, ignoring the description, parameter schema, and argument types, all of which are part of the specification the model receives at inference time. That metric is lexical rather than semantic: synonymous names with no shared tokens score as dissimilar despite being functionally interchangeable. It could be helpful to calculate the semantic similarity of the tool name and description using a cross-encoder and/or a bi-encoder, while comparing it with a Jaccard-based baseline. The results of this evaluation would be relevant for this study because they could inform the choice of the tool names and descrip-

tions. Those selection-facing names could be chosen to minimize similarity, potentially maximizing performance. This would benefit from additional research, but it is out of scope for the current study, whose purpose is solely to introduce and benchmark the proposed architecture. Some future work could focus on expanding the concept of similarity between tools, possibly using a cross-encoder Semantic Text Similarity (STS) model.

9.5. Hardening the Security and Privacy Model

The security model described in Section 5.2 relies on network-level perimeter control rather than authentication, authorization, or encryption at the application layer. Future work should replace this perimeter-trust assumption with per-agent authentication and encrypted local transport, for instance via mutual TLS or a pre-shared-key scheme. Tool-level authorization is also an open problem: the current implementation allows any agent admitted to the network to invoke any tool, with no restriction on which leader-capable devices may invoke them. Similarly, detecting compromised or malicious nodes from their network behavior is not addressed, as the leader election and failover mechanisms assume benign rather than adversarial failures. Explainable intrusion detection systems developed for IoT networks, such as the one proposed by Ahmad et al. [41], offer a starting point for adapting anomaly detection to the heterogeneous agents considered here. A mesh-network deployment would additionally lack the single admission-control point provided by the access point in the evaluated configuration, requiring its own provisioning mechanism, for example, pairwise pre-shared keys distributed at setup time.

Author Contributions: Conceptualization, L.L.Z., M.L., M.B., D.P.P.; methodology, L.L.Z., M.L., M.B., D.P.P.; software, L.L.Z.; validation, L.L.Z.; formal analysis, L.L.Z., M.L., M.B., D.P.P.; investigation, L.L.Z., M.L., M.B., D.P.P.; resources, L.L.Z., M.L., M.B., D.P.P.; data curation, L.L.Z.; writing—original draft preparation, L.L.Z., D.P.P.; writing—review and editing, L.L.Z., M.L., M.B., D.P.P.; visualization, L.L.Z.; supervision, D.P.P.; project administration, D.P.P.; funding acquisition, D.P.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The GitHub repo containing the code of the proposed architecture and the code used to benchmark it is available on: https://github.com/Gigizap/heterogeneous-edge-ai-network. The BFCL was visualized at the link: https://gorilla.cs.berkeley.edu/leaderboard.html. Ollama functiongemma BFCL scores were retrieved from: https://ollama.com/library/functiongemma. The ToolRet leaderboard was visualized at: https://huggingface.co/spaces/mangopy/ToolRet-leaderboard. Last accessed on 9 June 2026. The benchmark code and data are available in the "BENCHMARK CODE" folder in the first Repo linked. This folder has 3 subfolders inside it: "POWER CONSUMPTION" (for power consumption benchmarks), ACCURACY (accuracy benchmarks: contains generated datasets, answers, and tools: dataset.json, dataset_tools_w_replies.json, tools.json), and "FUNCTIONGEMMA TEST HANDLER" (test supporting the last claim in Conclusions about constrained decoding for tool-calling). README.md files are provided in the repo for further details.

Acknowledgments: During the preparation of this manuscript/study, the author(s) used [Claude opus 4.7, 28 May 2026] for the purposes of [generating 500 human-like queries to benchmark the approach, coding assistant for the implementation of the software (not for conceiving it), bug correction in the software], [Claude sonnet 4.6, 22 April 2026] for the purposes of [retrieving few relevant papers related to tool-calling benchmarks (they were then downloaded and read by the authors, their summaries were handwritten by the authors into the manuscript without help from GenAI automated text generation)].

Conflicts of Interest: The authors were employed by the System Research and Applications, STMicroelectronics, Italy. They declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

A2A	Agent to Agent
AI	Artificial Intelligence
DL	Deep Learning
GBps	Gigabyte per Second
GenAI	Generative Artificial Intelligence
GGUF	GPT-Generated Unified Format
IR	Information Retrieval
LLM	Large Language Model
MCP	Model Context Protocol
P2P	Peer to Peer
RAG	Retrieval-Augmented Generation
SPOF	Single Point of Failure
SPS	Samples Per Second
TinyML	Tiny Machine Learning
TFLOPS	Tera Floating-point Operations Per Second
TOPS	Tera Operations Per Second
TPS	Tokens Per Second
TTFT	Time to First Token
VLM	Vision-Language Model
WDM	Workload Distribution Manager

Appendix A. Communication Protocol Message Formats

This appendix contains a table listing the JSON payloads used by the proposed implementation.

Table A1 lists the message types exchanged between agents on each of the five communication planes introduced in Section 5.2. For each message, the transport protocol, the message type, and the payload shape used by the implementation are reported.

Table A1. Summary of Functional Communication Protocols.

Plane	Command	Shape/Details

Discovery (UDP)	HELLO (UDP)	`{"type":"HELLO",`
		`"id":<agent_id>,`
		`"port":<tcp_port>}`

Election (TCP)	HELLO (TCP)	`{"type":"HELLO",`
		`"from":<id>,`
		`"score":<int>,`
		`"has_leader_preset":<bool>,`
		`"client_preset":<str|null>,`
		`"leader_preset":<str|null>}`

	ELECTION	`{"type":"ELECTION",`
		`"from":<id>,`
		`"score":<int>}`

	LEADER_CLAIM	`{"type":"LEADER_CLAIM",`
		`"from":<id>,`
		`"score":<int>}`

Table A1. *Cont.*

Plane	Command	Shape/Details
Backup (TCP)	CONV_BACKUP	`{"type":"CONV_BACKUP", "from":<id>, "chat_id":<str>, messages":[...], "leader_id":<id>, "ts":<float>}`
	CONV_RESTORE_REQ	`{"type":"CONV_RESTORE_REQ", "from":<id>, "chat_id":<str>"*"}`
	CONV_RESTORE_RESP	`{"type":"CONV_RESTORE_RESP", "from":<id>, "chat_id":<str>, "messages":[...], "ts":<float>}`
Operational (TCP)	fetchskills	`{"text":"fetchskills"}`
	tool-defs reply	`{"text":"<JSON array of tool defs>"}`
	tool invocation	`{"text":"{"name":<fn>, "arguments":{...}}"}`
	tool result reply	`{"text":"<result>"}`
	echo/error replies	`{"text":"echo:  ..."} or errors`
Console (TCP)	console broadcast	`{"text":<prompt input>}`

Rules: Every TCP message carries an auto-stamped "from":Agent-ID field.

Appendix B. Complete Tool List

Table A2 lists the entirety of the tools the LLMs were tested on.

Table A2. The 24 handcrafted tools used for benchmarking, grouped by category, with their input parameters.

Tool	Description	Parameters
Face/identity		
`identify_face`	Match the visible face against enrolled users and return their name.	—
`add_face`	Enroll the currently visible person under a given name.	`name`
`remove_face`	Delete a specific person from the database by name.	`name`
`rename_face`	Change the stored name of an enrolled person.	`old_name`, `new_name`
`list_faces`	List all enrolled people.	—
`delete_all_faces`	Permanently erase all enrolled faces (destructive).	—
`assess_face_attributes`	Estimate age, gender, mood, and expression (no name).	—
Vision/scene		
`describe_scene`	Summarize everything visible in the camera view.	—
`read_text`	Extract text (OCR) from the current view.	—
`detect_gesture`	Identify hand/body gestures (wave, point, nod, etc.).	—
`count_general_entity`	Count instances of an object in the current frame.	`entity`
`estimate_distance`	Estimate distance (m) to a target object.	`target`
`take_photo`	Capture and save a still image.	—

Table A2. *Cont.*

Tool	Description	Parameters
Monitoring		
`await_motion`	Start a background watcher that alerts on any motion.	—
`await_specific_name`	Alert when a named enrolled person appears.	`name`
`await_general_entity`	Alert when a given object/animal type appears.	`entity`
`stop_monitoring`	Cancel all active background watchers.	—
Audio		
`classify_sound`	Identify a non-speech sound (bark, siren, knock, etc.).	—
`measure_noise_level`	Report ambient volume in decibels.	—
`transcribe_speech`	Record audio for *N* seconds and transcribe to text.	`duration_seconds`
Sensors/system		
`check_weather_outside`	Report outdoor weather conditions.	—
`check_air_quality`	Report indoor air quality (CO_2, PM2.5).	—
`get_camera_status`	Report camera health/diagnostics.	—
`list_capabilities`	List all available system actions.	—

Appendix C. Dataset Generation Prompt

The following prompt was used to generate the synthetic queries described in Section 6.1.1:

> "You are an expert in generating synthetic data simulating natural user queries for a smart home agent. There are 24 tools. Write 20 queries for each tool. CONTEXT: An agent reads the user query and chooses the right tool (or decides no tool is needed). When a tool is called, various devices (like cameras, microphones, or sensors) can execute it and reply differently based on what they see. The agent will then translate these device replies back into natural language for the user. QUERIES ARE NEEDED TO BENCHMARK THIS SYSTEM. Keep the queries relatively short and natural. Some of them can be slightly imprecise, just like a real user would say (e.g., "who's at the door down there?" implies using the `identify_face` tool). EVERY QUERY MUST HAVE ONLY ONE GROUND TRUTH. Also include the correct parameters based on the query (if the function requires them).
> Output each query in this format:
> `{`
> `"query": "Change Sarah to Sara.",`
> `"tool": "rename_face",`
> `"parameters": {`
> `"old_name": "Sarah",`
> `"new_name": "Sara"}`
> `}`
> Leave `"parameters": {}` if they are not needed. Output a .json file."

Appendix D. Benchmark C Grading Details

The final replies produced in Benchmark C are graded automatically by an LLM-as-judge (GLM-4.7-Flash). For each record, the judge receives the user query, the function/tool-call that was issued, the data the function returned, and the model's final reply, supplied as a single structured JSON object containing the fields `user`, `tool_call`, `function_reply`, and `model_final_reply`. The judge is queried at temperature 0 to make the grading

as deterministic as possible, and is instructed to return only a JSON object of the form `{"grade": <integer 1-10>}`.

Appendix D.1. Judge System Prompt

The exact system prompt supplied to the judge for every record is reproduced verbatim below:

```
You are an expert evaluator. Given a user query, the function/
tool-call that was made, and the data the function returned,
grade from 1 to 10 how well "model_final_reply" answers the query.
Functions may have been executed across different devices/sources,
and the task varies by query. Judge the reply on: - Relevance:
does it directly address what the user asked? - Faithfulness: is
it grounded in the returned data, with no invented or contradicted
facts? - Use of evidence: does it sensibly weigh/prioritize
the data (e.g., confidences, conflicting sources) rather than
dumping everything indiscriminately? - Clarity: is it a coherent,
useful summary? Scoring guide: 1-3: ignores the query, misreads
the data, or hallucinates. 4-6: partially addresses it; weak
prioritization or notable omissions. 7-8: addresses the query and
is grounded, minor gaps. 9-10: directly and accurately answers,
reasons well over the data, concise. Respond with ONLY valid JSON
and nothing else: {"grade": <integer 1-10>}. Always state the
grade in JSON.
```

Appendix D.2. Grade Extraction and Handling of Unparseable Outputs

The integer grade is recovered from the judge's raw answer through a cascade of increasingly permissive parsing strategies, applied in order. First, the entire answer is parsed as JSON and accepted if it contains an integer `grade` field within the valid 1 to 10 range. If that fails, any fenced code block (delimited by triple backticks, optionally tagged `JSON`) is parsed in the same way. If that also fails, any brace-delimited object literal containing the key `grade` is parsed. Finally, a loose regular-expression match of the form `grade: N` is attempted, accepting the value only if it falls within range. A grade is recorded only when one of these strategies yields an integer in $[1, 10]$. Records for which no valid grade can be extracted are saved for manual annotation. Each such case is assigned a placeholder identifier and written to a separate pending-annotation file together with the complete datapoint and the judge's raw answer, so that it can be resolved by manual annotation afterward. In addition, the raw judge output is logged for every record, whether or not a grade was successfully parsed, to allow the grading process to be audited and reproduced. This ensures that the reported grades are fully traceable and that no record is silently dropped from the evaluation. All the data and the code are available in the aforementioned project GitHub repository.

Appendix E. Validation of GLM-4.7-Flash as a Judge

Benchmark C relies on GLM-4.7-Flash as an automatic judge. This appendix assesses whether that judge is a reliable grader of summarization quality, focusing on whether it produces a reliable relative ranking of systems rather than absolute scores. To this end, news-article summarization is evaluated as a proxy task, as it is related, though not identical, to the summarization of tool results graded in Benchmark C. The data were gathered from the SummEval study by Fabbri et al. [42], in which three expert human annotators scored news-article summaries. Both the SummEval evaluation [42] and the

Benchmark C evaluation performed in this work assigned scores by pondering four quality metrics, though the specific metrics differ slightly between the two. The GLM-4.7-Flash grades are then compared against the SummEval human expert annotations to verify whether they follow a similar trend. The grading and analysis code is available in the project repository.

Appendix E.1. Dataset

The data come from the SummEval study [42]. SummEval starts from 100 news articles sampled from the CNN/DailyMail test set. For each article, 16 summarization systems produced one summary, giving $N = 100 \times 16 = 1600$ summaries. Each summary was annotated by 3 expert annotators, who scored it on four quality dimensions, coherence, consistency, fluency, and relevance, each on an integer Likert scale from 1 to 5. These expert annotations are used as the human reference. The annotations and the decoded summaries are read from the released file model_annotations.aligned.jsonl (https://storage.googleapis.com/sfr-summarization-repo-research/model_annotations.aligned.jsonl (accessed on 16 July 2026)), obtained from the public SummEval distribution hosted by the benchmark authors.

Appendix E.2. Judging Protocol

Each of the 1600 summaries was graded once by GLM-4.7-Flash. For every summary the model received, the system prompt reproduced in Appendix E.5, the source article, and the summary. The system prompt states the four SummEval dimension definitions and instructs the model to weigh all four dimensions and return a single overall grade as an integer from 1 to 5. The 1 to 5 scale is used here to match the SummEval Likert scale. For each summary, the model produced a JSON object containing a short justification followed by the grade, so that the grade followed from the stated reasoning. Decoding was deterministic (temperature 0). The grade for summary i is denoted $g_i \in \{1, \ldots, 5\}$ and is compared against the overall score assigned by the experts.

Appendix E.3. Reference Scores and Comparison Metrics

Let $e_{i,d}^{(k)}$ be the score that expert $k \in \{1, 2, 3\}$ assigns to summary i on dimension $d \in \mathcal{D} = \{\text{coh}, \text{con}, \text{flu}, \text{rel}\}$. A single overall score per expert is the mean of that expert's four dimension scores, given by Equation (A1).

$$o_i^{(k)} = \frac{1}{|\mathcal{D}|} \sum_{d \in \mathcal{D}} e_{i,d}^{(k)}, \qquad |\mathcal{D}| = 4. \tag{A1}$$

Averaging the three expert overall scores gives a single averaged-expert reference, given by Equation (A2). This average is used to calculate the metric defined in Equation (A7) and to order the 1600 summaries from highest-graded ones to the lowest, as reported in the bottom panel of Figure A1.

$$\bar{o}_i = \frac{1}{3} \sum_{k=1}^{3} o_i^{(k)}. \tag{A2}$$

Given two score vectors u and v of length N, each being either the judge grades g or an expert overall score, their agreement is measured with two rank correlations. This is because the grades are ordinal and the claim under test is that the judge orders summaries as the experts do, not that it reproduces their absolute scores. The first method is the tie-corrected Kendall coefficient of Equation (A3). Let C be the number of concordant pairs,

D the number of discordant pairs, and T_u and T_v the number of pairs tied only in u and only in v; then

$$\tau_b(u,v) = \frac{C-D}{\sqrt{(C+D+T_u)\,(C+D+T_v)}} \in [-1,1]. \tag{A3}$$

The tie correction is required because the grades are discrete and contain many ties. The second coefficient is the Spearman correlation of Equation (A4), the Pearson correlation of the ranks,

$$\rho(u,v) = \mathrm{Pearson}\big(R(u), R(v)\big), \tag{A4}$$

where $R(\cdot)$ replaces each entry by its rank, tied entries receiving their average rank. The inter-expert agreement, which bounds the agreement any judge can be expected to reach, is the mean pairwise correlation between the three experts' overall scores, given by Equation (A5). With $\mathcal{P} = \{(1,2),(1,3),(2,3)\}$,

$$\tau_{EE} = \frac{1}{3}\sum_{(k,l)\in\mathcal{P}} \tau_b\big(o^{(k)}, o^{(l)}\big), \qquad \rho_{EE} = \frac{1}{3}\sum_{(k,l)\in\mathcal{P}} \rho\big(o^{(k)}, o^{(l)}\big). \tag{A5}$$

The judge-versus-expert agreement is measured against each expert separately and then averaged over the three experts, given by Equation (A6). This is the primary comparison, because it mirrors the construction of the inter-expert agreement above: both average correlations between single graders.

$$\tau_{LE} = \frac{1}{3}\sum_{k=1}^{3} \tau_b\big(g, o^{(k)}\big), \qquad \rho_{LE} = \frac{1}{3}\sum_{k=1}^{3} \rho\big(g, o^{(k)}\big). \tag{A6}$$

As a variant, the judge is also correlated against the single averaged-expert reference $\bar{o}$ of Equation (A2), given by Equation (A7).

$$\tau_{L\bar{E}} = \tau_b(g,\bar{o}), \qquad \rho_{L\bar{E}} = \rho(g,\bar{o}). \tag{A7}$$

All correlations are computed once, over the full set of $N = 1600$ summaries.

Appendix E.4. Results

Table A3 reports the Kendall (A3) and Spearman (A4) values for the primary comparison, the judge against the experts (A6) and the experts against each other (A5). Figure A1 shows the calibration and ranking views of the judge grades against the averaged-expert reference.

The judge-versus-expert correlations are positive ($\tau_{LE} = 0.37$, $\rho_{LE} = 0.45$) and lower than the inter-expert agreement ($\tau_{EE} = 0.58$, $\rho_{EE} = 0.70$), as reported in Table A3. Relative to the inter-expert reference, the judge reaches about 0.64 of the Kendall value and about 0.65 of the Spearman value. With only three experts, the inter-expert reference is itself estimated from three pairwise values and should be read as approximate. Correlating the judge against the single averaged-expert reference of Equation (A7) instead yields slightly higher values ($\tau_{L\bar{E}} = 0.40$, $\rho_{L\bar{E}} = 0.50$), as expected from averaging the three experts into one less noisy reference; the per-expert average of Equation (A6) remains the directly comparable figure. The calibration panel of Figure A1 shows that the mean judge grade increases monotonically across the expert-score bins, while remaining flatter than the diagonal, indicating that the judge grades the lowest-scored summaries higher and the highest-scored summaries lower than the experts, compressing its grades toward the center of the scale. The ranking panel of Figure A1 shows the same ordering behavior as the rolling mean of the judge grade decreases together with the averaged-expert reference when summaries are sorted from best to worst. These results indicate positive rank agreement

between the judge and the experts, below the inter-expert level. Benchmark C uses these grades only for relative comparison across model configurations and reply counts on a fixed grid, rather than as absolute quality scores.

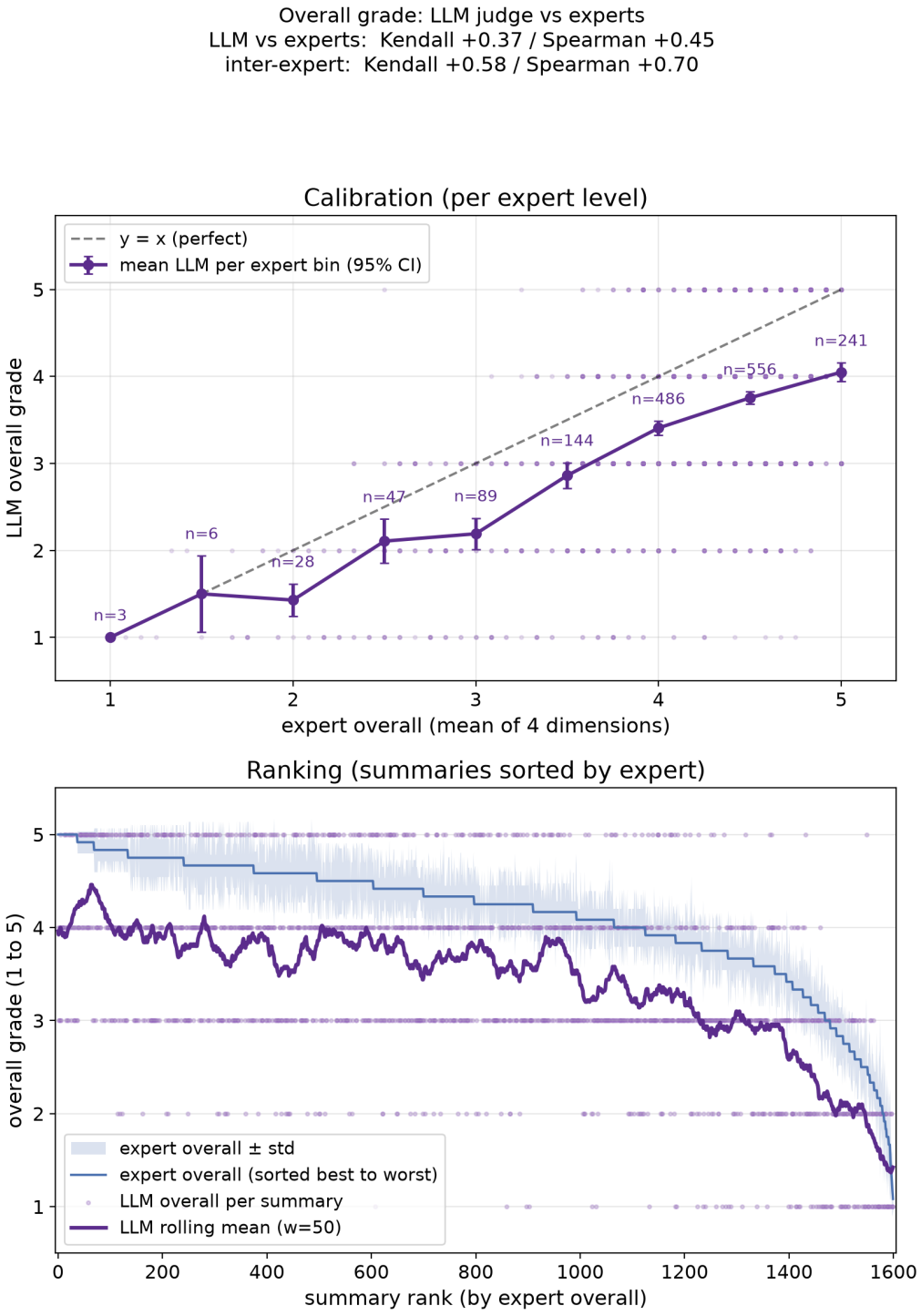


Figure A1. Overall grade of GLM-4.7-Flash against the expert reference $\bar{o}$ of Equation (A2). Top: calibration, the mean judge grade for summaries grouped into half-point bins of the averaged-expert score, with 95% confidence intervals and bin sizes; the dashed line is perfect agreement. Bottom: ranking view, the summaries ordered from best to worst by the averaged-expert score, with the averaged-expert reference and the rolling mean of the judge grade.

Table A3. Rank correlation of the overall grade, over all 1600 summaries.

Comparison	Kendall τ_b	Spearman ρ
Judge vs. experts (τ_{LE}, ρ_{LE})	0.37	0.45
Inter-expert (τ_{EE}, ρ_{EE})	0.58	0.70

Appendix E.5. Judge System Prompt

The system prompt supplied to GLM-4.7-Flash for every summary is reproduced verbatim below:

```
You are an expert evaluator of news-article summaries. Given a
news article and a summary of it, grade from 1 to 5 how good the
summary is OVERALL (1 = worst, 5 = best). Judge the summary on
these four dimensions, then combine them into a single overall
grade: - Coherence: the collective quality of all sentences. The
summary should be well-structured and well-organized; rather than
a heap of related information, the sentences should connect and
flow in a logical order, building from sentence to sentence into a
single coherent body of information about a topic. - Consistency:
the factual alignment between the summary and the source. A
factually consistent summary contains only statements entailed
by the source document. This measures whether everything stated
is faithful to and supported by it, with no invented or distorted
information. Penalize summaries that contain hallucinated facts.
- Fluency: the quality of individual sentences. Sentences should
be clean, grammatical, and well-formed, reading smoothly on
their own, with no formatting problems, capitalization errors,
or obviously ungrammatical sentences (e.g., fragments, missing
components) that make the text difficult to read. - Relevance:
selection of important content from the source. The summary
should capture the source's important information and only that,
well-focused, including nothing unimportant. Penalize summaries
that contain redundancies and excess information. Scoring guide
(overall, 1 to 5): 1: fails on most dimensions; incoherent,
inconsistent, has lots of issues. 2: weak; problems on one or
more dimensions, not good, has major issues. 3: acceptable;
captures the gist but with notable gaps or errors; has issues.
4: good; faithful, relevant, and readable, only has minor
issues. 5: excellent; coherent, faithful, fluent and complete,
no issues spotted. First, write a short reason that weighs the
four dimensions, THEN give the overall grade. The grade must
follow from and agree with your reasoning. Respond with ONLY valid
JSON and nothing else: no markdown code fences, no comments, no
text before or after. Always state the grade in JSON. You are
evaluating a summary of a news article. Give one overall quality
grade from 1 to 5.
```

Appendix F. LLM Handlers

A "handler" is the adapter that translates between the application and one model family: it formats the request the model expects and parses the reply. The two families studied here are FunctionGemma and Qwen3. Both handlers let a small local language

model invoke external "tools" (functions) by emitting a structured fragment of text that the handler must read back into a function name and a set of arguments. The central practical difference is that Qwen3 emits arguments as standard JSON, which a general-purpose reader decodes directly, whereas FunctionGemma uses a non-standard encoding that must be scanned and interpreted with a custom algorithm. Despite these differences, both parsers are designed to return results in the same downstream format. Other details about the handlers used are visible in Table A4.

Table A4. Comparison of the two implemented handlers, focusing on the output parsers, relevant for tool-calling measurements.

Aspect	FunctionGemma Handler	Qwen3 Handler
Text format the model produces		
Call delimiters	Custom opening and closing tags surrounding the call	A single tagged block whose contents are standard JSON
Argument encoding	Bespoke `key:value` pairs; text values are individually marked with escape tokens	A standard JSON object
Generation stop signal	Halts after the first complete call	Halts at the end of the turn, allowing several calls at once
Main parsing strategy		
Approach	Hand-written scan over the characters, locating the call and its argument block	A pattern match extracts the block, which is then decoded by a standard JSON reader
Splitting the arguments	Custom logic tracks nesting depth and escape markers to separate one argument from the next	Not required—the JSON reader returns the arguments directly
Determining value types	Inferred by hand (text, true/false, whole number, or decimal)	Obtained automatically from the JSON reader
Recovery from malformed or truncated output		
Output cut off mid-call	Recovers the call by locating its final closing marker	Recovers the trailing call by matching its braces, aware of quoted text
Fragment with no recoverable end	Discarded	Decoding fails, a recovery attempt is made, and the fragment is discarded if it still fails
Output missing its opening tag	Recovered (guards against a known repetition failure of the smallest model)	Not recovered—the opening tag is required
Arguments wrapped as text	Not applicable	Validated, and otherwise stored under a default field
Call with no function name	Discarded	Discarded
Result handed to the application		
Final form	Identical: a function name paired with its arguments, wrapped in the same response structure	

Note. Both parsers prioritize safe failure: any fragment that cannot be read with confidence is dropped rather than passed on in a corrupted state.

Appendix G. Verification of the FNB-C2 Power Meter

This appendix reports the verification of the FNIRSI FNB-C2 power meter used for the measurements of Section 7. Its voltage and current readings were cross-checked against a Keysight B2902B precision source/measure unit, treated throughout as the reference (ground truth). The FNB-C2 sits in line between the bench power supply and the load,

$$\text{power supply} \longrightarrow \text{FNB-C2} \longrightarrow \text{cable} \longrightarrow \text{Keysight B2902} \longrightarrow \text{resistive load}$$

while the Keysight measures at the load with one channel across it (voltage) and one in series (current). For every configuration, both instruments were logged simultaneously at 5 SPS, and their readings were averaged over a 10 s acquisition window.

Appendix G.1. Voltage and Current Are Verified Differently

The FNB-C2 measures upstream of the connecting cable, whereas the Keysight measures directly at the load. The cable has a small but finite resistance R_{cable}, so under load a current I produces an ohmic drop $I\,R_{\text{cable}}$ between the two voltage-sensing points. A voltage comparison taken under load would therefore be dominated by this cable drop rather than by any difference between the two meters, and would misattribute a wiring effect to the instrument. The current, by contrast, is common to the whole series path and is theoretically identical at both meters regardless of the cable. For this reason, the voltage accuracy is verified with no load connected: with $I = 0$, the cable drop vanishes, and the two instruments sense the same potential, so their difference reflects only meter accuracy. The current accuracy is verified under load, where the shared series current makes the two readings directly comparable. The cable resistance itself is then quantified afterward (Table A6), confirming that the choice above was justified.

Appendix G.2. Voltage (No Load)

With no load connected, the FNB-C2 read 5.0572 V against the Keysight reference of 5.0607 V: a difference of 3.5 mV, or 0.07% of reading.

Appendix G.3. Current (Under Load)

Six resistive configurations (loads A–F), built from nominally 12 Ω, 5 W resistors (loose tolerance, as their precision is not critical for this verification) combined in series and parallel, sweep the current from about 0.12 A to 2.84 A, spanning and exceeding the range drawn by the tested boards. Table A5 reports, for each load, the reference current I_{ref}, the FNB-C2 current I_{FNB}, and their difference $\Delta I = I_{\text{FNB}} - I_{\text{ref}}$.

Table A5. Current verification of the FNB-C2 against the Keysight B2902B reference. Each value is the mean over a 10 s window. Differences and percentages are computed from the unrounded means; the displayed per-reading values are rounded, so recomputing from the printed figures may differ in the last digit.

Load	I_{ref} [A]	I_{FNB} [A]	ΔI [mA]	$\Delta I / I_{\text{ref}}$ [%]
A	0.11654	0.11738	+0.84	+0.72
B	0.4043	0.4069	+2.56	+0.63
C	0.7737	0.7787	+4.97	+0.64
D	1.4286	1.4360	+7.39	+0.52
E	2.1735	2.1856	+12.12	+0.56
F	2.8193	2.8342	+14.87	+0.53

The FNB-C2 reads slightly high at every load, by a nearly constant fraction that stays below 0.75% of the reading across the full sweep (equivalently, within 15 mA in absolute terms). The deviation is systematic rather than random: it is always positive and far exceeds the FNB-C2 per-window standard deviation, which is at most 2.04 mA (and below 1.00 mA for every load except the heaviest).

Appendix G.4. Reconciling with the Datasheet Specification

The manufacturer specifies the FNB-C2 to $\pm(0.2‰ + 2$ digits$)$ for voltage and $\pm(0.5‰ + 2$ digits$)$ for current, i.e., about ±0.02% and ±0.05% of reading. The deviations measured here are larger: roughly 0.07% for voltage and 0.5–0.75% for current. This has three reasons. First, these figures are the instrument's own best-case specification under reference conditions, whereas the values above come from comparing two independent instruments on a real bench. Such a comparison reflects the combined uncertainty of both

meters and can therefore exceed either datasheet figure alone, even if the Keysight, as a calibrated laboratory SMU, is expected to contribute far less of that budget than the FNB-C2. Second, such specifications for low-cost USB meters can be optimistic, which is precisely why this external cross-check is worthwhile. Third, and most important for this work, the current deviation is a small, systematic, repeatable positive gain bias below 0.75%. Because it is systematic, it shifts every power figure in Section 7 by less than 1%, negligible next to the differences between configurations, and because every measurement uses the same FNB-C2, it largely cancels in the same-meter ratios (such as the energy-efficiency ratios of Section 7).

Appendix G.5. Quantifying the Cable Resistance

Having attributed the loaded voltage difference to the connecting cable, it is possible to verify that attribution directly. Under load, the FNB-C2 and Keysight voltages differ by tens to hundreds of millivolts; dividing this difference by the reference current, $R_{\text{cable}} = (V_{\text{FNB}} - V_{\text{ref}})/I_{\text{ref}}$, should return the cable resistance and, if the interpretation is correct, be roughly the same at every load. Table A6 confirms this: over a more than twentyfold change in current, the ratio stays in a narrow band around 0.25 Ω (0.21–0.28 Ω), as expected for a fixed series resistance. This is exactly the wiring drop that the no-load voltage check was designed to exclude, and it confirms that the loaded voltage difference is not a meter error.

Table A6. Cable resistance estimated from the loaded voltage difference between the FNB-C2 (upstream of the cable) and the Keysight reference (at the load), $R_{\text{cable}} = (V_{\text{FNB}} - V_{\text{ref}})/I_{\text{ref}}$. Each value is the mean over a 10 s window. ΔV and R_{cable} are computed from the unrounded means; the displayed values are rounded, so recomputing from the printed figures may differ in the last digit.

Load	$\Delta V = V_{\text{FNB}} - V_{\text{ref}}$ [mV]	I_{ref} [A]	R_{cable} [Ω]
A	24.27	0.11654	0.208
B	95.5	0.4043	0.236
C	205.8	0.7737	0.266
D	397.8	1.4286	0.278
E	607.4	2.1735	0.279
F	712.0	2.8193	0.253

Appendix G.6. Summary

At no load, the FNB-C2 voltage matches the reference to 3.5 mV (0.07%), and under load, its current matches to better than 0.75% of the reading with a small, repeatable positive bias. A larger voltage difference in voltage measurements under load is accounted for by a cable resistance of about 0.25 Ω (Table A6). Since this bias is systematic and common to every measurement taken with the same instrument, the FNB-C2 readings used in Section 7 are reliable at the accuracy required by this work.

Appendix H. The Reason for Unexpected Disconnections

During early testing, some disconnections were reported between the agents. Their cause was traced to the HailoRT Python bindings (`pyhailort`, HailoRT 5.3): the library does not release the CPython Global Interpreter Lock (GIL) while its GenAI calls are in flight. Since the liveness heartbeats of the communication layer are sent by a Python thread of the same process, no heartbeat can be emitted while the GIL is held. After 15 s without heartbeats, the peers consider the device dead and trigger a leader re-election, even though the device is healthy and simply busy on the accelerator.

The behavior was verified with a simple probe. A background thread increments a pure-Python counter as fast as possible, while a second thread emits a heartbeat every

2 s; both run for the whole experiment. A 10 s idle baseline defines the counter's normal rate (GIL fully available). Each HailoRT call is then executed on the main thread and, for each phase, the counter rate relative to the baseline and the number of heartbeats actually emitted versus expected are recorded. If a phase holds the GIL, both collapse to zero. The results are reported in Table A7: model loading, token generation, and model release all hold the GIL for their entire duration, while the two idle control phases behave normally. In this run, the model was still cached in the Hailo on-module memory from a previous execution, so loading took only about 7 s; on a cold start, loading the same Qwen3:1.7b `.hef` was measured to take up to about 31 s, during which no heartbeat is emitted, comfortably exceeding the 15 s liveness threshold.

In practice, this bug rarely disrupts operation. In the proposed pipeline, each query is split into two short generations (tool call, then reply from tool results), so individual GIL-held windows almost always remain below the 15 s threshold, especially after the first load. When a false-positive disconnection does occur, the leader election logic recovers automatically: as soon as the blocking call returns, the node resumes sending heartbeats and rejoins the network, and any node that claimed leadership in the meantime steps down in its favor. The issue can be eliminated entirely by running the HailoRT inference in a separate process, which owns its own GIL, making the accelerator behave like an external inference server (as `llama.cpp` does). An equivalent alternative is `hailo-ollama`, Hailo's Ollama-compatible endpoint; however, Qwen3:1.7b is not yet supported by `hailo-ollama` in the HailoRT version used here (5.3), so the in-process bindings were retained for this work.

Table A7. GIL availability probe on the Raspberry Pi 5 + Hailo-10H node (HailoRT 5.3, Qwen3:1.7b, 20 generated tokens). Rate is the pure-Python counter throughput relative to the idle baseline; heartbeats are 2 s periodic calls emitted by a concurrent thread (actual/expected).

Phase	Duration [s]	Rate [% Baseline]	Heartbeats	GIL
Idle baseline	10.01	100.2	4/5	released
VDevice creation	0.06	0.0	0/0	held
Model load (cached)	7.30	0.0	1/4	held
Idle (control)	1.01	98.8	0/0	released
Generation (20 tok.)	6.20	1.4	2/3	held
Model release	2.86	0.3	1/1	held

Appendix I. Robustness of the Benchmark to the Query-Generating LLM

All test queries of Section 6 were phrased by a single model (Claude Opus 4.7), raising the concern that the reported accuracies reflect that model's phrasing rather than the benchmarked models. To test this, the identical test skeleton was regenerated through two further generators, ChatGPT and GLM5.2, so that the ground-truth tool, the parameters, the abstention cases, the fixed-seed tool pools, and the simulated tool replies are held constant, and only the surface phrasing of each query differs. Tool-name mismatches across generators are 0 of 500, and the only differences are 154 free-parameter values (such as a sampled name), so every result below is attributable to phrasing alone.

Appendix I.1. Query-Set Diversity

The ablation is only meaningful if the three generators actually phrase queries differently, so that any downstream invariance is a property of the benchmark rather than of near-identical inputs. Figure A2 shows they differ on every surface metric, and the pairwise unigram vocabulary overlap (Jaccard) is only 0.44 to 0.47.

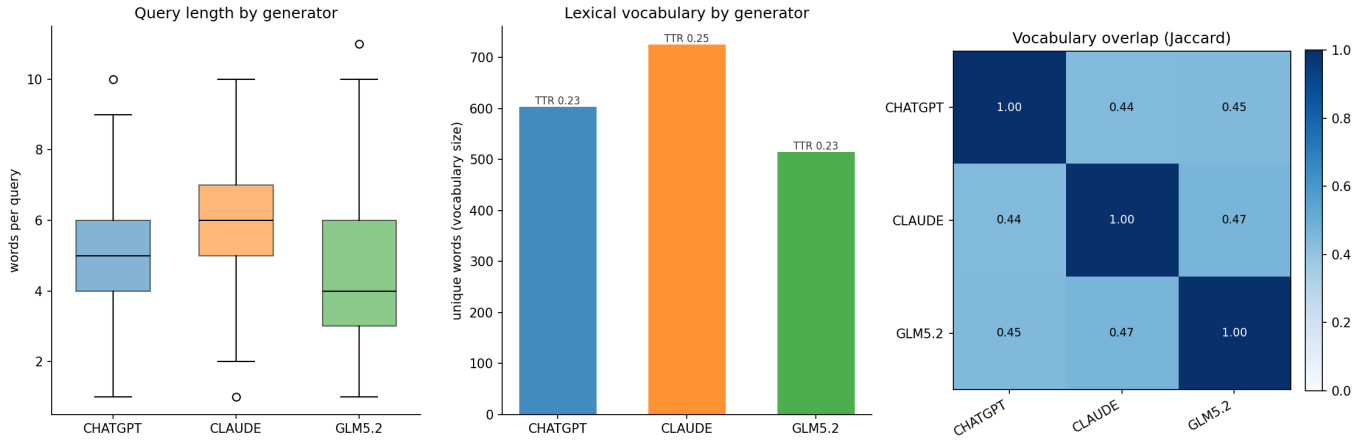


Figure A2. Surface diversity of the three query sets (500 queries each): mean query length, vocabulary size, type-token ratio (TTR), and distinct-2, per generator. The pairwise unigram Jaccard vocabulary overlap is only 0.44 to 0.47 (ChatGPT–Claude 0.44, ChatGPT–GLM5.2 0.45, Claude–GLM5.2 0.47).

Appendix I.2. Accuracy Is Invariant to the Generator (Parts A and B)

Across every (benchmarked model, pool size) cell the mean accuracy range between the three generators is 2.5 percentage points in Part A (worst case 5.8) and 3.1 in Part B (worst case 7.5), and the benchmarked-model ranking is preserved: the mean Spearman ρ between generator pairs is 0.89 to 1.00 in Part A and 0.77 to 0.90 in Part B (Part A ranks three benchmarked models, so only six orderings are possible; Part B ranks four entries including the reranker, giving twenty-four possible orderings. These correlations should be read as indicative of rank stability rather than as a precise estimate.), with the reranker included as a ranked entry in Part B. A paired item-level McNemar test (exact binomial), matched by `sentence_id`, finds a significant phrasing-dependent flip in only 1 of the 54 (benchmarked model, pool size, generator-pair) tests of Part A and 1 of the 48 of Part B, at $\alpha = 0.05$ uncorrected, fewer than the $0.05 \times 54 = 2.7$ and $0.05 \times 48 = 2.4$ false positives expected by chance in Parts A and B respectively. The item-level flip rate is 32.5% for FunctionGemma-270M in Part A and 30.2% in Part B, against 8.7% to 15.3% for the two Qwen3 variants across both parts. Of FunctionGemma-270M's 30 (pool size, generator-pair) cells in Parts A and B combined, the McNemar test is significant in one: pool size 20, ChatGPT versus Claude, in Part A (104 versus 75 flips, $p = 0.036$). Figure A3 shows the Part A spread; Part B behaves the same way.

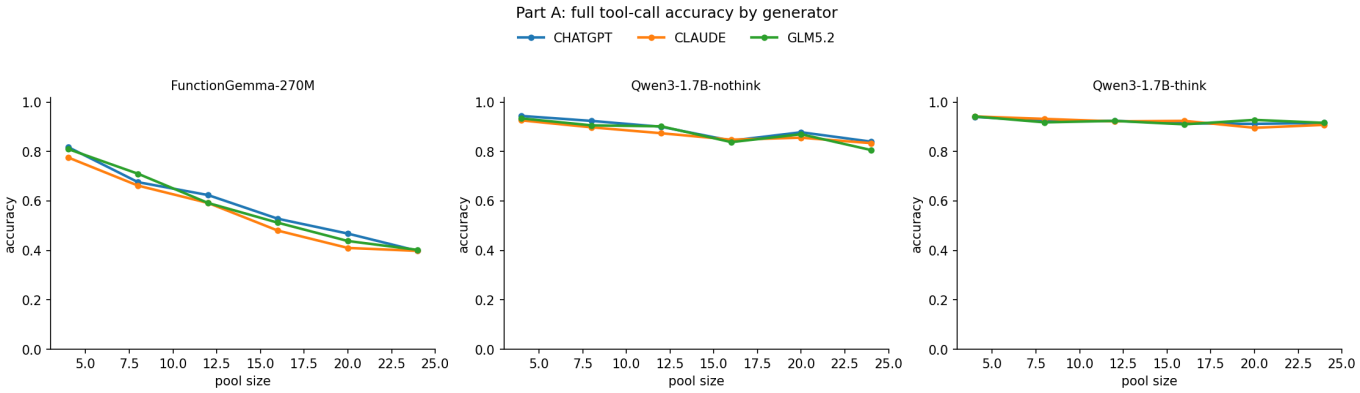


Figure A3. Part A accuracy across the three generators, per benchmarked model and pool size.

Appendix I.3. Summarization Grades (Part C)

Part C is the one place a phrasing effect is measurable: a paired Wilcoxon signed-rank test on the judge grade, matched by `record_index`, is significant in 12 of 45 (model, N replies, generator-pair) cells. The effect is small, on the order of 0.1 to 0.3 points on the 1 to 10 grade scale (for example, 7.99 against 8.31), and stays far below the model-to-model differences the benchmark reports. GLM5.2 is on the higher-scoring side in 11 of the 12 significant cells (8 against ChatGPT, 3 against Claude). As the judge itself is a GLM-family (`glm-4.7-flash`), this one-sided pattern may partly reflect in-family judge preference rather than query phrasing alone. Figure A4 plots the mean grade against the number of aggregated replies for each generator.

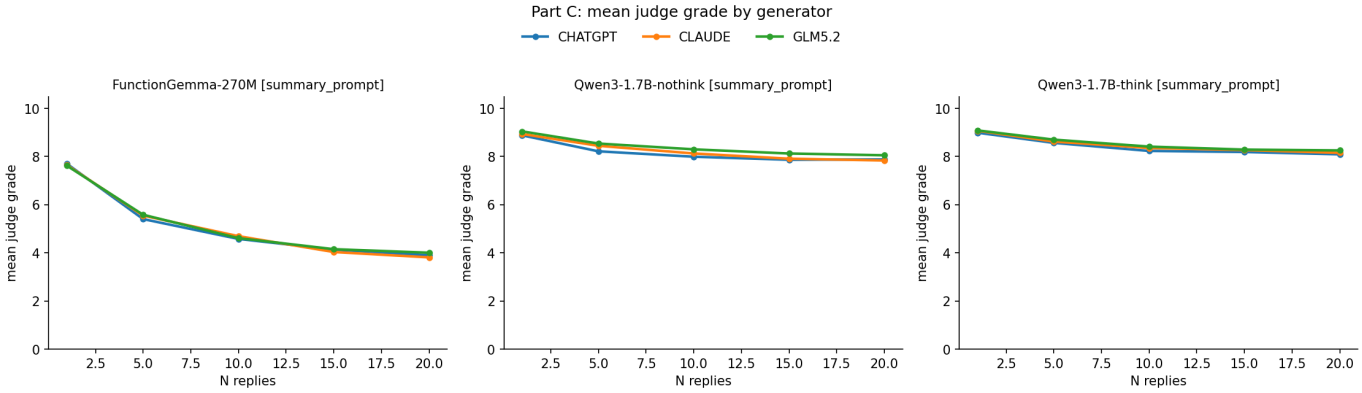


Figure A4. Part C mean judge grade versus number of aggregated replies, per generator.

Appendix I.4. Variance Decomposition

To place the phrasing effect on the same scale as the signal the benchmark measures, the variance in per-cell accuracy or grade is decomposed with an ANOVA effect-size (η^2) computation, run separately for each part. Each part is a complete balanced factorial over the benchmarked model, the query generator, and pool size (the number of replies in Part C), so the reported η^2 values do not depend on the sum-of-squares convention. Table A8 shows that the query generator explains 0.1 to 0.2% of the variance in every part, against 76 to 78% for the benchmarked model; restricting Part C to the Qwen-only cells raises $\eta^2(\text{generator})$ only to 4.3%.

Table A8. Variance decomposition (η^2, ANOVA), run separately per part. The third term is pool size for Parts A and B and the number of replies for Part C.

Part	Model	Generator	Size/Replies	Residual
A ($n = 54$)	77.2%	0.1%	12.1%	10.6%
B ($n = 48$)	78.4%	0.2%	8.1%	13.3%
C ($n = 45$)	76.5%	0.1%	15.7%	7.7%

The ablation therefore indicates that the tool-calling results of Section 6 are not an artifact of the particular LLM used to phrase the queries. It isolates phrasing bias specifically and does not substitute for real end-user dialogue, which remains a limitation discussed in Section 9.

Appendix I.5. Sensitivity to Noisy Queries

The three generators above all produce clean, grammatical English. To probe robustness to the malformed input typical of real chat messages, a NOISY variant of the Claude query set was derived by a deterministic, seeded, rule-based perturbation of the query text only, keyed by `seed + item_index` for reproducibility. The ground-truth tool, parameters, and simulated replies are copied through byte-identical (0 of 500 mismatches), so the answer key and the grading standard are unchanged, and any accuracy change is attributable to the input noise alone.

Each query passes through four independent perturbation stages:

- Abbreviation substitution: 15 fixed patterns (`you're` → `ur`, `you` → `u`, `your` → `ur`, `please` → `pls`, `thank you` → `thx`, `thanks` → `thx`, `okay` → `ok`, `want to` → `wanna`, `going to` → `gonna`, `right now` → `rn`, `because` → `bc`, `don't` → `dont`, `can't` → `cant`, `won't` → `wont`, `what is` → `whats`), each applied with probability 0.5 at its first match.
- Function-word dropping: the 25 tokens `a`, `an`, `the`, `is`, `are`, `was`, `were`, `do`, `does`, `did`, `please`, `can`, `could`, `would`, `just`, `so`, `if`, `some`, `any`, `be`, `then`, `right`, `now`, `and`, `or` are dropped with probability 0.1 per token. Prepositions (`to`, `of`, `in`, `on`, `at`, `for`, `with`), directional particles (`up`, `down`, `out`, `off`) and pronouns (`you`, `your`, `my`, `me`, `i`, `it`, `this`, `that`, `there`, `here`) are excluded from this list, so dropping never strands the sentence's referent or breaks a phrasal verb such as `make out` or `turn off`.
- Character-level typos: each remaining word longer than three characters receives, with probability 0.2, a single interior edit, either a QWERTY-adjacent substitution, an adjacent-character transposition, or a character duplication.
- Case folding: the whole string is lowercased with probability 0.8, then trailing punctuation is stripped.

Ground-truth entity values, taken from the item's parameter values, are exempt from the dropping and typo stages, so noise never corrupts a token the grader matches against.

Applied across the 500 queries, the stages took effect on 8% (abbreviation), 11% (dropping), 49% (typos), and 75% (lowercasing) of items, and every query changed in at least one way. The result preserves content while altering surface form: mean word-token retention against the clean query is 85.6% and mean character-level similarity is 0.92, and the mean query length is essentially unchanged (5.7 versus 5.9 words). Because typos create new word forms, the noisy set has a larger vocabulary (941 versus 725 unique words) and a higher type-token ratio (0.33 versus 0.25); 77% of its queries are fully lowercase, against none in the clean set, and its unigram vocabulary overlap with the clean set is 0.65 (Jaccard).

Accuracy drops under noise in every cell, but by a small margin and without changing the benchmarked-model ranking. In Part A, the mean drop is 5.7 percentage points across the 18 (benchmarked model, pool size) cells, the largest single-cell drop is 10.4 percentage points, the ranking is preserved (Spearman $\rho = 1.00$; see the small-sample caveat in Appendix I.2), and a paired McNemar test is significant in 17 of the 18 cells. In Part B, the mean drop is 4.6 points across 16 cells, $\rho = 0.79$, and McNemar is significant in 10 of 16 cells. In Part C, the mean judge-grade drop is 0.09 on the 1 to 10 scale, with a Wilcoxon test significant in 4 of 15 cells. A variance decomposition attributes 2.3% (Part A) and 2.5% (Part B) of the total variance to the clean/noisy condition, against 78% to 80% for the benchmarked model. Figure A5 shows the Part A gap.

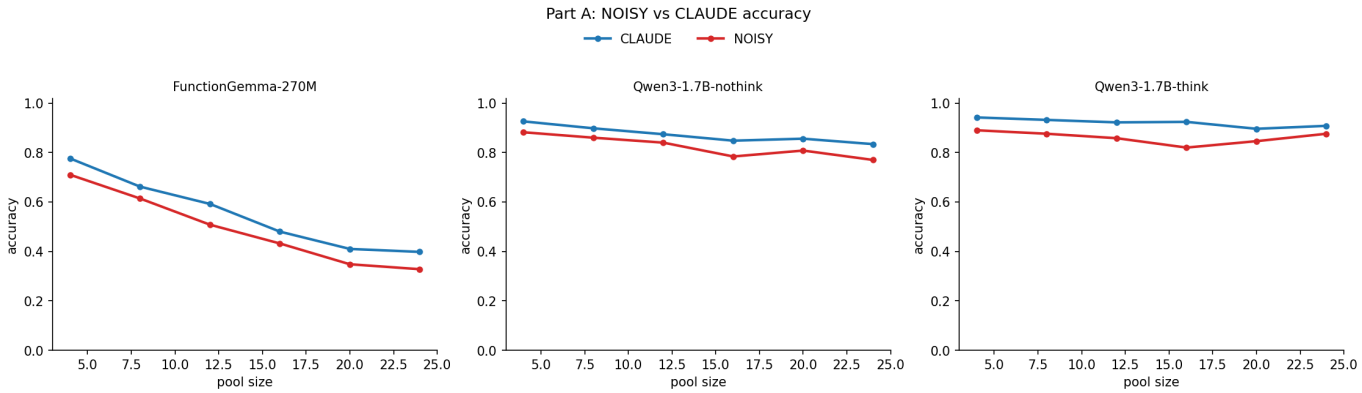


Figure A5. Part A accuracy on the clean Claude queries versus their noisy variant, per benchmarked model and pool size. The noisy line sits consistently below the clean one, by a small and roughly constant margin.

The perturbation protects ground-truth entity values such as names from corruption by construction (since a wrong name can be called correctly, but it will not match its ground truth). This was done to measure robustness to noisy framing rather than measuring how often noise landed on the entity values themselves. However, this is acknowledged as a scope limitation.

Appendix J. Complete Flowcharts of the Architecture

This appendix collects the complete flowcharts of the system behavior outlined in Section 5. Together they trace a device from power-on, through peer discovery, leader election and failover, to the answering of a user query. The diagrams share a common notation. Ovals are terminators (the entry and exit points of a flowchart); rectangles are processing steps; diamonds are decisions; and parallelograms are network input/output steps (a message sent to, or received from, another agent). Two conventions are specific to these diagrams. First, a colored call box whose text reads "(see … diagram)" transfers control to the flowchart of the same color: it is a jump to the referenced diagram rather than an ordinary step. Second, the green box with a dashed border denotes the leader's merged tool list: the data structure that stores the tools exposed by the sensing agents; steps that write it ("Merge into tool list", "Remove tool from tool list") or read it (the dashed "reads" arrow) operate on that structure.

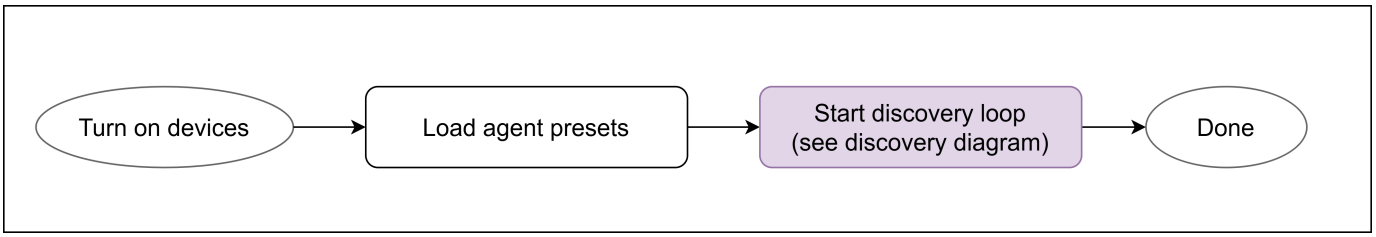


Figure A6. Device startup sequence. When a device is powered on, it first loads its agent presets from the local configuration (the presets decide which agents, sensing, and/or leader the device runs). It then enters the peer discovery loop, drawn as a colored call box because control passes to the discovery flowchart of Figure A7. Discovery runs continuously for the lifetime of the device, so startup itself terminates once the loop has been launched.

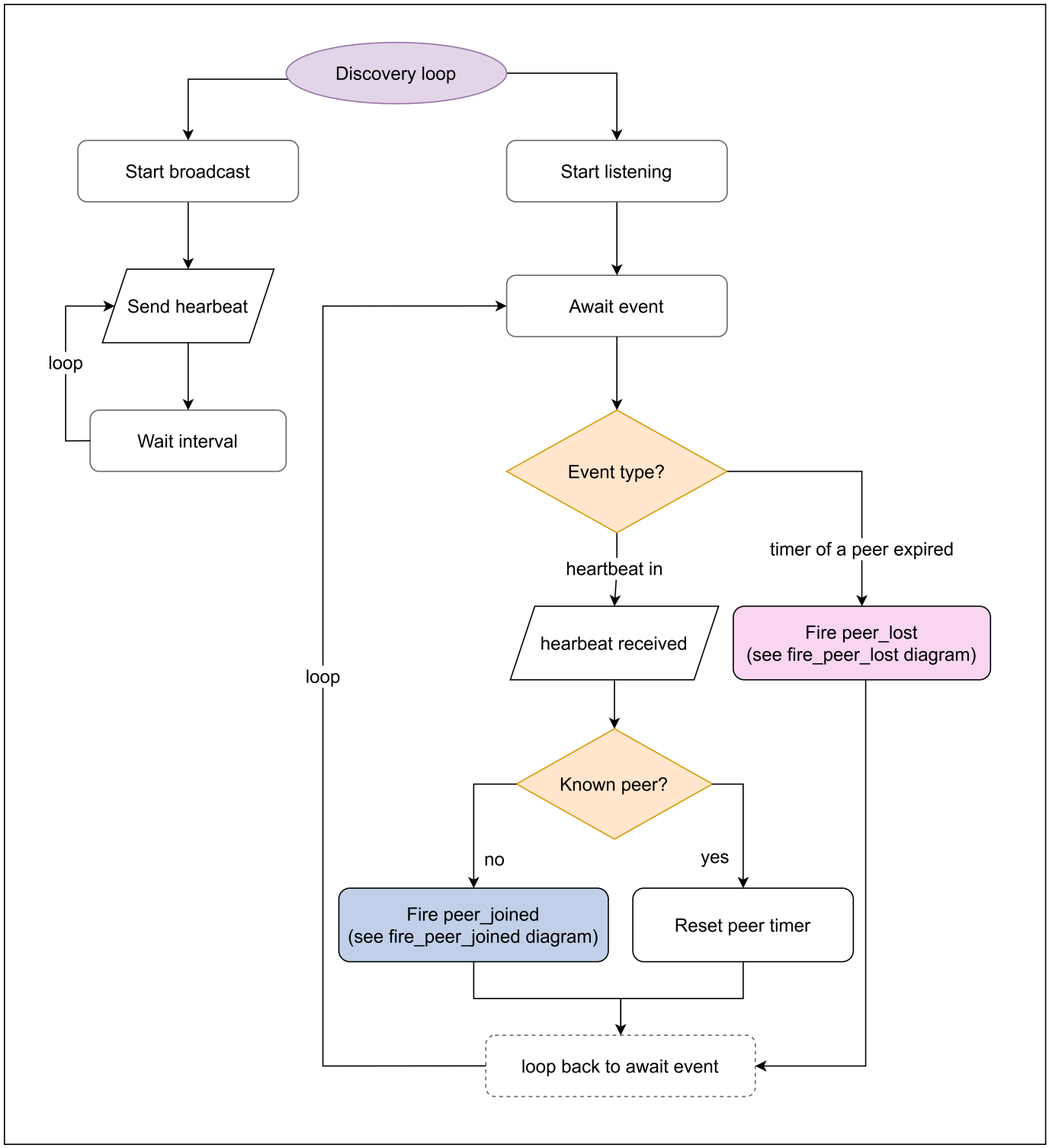


Figure A7. Peer discovery loop. The loop runs two concurrent activities. On the left, the device broadcasts: it periodically sends a heartbeat on the network and waits one interval before repeating, so that other devices learn it is alive. On the right, the device listens: it awaits an event and branches on the event type. A received heartbeat leads to a check on whether the sender is already a known peer; if it is not, the peer_joined handler is invoked (the colored call box, detailed in Figure A8), otherwise the peer's liveness timer is reset. If instead a peer's timer expires (no heartbeat was received in time), the peer_lost handler is invoked (Figure A9). After handling any event, the device loops back to awaiting the next one.

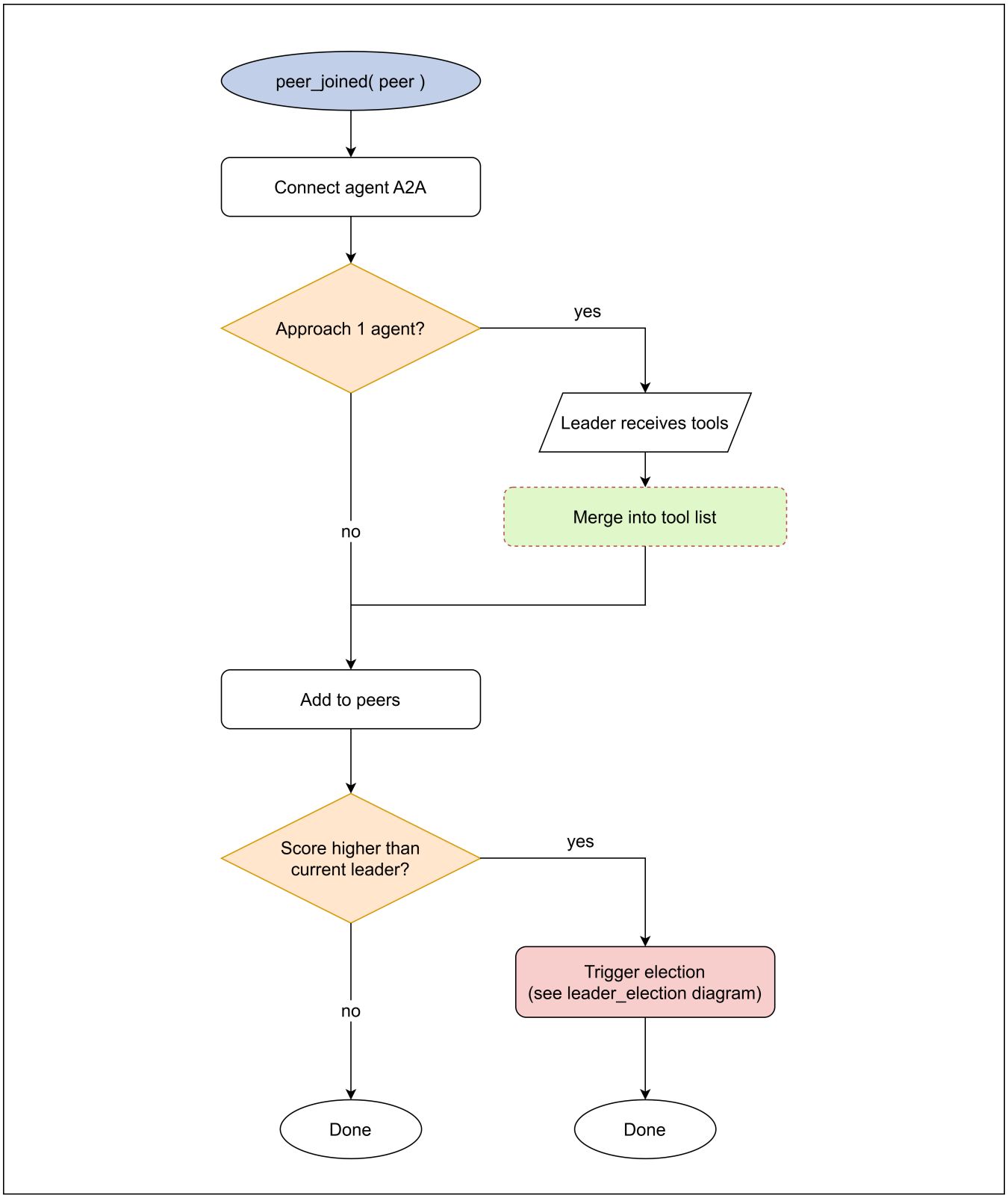


Figure A8. The peer_joined(peer) handler, run when a previously unknown peer is discovered. The device first opens an agent-to-agent (A2A) connection to the new peer. If the peer is an Approach 1 sensing agent, the leader receives that peer's tool definitions and merges them into its tool list (removing duplicates); this branch is skipped for peers that do not expose tools this way. The peer is then added to the local peer set. Finally, if the new peer's device score is higher than that of the current leader, a leader election is triggered (the colored call box, Figure A10) so that the more capable device can take over; otherwise, the handler simply terminates.

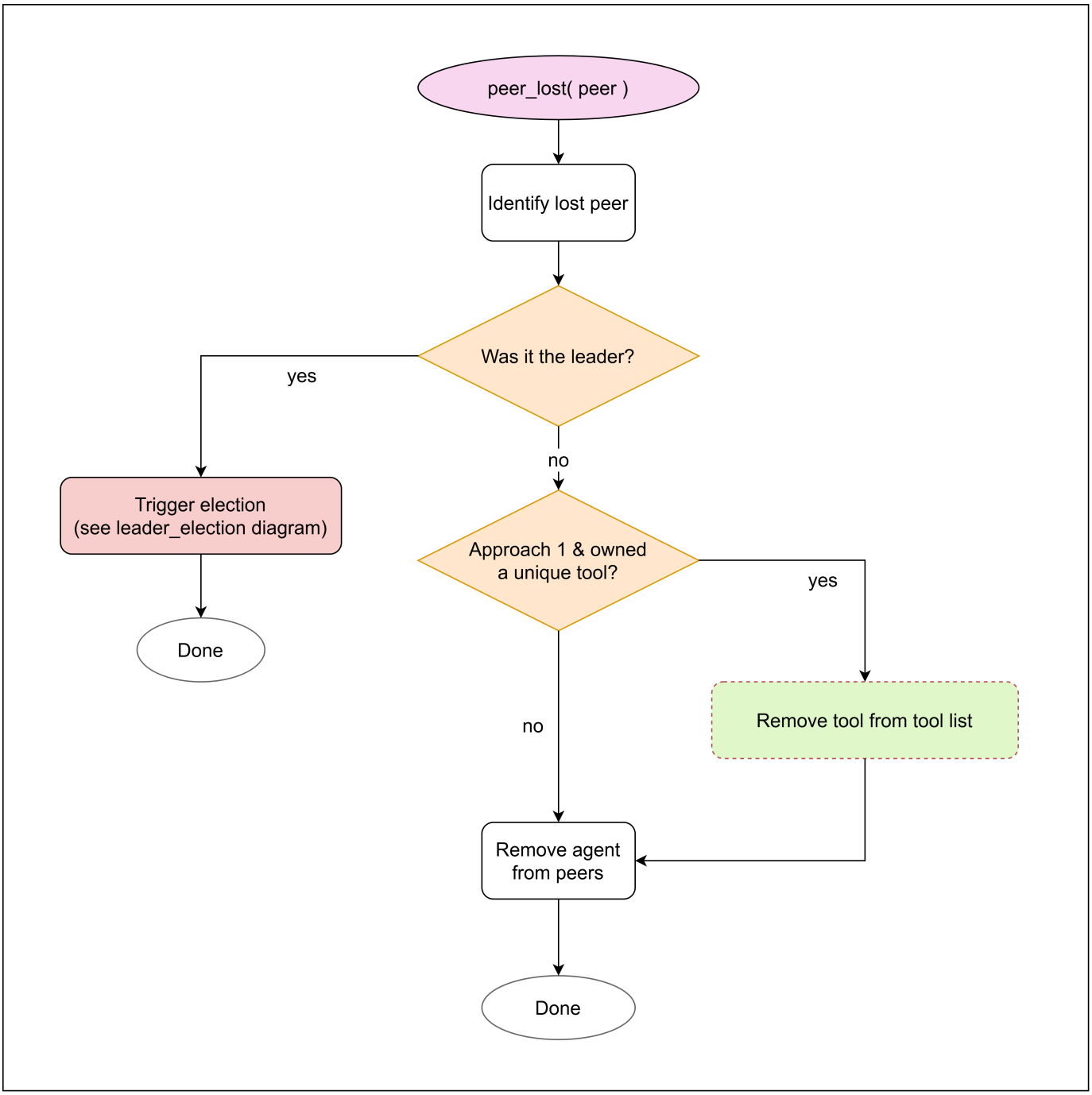


Figure A9. The peer_lost(peer) handler, run when a peer's heartbeat times out. The device identifies the lost peer and checks whether it was the current leader. If it was, a leader election is triggered (the colored call box, Figure A10) so that a surviving device can take over the coordinator role. If it was not the leader, the device checks whether the lost peer was an Approach 1 agent that owned a tool no other peer provides; in that case, the tool is removed from the leader's tool list so that the LLM is no longer offered a tool that can no longer be executed. In all non-leader cases, the peer is finally removed from the local peer set, and the handler terminates.

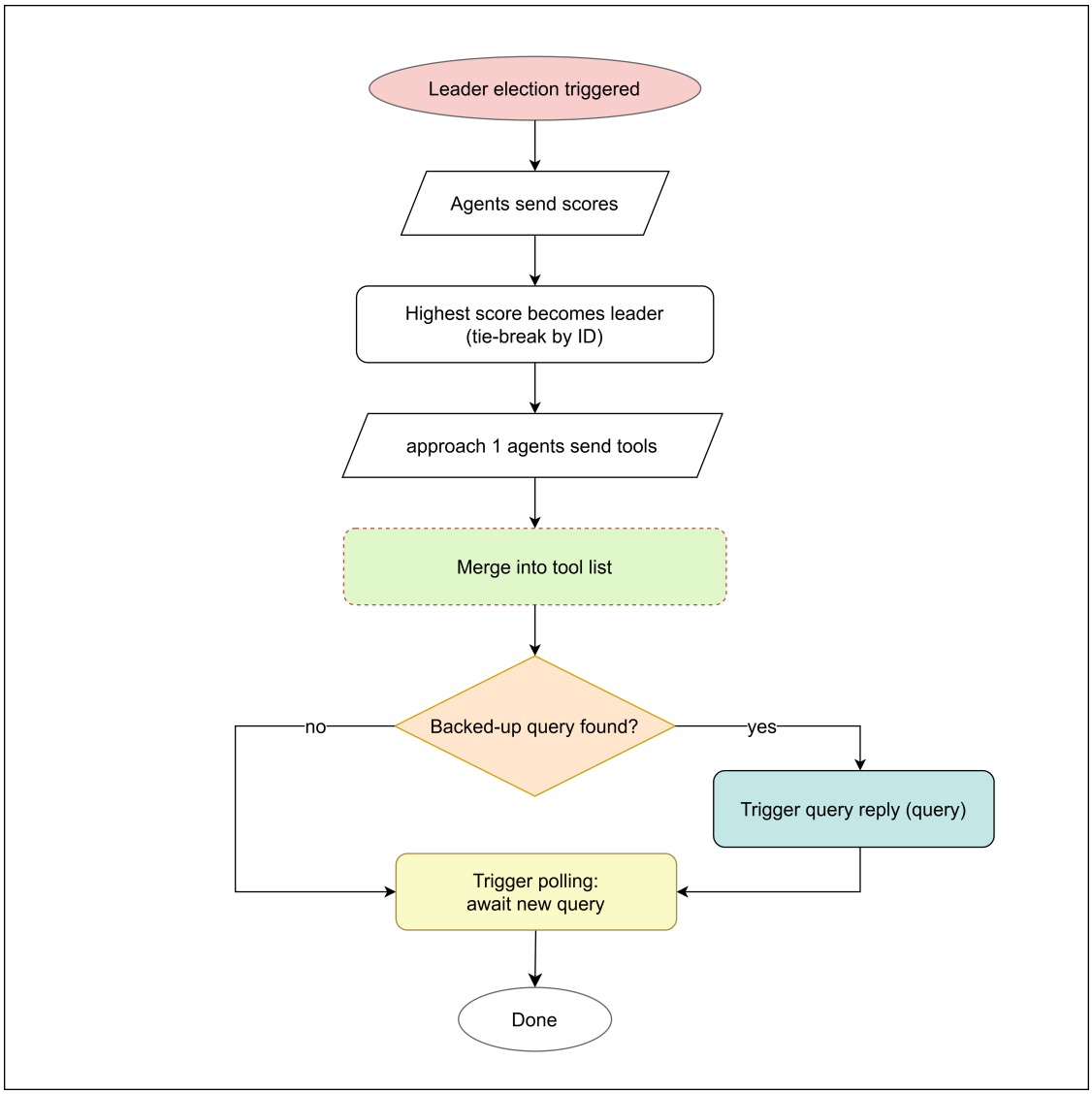


Figure A10. The leader election procedure, triggered when the leader is lost or when a more capable device joins. The candidate devices send their scores, and the device with the highest score becomes the new leader, with ties broken deterministically by agent-ID. The Approach 1 sensing agents then send their tools to the new leader, which merges them into its tool list. Because the previous leader may have been serving a request when it failed, the new leader checks whether a backed-up query was recovered from its replica: if so, it resumes that query by invoking the query reply procedure (the colored call box, Figure A12). Either way, the new leader ends by entering the polling loop that awaits the next user query (Figure A11).

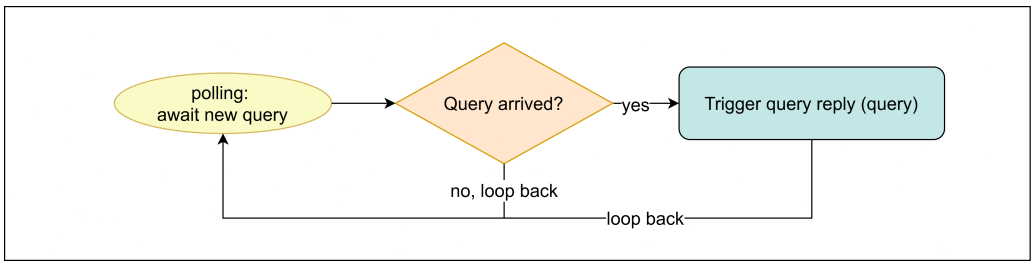


Figure A11. The leader's polling loop for user queries. The leader waits for a new query and checks whether one has arrived. When a query arrives, it invokes the query reply procedure (the colored call box, Figure A12) to answer it and then loops back to await the next query; when no query has arrived, it simply loops back and keeps waiting. This loop is the leader's steady state between requests.

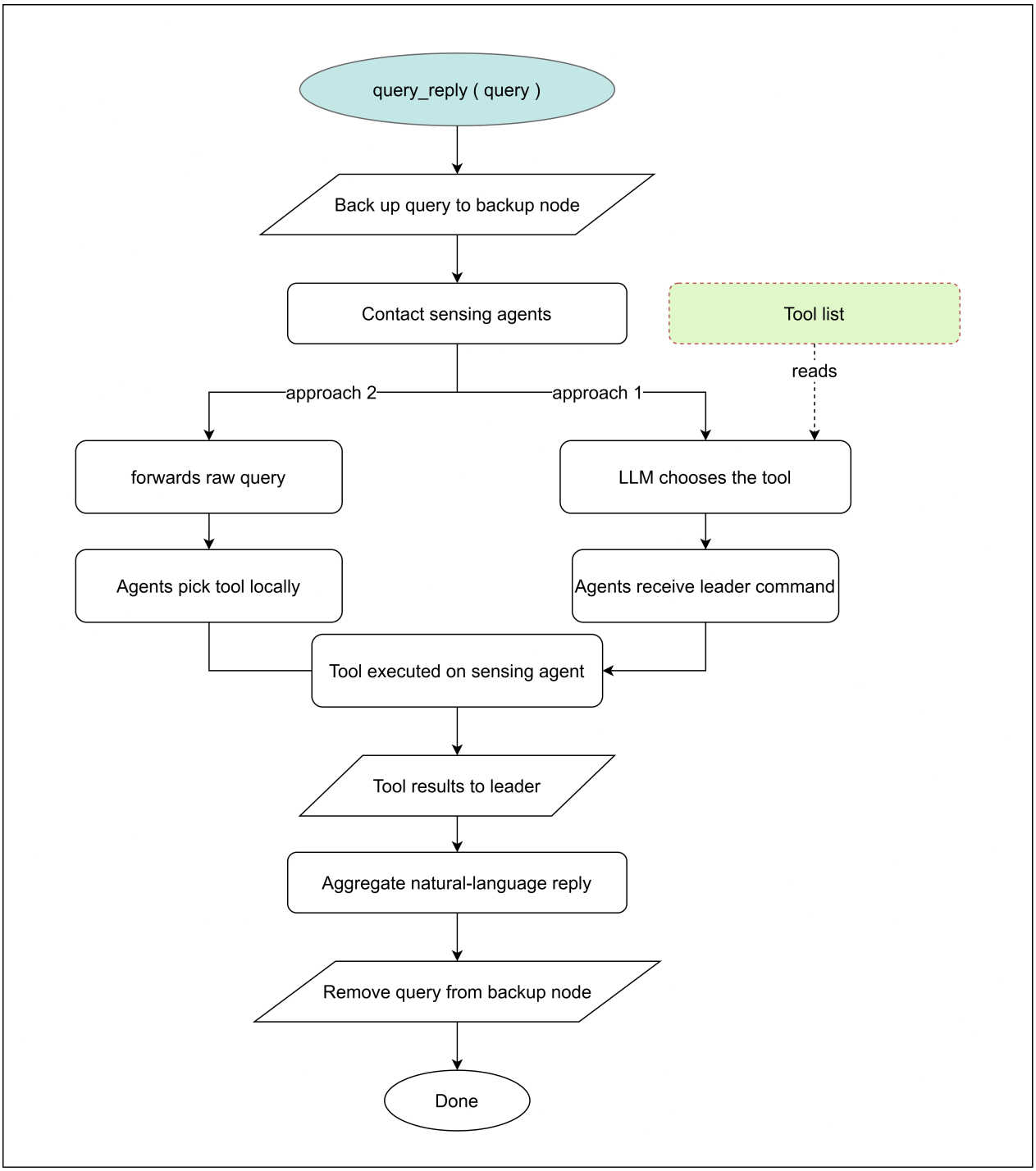


Figure A12. The query_reply(query) procedure that answers a single user query and is the point where the two approaches diverge. The leader first backs up the query to the backup node (the device with the second-highest score) so that the request survives a leader failure, then contacts the sensing agents. Under Approach 2, the leader forwards the raw query, and each sensing agent picks the matching tool locally. Under Approach 1, the leader's LLM chooses the tool, reading the candidate tools from the merged tool list (the dashed "reads" arrow), and the sensing agents receive the resulting command. In both cases, the selected tool is executed on the sensing agent, and its results are returned to the leader, which aggregates them into a single natural language reply for the user. Finally, the leader removes the now-served query from the backup node, and the procedure terminates.

References

1. Grand View Research. Video Surveillance Market (2025–2030). Technical Report. 2024. Available online: https://www.grandviewresearch.com/industry-analysis/video-surveillance-market-report (accessed on 24 June 2026).
2. Mordor Intelligence. Video Surveillance Systems Market Size and Growth Outlook 2031. Technical Report. 2025. Available online: https://www.mordorintelligence.com/industry-reports/video-surveillance-systems-market (accessed on 24 June 2026).
3. Genetec Inc. State of Physical Security 2026. Technical Report, Genetec Inc. 2026. Available online: https://www.genetec.com/binaries/content/assets/genetec/reports/report_en_state-of-physical-security-2026_web.pdf (accessed on 24 June 2026).
4. Axis Communications. Axis Perspectives 2026. Technical Report, Axis Communications. 2026. Available online: https://www.axis.com/dam/public/permalink/256299/axis-perspectives-2026-en-US_256299.pdf (accessed on 24 June 2026).
5. Andriulo, F.C.; Fiore, M.; Mongiello, M.; Traversa, E.; Zizzo, V. Edge Computing and Cloud Computing for Internet of Things: A Review. *Informatics* **2024**, *11*, 71. [CrossRef]
6. Xu, R.; Razavi, S.; Zheng, R. Edge Video Analytics: A Survey on Applications, Systems & Enabling Techniques. *IEEE Commun. Surv. Tutor.* **2023**, *25*, 2951–2982. [CrossRef]
7. Piccialli, F.; Chiaro, D.; Sarwar, S.; Cerciello, D.; Qi, P.; Mele, V. AgentAI: A comprehensive survey on autonomous agents in distributed AI for industry 4.0. *Expert Syst. Appl.* **2025**, *291*, 128404. [CrossRef]
8. Sapkota, R.; Roumeliotis, K.I.; Karkee, M. AI Agents vs. Agentic AI: A Conceptual taxonomy, applications and challenges. *Inf. Fusion* **2026**, *126*, 103599. [CrossRef]
9. Buynitsky, A.; Ehsani, S.; Pallakonda, B.; Mishra, P. Camera Control at the Edge with Language Models for Scene Understanding. In *Proceedings of the 2025 11th International Conference on Control, Automation and Robotics (ICCAR)*; IEEE: New York, NY, USA, 2025; pp. 524–530. [CrossRef]
10. Baliyar Singh, R.K.; Dash, J.K.; Reddy, K.H.K. The role of Edge-AI in edge enabled IoT systems: A comprehensive performance analysis. *Peer-to-Peer Netw. Appl.* **2025**, *19*, 35. [CrossRef]
11. Singh, R. Intelligent Agents. In *Encyclopedia of Information Systems*; Bidgoli, H., Ed.; Elsevier: New York, NY, USA, 2003; pp. 639–647. [CrossRef]
12. Erdogan, L.E.; Lee, N.; Jha, S.; Kim, S.; Tabrizi, R.; Moon, S.; Hooper, C.; Anumanchipalli, G.; Keutzer, K.; Gholami, A. TinyAgent: Function Calling at the Edge. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2024. [CrossRef]
13. Qin, Y.; Liang, S.; Ye, Y.; Zhu, K.; Yan, L.; Lu, Y.; Lin, Y.; Cong, X.; Tang, X.; Qian, B.; et al. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In Proceedings of the The Twelfth International Conference on Learning Representations, Vienna, Austria, 7–11 May 2024. Available online: https://openreview.net/forum?id=dHng2O0Jjr (accessed on 11 May 2026).
14. Shi, Z.; Wang, Y.; Yan, L.; Ren, P.; Wang, S.; Yin, D.; Ren, Z. Retrieval Models Aren't Tool-Savvy: Benchmarking Tool Retrieval for Large Language Models. In *Proceedings of the Findings of the Association for Computational Linguistics: ACL 2025, Vienna, Austria, 27 July–1 August 2025*; Che, W., Nabende, J., Shutova, E., Pilehvar, M.T., Eds.; Association for Computational Linguistics: Stroudsburg, PA, USA, 2025; pp. 24497–24524. [CrossRef]
15. SharpAI. DeepCamera—Open-Source AI Camera Skills Platform. GitHub Repository. 2026. Available online: https://github.com/SharpAI/DeepCamera (accessed on 25 May 2026).
16. Wang, X.; He, J.; Tang, Z.; Guo, J.; Lou, J.; Qian, L.; Wang, T.; Jia, W. Adaptive AI Agent Placement and Migration in Edge Intelligence Systems. *arXiv* **2025**, arXiv:2508.03345.
17. Zhang, B.; Mor, N.; Kolb, J.; Chan, D.S.; Lutz, K.; Allman, E.; Wawrzynek, J.; Lee, E.; Kubiatowicz, J. The Cloud is Not Enough: Saving IoT from the Cloud. In Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15), Santa Clara, CA, USA, 6–7 July 2015. Available online: https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/zhang (accessed on 28 June 2026).
18. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. [CrossRef]
19. Agrawal, R.; Verma, P.; Sonanis, R.; Goel, U.; De, A.; Kondaveeti, S.A.; Shekhar, S. Continuous Security in IoT Using Blockchain. In *Proceedings of the 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*; IEEE: New York, NY, USA, 2018; pp. 6423–6427. [CrossRef]
20. Nguyen, T.; Nguyen, H.; Nguyen Gia, T. Exploring the integration of edge computing and blockchain IoT: Principles, architectures, security, and applications. *J. Netw. Comput. Appl.* **2024**, *226*, 103884. [CrossRef]
21. Wu, G.; Jiang, J.; Jiang, K.; Liu, X.; Nie, L. DSwinIR: Rethinking Window-Based Attention for Image Restoration. *IEEE Trans. Pattern Anal. Mach. Intell.* **2026**, *48*, 4350–4366. [CrossRef]
22. Foundation, R.P. Introducing the Raspberry Pi AI HAT+ 2: Generative AI on Raspberry Pi 5. 2025. Available online: https://www.raspberrypi.com/news/introducing-the-raspberry-pi-ai-hat-plus-2-generative-ai-on-raspberry-pi-5/ (accessed on 11 May 2026).

23. Singh, S.; Sharma, K.; Karna, B.K.; Raj, P. Pruning and Quantization for Deeper Artificial Intelligence (AI) Model Optimization. In *Proceedings of the Intelligent Control, Robotics, and Industrial Automation*; Sharma, S., Subudhi, B., Sahu, U.K., Eds.; Springer: Singapore, 2023; pp. 933–945. [CrossRef]
24. Kordjukovs, N.; Pau, D.P. Complexity Analysis for Categorized Edge Language Models. *Symmetry* **2026**, *18*, 766. [CrossRef]
25. Singh, A.; Ehtesham, A.; Lambe, M.; Grogan, J.; Singh, A.; Kumar, S.; Muscariello, L.; Pandey, V.; De Saint Marc, G.S.; Chari, P.; et al. Evolution of AI Agent Registry Solutions: Centralized, Enterprise, and Distributed Approaches. In *Proceedings of the 2025 IEEE 7th International Conference on Cognitive Machine Intelligence (CogMI)*; IEEE: New York, NY, USA, 2025; pp. 507–515. [CrossRef]
26. Kim, Y.; Gu, K.; Park, C.; Park, C.; Schmidgall, S.; Heydari, A.A.; Yan, Y.; Zhang, Z.; Zhuang, Y.; Liu, Y.; et al. Towards a Science of Scaling Agent Systems. *arXiv* **2026**, arXiv:2512.08296.
27. Hu, E. The Influence of the Topological Structure of Multiagent Systems on the Factual Consistency of Collaboratively Generated Text. In *Proceedings of the 2025 IEEE 4th International Conference of Safe Production and Informatization (IICSPI)*; IEEE: New York, NY, USA, 2025; pp. 179–183. [CrossRef]
28. Sutter, M. Meet NullClaw: The 678KB Zig AI Agent Framework Running on 1 MB RAM and Booting in Two Milliseconds. *MarkTech Post*, 2 March 2026.
29. LangChain. LangGraph. 2025. Available online: https://github.com/langchain-ai/langgraph (accessed on 11 May 2026).
30. Hao, S.; Liu, T.; Wang, Z.; Hu, Z. ToolkenGPT: Augmenting Frozen Language Models with Massive Tools via Tool Embeddings. In *Proceedings of the Advances in Neural Information Processing Systems*; Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S., Eds.; Curran Associates, Inc.: Red Hook, NY, USA, 2023; Volume 36, pp. 45870–45894. Available online: https://proceedings.neurips.cc/paper_files/paper/2023/hash/8fd1a81c882cd45f64958da6284f4a3f-Abstract-Conference.html (accessed on 28 June 2026).
31. Liu, X.; Peng, Z.; Yi, X.; Xie, X.; Xiang, L.; Liu, Y.; Xu, D. ToolNet: Connecting Large Language Models with Massive Tools via Tool Graph. *arXiv* **2024**, arXiv:2403.00839.
32. Wang, R.; Han, X.; Ji, L.; Wang, S.; Baldwin, T.; Li, H. ToolGen: Unified Tool Retrieval and Calling via Generation. In Proceedings of the The Thirteenth International Conference on Learning Representations, Singapore, 24–28 April 2025. Available online: https://openreview.net/forum?id=XLMAMmowdY (accessed on 28 June 2026).
33. Lumer, E.; Subbiah, V.; Burke, J.; Basavaraju, P.; Huber, A. Toolshed: Scale Tool-Equipped Agents with Advanced RAG-Tool Fusion and Tool Knowledge Bases. In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence - Volume 3: ICAART*; INSTICC, SciTePress: Setúbal, Portugal, 2025; pp. 1180–1191. [CrossRef]
34. Kokane, S.; Zhu, M.; Awalgaonkar, T.; Zhang, J.; Hoang, T.; Prabhakar, A.; Liu, Z.; Lan, T.; Yang, L.; Tan, J.; et al. ToolScan: A Benchmark for Characterizing Errors in Tool-Use LLMs. In Proceedings of the Building Trust Workshop at the International Conference on Learning Representations (ICLR), Singapore, 28 April 2025. Available online: https://arxiv.org/abs/2411.13547 (accessed on 28 June 2026).
35. Kate, K.; Pedapati, T.; Basu, K.; Rizk, Y.; Chenthamarakshan, V.; Chaudhury, S.; Agarwal, M.; Abdelaziz, I. LongFuncEval: Measuring the Effectiveness of Long Context Models for Function Calling. *arXiv* **2025**, arXiv:2505.10570.
36. Kadar Mohamed Jabarullah, M.A.; Chakka, S.N.; Kanungo, S. Edge-Cloud Hybrid Architecture for Distributed AI Applications. In *Proceedings of the 2025 International Conference on Computing Technologies & Data Communication (ICCTDC)*; IEEE: New York, NY, USA, 2025; pp. 1–5. [CrossRef]
37. Abdelmoniem , A.M.; Jaber , M.; Anwar , A.; Zhang , Y.; Gao , M. Towards a Decentralized Collaborative Framework for Scalable Edge AI. *Future Internet* **2024**, *16*, 421. [CrossRef]
38. Patil, S.G.; Mao, H.; Yan, F.; Ji, C.C.J.; Suresh, V.; Stoica, I.; Gonzalez, J.E. The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models. In *Proceedings of the 42nd International Conference on Machine Learning, Vancouver, BC, Canada, 13–19 July 2025*; Singh, A., Fazel, M., Hsu, D., Lacoste-Julien, S., Berkenkamp, F., Maharaj, T., Wagstaff, K., Zhu, J., Eds.; Proceedings of Machine Learning Research; PMLR: Boston, MA, USA, 2025, Volume 267, pp. 48371–48392. Available online: https://proceedings.mlr.press/v267/patil25a.html (accessed on 28 June 2026).
39. Patel, P.; Choukse, E.; Zhang, C.; Shah, A.; Goiri, Í.; Maleki, S.; Bianchini, R. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *Proceedings of the 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*; IEEE: New York, NY, USA, 2024; pp. 118–132. . [CrossRef]
40. Pope, R.; Douglas, S.; Chowdhery, A.; Devlin, J.; Bradbury, J.; Heek, J.; Xiao, K.; Agrawal, S.; Dean, J. Efficiently Scaling Transformer Inference. In *Proceedings of the Proceedings of Machine Learning and Systems*; Song, D., Carbin, M., Chen, T., Eds.; MLSys: Indio, CA, USA, 2023; Volume 5, pp. 606–624. Available online: https://proceedings.mlsys.org/paper_files/paper/2023/file/c4be71ab8d24cdfb45e3d06dbfca2780-Paper-mlsys2023.pdf (accessed on 12 July 2026).

41. Ahmad, J.; Latif, S.; Khan, I.U.; Alshehri, M.S.; Khan, M.S.; Alasbali, N.; Jiang, W. An interpretable deep learning framework for intrusion detection in industrial Internet of Things. *Internet Things* **2025**, *33*, 101681. [CrossRef]
42. Fabbri, A.R.; Kryściński, W.; McCann, B.; Xiong, C.; Socher, R.; Radev, D. SummEval: Re-evaluating Summarization Evaluation. *Trans. Assoc. Comput. Linguist.* **2021**, *9*, 391–409. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.