Politecnico di Torino

Politecnico di Torino

Computer Engineering (Ingegneria Informatica)

Master's Degree Thesis

An AI-Driven Multimodal Framework for AR-Guided PCB Operations: The ARBoard Intelligent Backbone

Supervisors
Prof. Paolo BERNARDI
Dr. Giorgio INSINGA

Candidate
Reem KHATTAR

March 2026

Acknowledgements

This thesis marks the end of an amazing and challenging chapter of my life, one that has shaped me in many ways and will stay with me forever.

To the Lebanese University, thank you for placing your confidence in me and giving me the opportunity to pursue this double-degree Master's program at Politecnico di Torino, an experience that has profoundly enriched both my academic and personal journey.

To my supervisors, Paolo Bernardi and Giorgio Insinga, thank you for entrusting me with this big responsibility of being part of the development of ARBoard from its very beginning. Being involved in every stage and witnessing it grow has been an incredibly rewarding experience. Your guidance, expertise, and unwavering support have been invaluable.

To my parents, who kept believing in me even when I didn't, who never made me feel like I couldn't do what I wanted to do or be who I wanted to be. Thank you for opening every door so I could find my own way, for giving me courage to dream big, and for always being the steady ground beneath my feet. Everything I am today is because of you, you have given me more than you ever had and I will never be able to thank you enough for that.

To my little sister, thank you for being the remarkable person you are and for always being there in your own unique way. Watching you grow is a constant inspiration. I am so proud of you, and even prouder to be your sister. Thank you for not letting me take myself too seriously and for being a quiet but constant source of strength. You give me the will to be better every day and I couldn't be more grateful for you.

To my family, I am so fortunate to have been born into a home where love is so abundant. Knowing I have you in my corner, cheering for me with so much pride, keeps me moving forward. Thank you for the way you've always celebrated me; your endless faith in me is a gift.

To everyone who shared this journey with me, friends, colleagues, mentors, thank you for your encouragement, and for making this path more meaningful and memorable.

Most importantly, I thank God for everything. I know that I would surely not be here without You.

> *"I hereby command you: Be strong and courageous; do not be frightened or dismayed, for the Lord your God is with you wherever you go." (Joshua 1:9)*

Abstract

This thesis develops an integrated intelligent framework for ARBoard, an Augmented Reality (AR) assistance platform designed to optimize manual operations and diagnostic workflows on Printed Circuit Boards (PCBs). While the broader system includes spatial mapping and visualization, this work contributes a software backend that unifies a specialized text-and-voice AI assistant with automated board data extraction. These two functional layers are integrated via a centralized relational database.

The **Reasoning Layer** introduces a custom Benchmark Engine designed to quantify tool-calling performance across various open-source architectures, including the GPT-OSS, Qwen, and MiniMax series, to compare and evaluate the models. To bridge the domain gap in technical PCB documentation, a Teacher-Student distillation pipeline was implemented, using high-reasoning models, to synthesize high-fidelity, bilingual synthetic dataset. This facilitated the Supervised Fine-Tuning of a compact 8-billion parameter model.

Comparative testing against the benchmark, showed that the model successfully internalizes specialized knowledge, achieving performance parity with significantly larger parameter-count models. The model maintained high-intent recognition accuracy while ensuring the speech processing and inference are done in under 10 seconds, enabling real-time performance.

The **Perception Layer** facilitates automated data entry through a hybrid Optical Character Recognition (OCR) pipeline. By integrating traditional Template Matching with an AI-based autosuggestion engine, the system captures component labels that often confound standard OCR models. The pipeline is reinforced by Euler-number topological verification, which serves as a robust geometric invariant to disambiguate visually similar characters. A multi-stage geometric filtering process, utilizing area-based thresholding helped prune non-textual artifacts and noise. A specialized spatial proximity grouping algorithm then executes the structural reconstruction of component labels from fragmented bounding box detections.

Experimental validation showed that this nearly eliminates false positives in noisy user manual and schematics files. This approach significantly reduces manual data-entry latency and human-errors, and ensures automatic synchronization with the system's database.

By deploying the reasoning layer within a fully local execution environment, the framework ensures strict data privacy and reduces inference latency. The perception layer allows extracting the board information even when only basic documentation is provided. The integration of conversational intelligence with robust geometric perception establishes a reliable, real-time assistant, effectively reducing the manual and cognitive overhead of PCB maintenance.

Contents

List of Figures

List of Tables

I. Introduction

I.1 Motivation

In complex engineering environments, such as electronic board assembly and high-density PCB (Printed Circuit Board) prototyping, the volume of technical metadata exceeds the capacity for efficient manual retrieval. A single modern circuit board can contain thousands of individual components, nets, and pins, all documented across fragmented sources: IPC-2581 manufacturing files [1], digital schematics, and lengthy PDF user manuals. For a technician or engineer, the process of locating a specific signal or verifying the specifications of a component requires constant context switching, moving their attention away from the physical board to search through records. This manual retrieval process is not only time-consuming, but introduces a significant margin for human error, which can be costly in high-precision manufacturing.

The motivation for this work comes from the need to provide a seamless, hands-free information flow that follows the natural workflow of the technician. To achieve this, two critical hurdles must be overcome: Data Accessibility and Interaction Intelligence.

While modern production standards such as IPC-2581 provide rich, machine-readable metadata, they are not always available. In legacy hardware maintenance or third-party repair scenarios, design files may be lost, leaving only human-readable documentation, such as PDF schematics and placement drawings. To ensure the AR system remains functional in these cases, a secondary data pipeline is required. This work explores the development of Text Extraction modules that utilize Template Matching [2] and Computer Vision to reverse-engineer a searchable spatial database from these visual formats. Rather than recreating a full electrical digital twin, this module focuses, in this case, on identifying individual components and calculating their precise coordinates on the board. By extracting reference designators and mapping their physical positions from unstructured PDFs or images, the system provides the necessary spatial grounding for the AR interface to function even in the absence of primary production files.

Even with a populated database, traditional search-and-click interfaces are inefficient in industrial environments where the user's hands are occupied with tools or testing equipment. This thesis focuses on the development of an Intelligent Backbone, an AI Assistant that moves beyond simple chatbots to function as an AI Agent. Unlike standard Large Language Models (LLMs) limited to text generation, an agentic system is capable of intent mapping: translating noisy, natural language queries into deterministic software actions, such as highlighting a specific net or component in an Augmented Reality (AR) viewer.

The transition from a general-purpose LLM to an industrial controller requires

extreme reliability and strict data security. In an industrial startup context, PCB designs and manufacturing files represent highly sensitive intellectual property. To ensure total data sovereignty and privacy, this research utilizes a local-first architecture. By deploying specialized 8B-parameter models on high-performance local infrastructure, the system processes sensitive metadata without external cloud dependency.

However, running locally necessitates overcoming the reliability gap. Smaller local models are often more prone to hallucinations and lack the phonetic robustness to handle technical shorthand (e.g., misinterpreting "R1" as "Are one"). Consequently, this work is motivated by the engineering requirement to build a robust middleware that includes specialized data grounding, synthetic dataset generation for domain-specific fine-tuning, and a rigorous benchmarking framework to ensure that every AI-driven action is accurate, validated, and private.

Another visual diagnostic module was added as well; the thermal analysis module lets technicians map infrared (thermal) images of a powered PCB onto the board's spatial database so that per-component temperatures can be read automatically and tied to reference designators. That supports predictive maintenance and fault diagnosis by correlating heat anomalies with specific components.

I.2 Problem Statement

The development of an intelligent, AR-driven interface for PCB manufacturing faces a critical "Information-to-Action" gap. While the vision of a hands-free AI assistant is compelling, the practical implementation is hindered by different technical hurdles.

The Metadata Availability Gap: For different reasons, industrial environments often lack structured, machine-readable design files (like IPC-2581). When only static PDF schematics or placement drawings are available, the AR system cannot automatically "know" where components are located. There is currently a lack of automated, low-overhead pipelines that can reconstruct a functional technical database from these legacy visual formats to enable AR overlays, since standard OCR methods *mentioned in the Background section* have failed to read textual labels in visually dense and font-variable board schematics and placement images, or in the available documentation.

The Reliability Gap in Local LLMs: PCB designs are sensitive intellectual property. To protect industrial intellectual property ensuring that the critical board design data remains confined to the local network, the AI must run on local infrastructure. While on-premise deployment ensures data sovereignty, it mandates the use of smaller-parameter models (SLMs) due to edge hardware constraints. These models introduce a significant reliability gap: achieving the logical consistency required for **deterministic function calling**. Ensuring these models consistently generate valid, schema-adherent JSON syntax without the reasoning overhead of massive cloud-based architectures requires

specialized fine-tuning and the implementation of **constrained decoding** or robust output-parsing strategies.

The Semantic Noise of Industrial Environments: Standard Speech-to-Text (STT) models are not trained on the technical shorthand of electronics (e.g., "R1", "U5"). In a noisy workshop, these terms are often misinterpreted, causing the downstream AI agent to fail because it cannot map the noisy transcription to the actual components in the database.

I.3 Objectives

The primary goal of this thesis is to engineer and evaluate a high-reliability "Intelligent Backbone" that enables a deterministic natural language interface for PCB documentation supported by an automated data extraction layer. The specific objectives are:

Develop a Component Extraction Pipeline: Implement a computer vision module using template matching to identify component reference designators and their positions within PDF documentation, providing the necessary spatial data when production files are unavailable.

Architect a Secure, Local Agentic Pipeline: Design a reasoning architecture on local GPU infrastructure that separates conversational intent from structured tool-call generation, ensuring data privacy while maintaining high operational accuracy and real-time communication.

Optimize Reliability via Fine-Tuning: Create a synthetic dataset generator that simulates industrial queries and STT noise to fine-tune open-source 8B-class models (e.g., Qwen, Llama) via LoRA [3] for domain-specific precision and deterministic JSON output.

Evaluate Performance through Local Benchmarking: Build an evaluation engine to rigorously compare different local models on metrics of JSON validity, tool selection accuracy, and parameter precision, identifying the most reliable "Intelligent Backbone" for the ARBoard platform [4].

II. Background

The development of the ARBoard [4] "Intelligent Backbone" operates at the intersection of Agentic AI, Edge Computing, and Computer Vision to revolutionize the lifecycle of Printed Circuit Boards (PCBs). This chapter establishes a theoretical framework for bridging the gap between unstructured industrial documentation, such as technical data sheets and schematics, and real-time, context-aware augmented reality assistance during assembly and debugging. The AR viewer, developed as a parallel component of the ARBoard platform, superimposes a suite of functional screens and interactive features onto the technician's field of vision. So far, it has provided a Component Mode and a Net Mode to trace signals and connections, an Assembly Mode to simplify the process of assembling the board, a digital twin of the board for clearer visualization, and an AI panel to assist at any moment throughout the process. In practice, this means that component data, schematic overlays, and the AI assistant panel are projected directly into the technician's field of view, enabling hands-free access to board documentation during assembly and debugging. To achieve this, the system fulfills two critical architectural requirements: first, a spatial database that indexes every reference designator to its precise coordinates on the physical PCB; and second, a reliable command channel that translates natural language queries into deterministic visual overlays. Moving inference to the edge, the backbone ensures that high-precision component highlighting and signal tracing remain responsive and secure, facilitating a significant reduction in time-to-market for complex electronics.

Figure II.1: The ARBoard augmented reality interface as seen through smart glasses.

II.1 PCB Data Standards and Documentation

In the electronics industry, the transfer of data from design to manufacturing is the foundation of any automation effort; when parsed, these data describe the

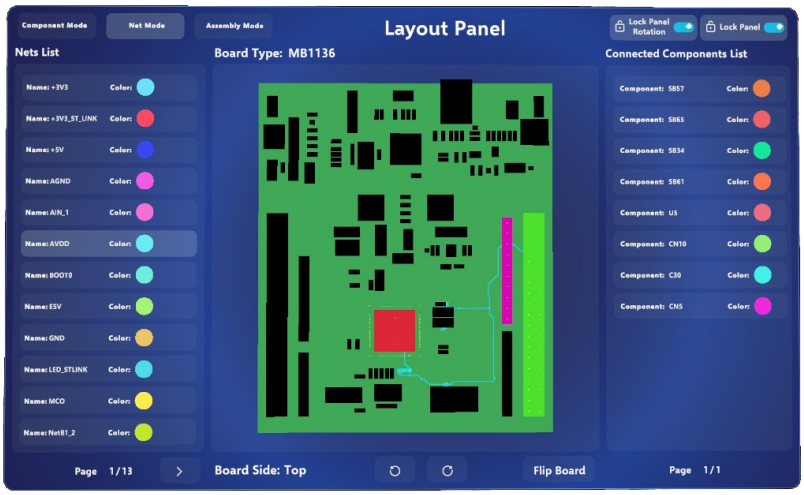


Figure II.2: ARBoard Net Mode view highlighting all pads connected to a selected signal net.

physical and logical structure of a board. Modern PCB manufacturing relies on three main standards. While Gerber remains the "universal language," IPC-2581 and ODB++ are preferred for complex, high-speed, or high-density (HDI) designs [1].

- **IPC-2581**: The IPC-2581 standard (also known as DPMX) is a vendor-neutral, XML-based data format that serves as a single source of truth for PCB design, fabrication, and assembly. Moving beyond the limitations of traditional Gerber files, which function primarily as visual vector instructions, IPC-2581 acts as an intelligent Digital Twin. It integrates logical netlists, the Bill of Materials (BOM), and precise component placement data into a single file. By consolidating the physical layout with manufacturing intent, the standard facilitates automated Design for Manufacturing (DFM) checks and ensures bidirectional data integrity between the designer and the fabrication house.

- **The Production Data Gap**: Despite the advantages of integrated formats, a data gap frequently occurs in maintenance, repair, and legacy hardware environments. In these scenarios, unified design files like IPC-2581 are often unavailable, leaving only fragmented documentation such as static PDF schematics or flat 2D placement drawings. Without a structured data export, the underlying intelligence of the PCB, specifically the link between a component's logical identity and its physical coordinates, is lost, necessitating manual or automated reconstruction. And there doesn't exist a straightforward way that does it automatically and efficiently yet.

- **Optical Character Recognition for PCB Text**: A key step in reconstructing information from legacy documentation is the automatic

extraction of textual annotations from board drawings and schematics. Optical Character Recognition (OCR) techniques can be applied to identify reference designators, net names, and other textual labels, even when they appear in varying fonts, rotations, or scales. Modern OCR pipelines, often based on convolutional or transformer-based detectors combined with sequence recognizers, enable robust detection of small, densely packed characters on noisy engineering drawings [5]. The resulting symbol-level text data provides the semantic layer that links visual regions of a document to logical entities in the underlying circuit.

- **Spatial Reconstruction via Template Matching**: To bridge this gap when production files are missing, Computer Vision techniques can be employed to extract spatial data from available visual documentation. Template Matching is a digital image processing technique used to locate specific sub-images (templates) within a larger target image [2]. In the context of PCB documentation, it can be used to localize components, fiducials, or symbol templates and to refine the positions of text regions identified by OCR. By calculating the pixel positions of these matched regions and translating them into a real-world coordinate system, the method contributes to rebuilding a structured representation of the board. This process provides the Spatial Mapping necessary for an Augmented Reality (AR) system to orient its visual indicators, allowing it to know exactly where to project information on a physical board even in the absence of original CAD data. The interest in this work is to identify the labels of the board's components and their positions on the layout.

- **Thermal Imaging for PCB Fault Analysis**: Beyond visual documentation, infrared cameras provide a complementary diagnostic layer by capturing the heat distribution across a populated PCB. Faulty components, mechanically stressed solder joints, and traces carrying excess current all dissipate anomalous power, producing thermal signatures that are invisible under white-light inspection [6]. Correlating these anomalies with specific component identities requires aligning the thermal image with the spatial layout of the board. This alignment is non-trivial: a handheld thermal camera and the reference documentation differ in viewpoint, scale, and geometric projection, so a rectification step is necessary before per-component temperature statistics can be extracted. The standard tool for this is the perspective transform, or homography. A homography is a 3×3 matrix that describes a linear map in projective space between two planar views of the same flat scene captured from different positions. Given four or more point correspondences between the two views, the matrix is estimated by solving the Direct Linear Transform (DLT) system, and one image is then warped onto the coordinate system of the other. Once the thermal image is rectified to the reference plane, each component's bounding polygon from the spatial database can be projected onto it, and the enclosed pixel values sampled to yield per-component temperature statistics such as mean and peak tempera-

ture.

II.2 Large Language Models and AI assistants

Throughout this thesis, the term *Agentic AI* refers to large language models that are coupled with external tools and memory to perform multi-step decision-making, rather than producing stand-alone text responses. *Tool calling* denotes the mechanism by which an LLM selects and parameterizes external functions in a structured format (such as JSON), and *context injection* describes the practice of directly supplying task-relevant structured data into the model's input, in contrast to retrieving passages at inference time via RAG.

- **From Generative to Agentic AI**: Large Language Models (LLMs) have evolved from simple text-prediction engines into sophisticated reasoning cores. The current state of the art in Artificial Intelligence has shifted from Generative AI, which focuses on human-like text production, to Agentic AI, which treats Large Language Models (LLMs) as decision-makers capable of interacting with external tools and environments.

- **Instruction Following and Determinism**: For industrial applications, an agent must translate natural language into deterministic software actions.

- **The Mechanism of Tool Calling**: Rather than generating a text-only response, the model produces a structured JSON object containing a function name and specific parameters. Tool Calling is the process by which an LLM identifies that a user's request requires an external action returned as a structured data object (typically in JSON format) containing the name of a function and the necessary arguments.

- **Data Grounding: Context Injection vs. RAG**: A critical requirement for an industrial assistant is complete visibility of the relevant data describing the system under analysis. A common approach for integrating external knowledge into large language models is Retrieval-Augmented Generation (RAG), which retrieves relevant information from a dataset at inference time and incorporates it into the model's prompt [7]. While RAG is effective for large and dynamic knowledge bases, it may introduce uncertainty due to retrieval errors or incomplete document selection. To mitigate these issues in scenarios where the dataset is relatively constrained and well-defined, an alternative strategy is context injection. In this approach, the relevant information describing all components of the system is directly incorporated into the model's input context, ensuring that the model has access to the complete set of available data during generation.

- **Relational Grounding**: Precise metadata—including logical netlists and component lists—can be stored in a centralized relational database (e.g., SQLite) and injected directly into the LLM's prompt [8].

- **Knowledge Internalization**: Another important aspect is the internalization of domain-specific knowledge. Through Supervised Fine-Tuning (SFT), the base language model is further trained on specialized data related to printed circuit boards (PCBs), enabling it to learn domain-relevant terminology, structures, and relationships. This process allows the model to better understand and generate responses about PCB documentation while maintaining relatively low inference latency compared to larger general-purpose models.

- **Privacy and Data Sovereignty**: Deploying Large Language Models (LLMs) within an industrial startup context introduces constraints that differ significantly from consumer-grade AI applications. Maintaining Data Sovereignty is a non-negotiable requirement; on-premise execution ensures that sensitive intellectual property and proprietary telemetry data remain within a closed-loop system, mitigating the security risks of third-party API transmission. Furthermore, local deployment provides Deterministic Latency, removing the unpredictability inherent in network-dependent inference. This infrastructure allows for high-fidelity, secure task execution without the risks associated with cloud-based data dependency.

- **Architectural Efficiency: Compact vs. Hyperscale Models**: While frontier-scale models provide high-order reasoning, their massive architectures—often estimated in the trillions of parameters—require distributed cloud clusters that introduce significant operational overhead. In contrast, Compact-Parameter Models are designed for local execution on modern GPU workstations. The primary technical challenge for these architectures is "Reasoning Density." Unlike hyperscale models that possess a vast cognitive surplus, compact models must be rigorously optimized to match the tool-calling reliability of their larger counterparts. This "density" constraint necessitates a shift from general-purpose pre-training to specialized, task-aligned architectures.

- **Specialized Optimization: PEFT and Knowledge Distillation**: To bridge the performance gap between compact and hyperscale models, Parameter-Efficient Fine-Tuning (PEFT) techniques, specifically Low-Rank Adaptation (LoRA), are employed [3]. LoRA is utilized to inject trainable rank-decomposition matrices into each layer of the Transformer architecture which can allow the adaptation of a model to specialized industrial tasks, such as precise tool-calling or domain-specific logic, by training only a minimal subset of external adapter weights while keeping the core backbone frozen. The Unsloth framework [9] provides a memory-efficient implementation of LoRA adapter attachment and model loading that is specifically optimized for single-GPU workstations, enabling full fine-tuning sweeps on hardware that would

otherwise be insufficient for multi-model experiments. To maximize the effectiveness of this adaptation, a Knowledge Distillation approach is used, where a larger, "teacher" model is utilized to generate high-quality synthetic datasets [10]. These datasets provide the compact model with dense, high-quality examples of complex reasoning chains that are often absent from general-purpose pre-training data. By fine-tuning on this synthetic, distilled intelligence, localized models can achieve high reliability in structured tasks without the prohibitive computational costs of full-parameter retraining.

- **Post-Training Quantization for Edge Deployment**: After supervised fine-tuning, model weights are stored in high-precision formats (typically BF16 or FP16) that are well-suited for gradient-based training but unnecessarily expensive for inference. Post-training quantization (PTQ) reduces the numerical precision of weights and, optionally, activations to lower-bit formats such as INT8 or INT4, yielding a proportional reduction in memory footprint and a corresponding increase in throughput with only a small accuracy penalty [11]. The quantization process requires a calibration pass over a small representative dataset to compute per-layer scaling factors that minimize the rounding error introduced by the reduced precision. NVIDIA FP4 (NVFP4) is a 4-bit floating-point format whose arithmetic is natively accelerated by Blackwell Tensor Cores, making it the target quantization format for deployment on Blackwell-class hardware. Quantization is applied using the NVIDIA ModelOpt library [12], which provides the calibration pipeline, the NVFP4 quantization configuration, and the HuggingFace-compatible checkpoint export step. The resulting quantized checkpoint retains the adapted weights from fine-tuning in a memory-efficient representation that can be loaded directly by an inference server, decoupling the training environment from the serving environment.

- **Inference Serving: vLLM**: Once quantized, a fine-tuned model must be served with sufficient throughput and low enough latency to meet the real-time requirements of an interactive AR assistant. vLLM [13] is an open-source inference engine that addresses this requirement through PagedAttention, a memory management technique that eliminates fragmentation in the key-value cache and allows the GPU memory reserved for in-flight requests to be reclaimed and reused at a fine granularity. vLLM exposes an OpenAI-compatible REST API, enabling any client that can issue HTTP requests to submit prompts and receive token-level streaming responses without awareness of the underlying serving infrastructure. In this project, vLLM serves two roles: it provides the inference endpoint for the teacher model used during synthetic dataset generation, and it hosts the fine-tuned, NVFP4-quantized production model that receives queries from the inference engine at runtime. The same API interface is used in both cases, allowing the generation pipeline and the deployment pipeline to share client code.

- **Evaluation of Structured Prediction Systems**: Assessing the out-

put quality of a tool-calling agent requires metrics that go beyond binary accuracy, because a response that retrieves most but not all of the correct entities is qualitatively different from a completely wrong response. The standard information-retrieval metrics Precision, Recall, and F1-score can be adapted to this setting by computing true positives, false positives, and false negatives over multisets of predicted and expected entities, thereby giving partial credit to responses that are partially correct. Jaccard similarity, defined as the cardinality of the set intersection divided by the cardinality of the set union, provides a symmetric alternative bounded in $[0, 1]$ that is commonly used alongside F1 when the positive class dominates. Applying these metrics to tool-call evaluation requires a semantic resolution step: a tool call that references a net name must be expanded to the full list of components on that net, and a call parameterized by component type must be expanded to the list of all matching component IDs, so that two logically equivalent but syntactically different responses receive the same score regardless of which tool name the model chose to invoke.

- **Speech-to-Text in Industrial Environments — Early Approaches**: The first step toward hands-free interaction is converting the technician's spoken query into text. In a workshop environment this is substantially harder than general-purpose transcription: background equipment noise, ventilation, and overlapping tool sounds degrade models trained primarily on clean speech; the working vocabulary contains dense alphanumeric identifiers (R12, C16, U5) that are acoustically ambiguous and statistically rare in standard training corpora; and in the target deployment the system must handle both Italian and English queries without requiring an explicit language switch.

  The earliest and most accessible approach to speech recognition in Python is the `SpeechRecognition` library, which exposes a unified interface over several backends. Its default backend, `recognize_google()`, proxies audio to Google's Web Speech API and returns a transcript. The library requires no local model and no GPU, making it the natural first choice for rapid prototyping. It produces acceptable results on clean, close-microphone English speech, but it has two fundamental limitations for the target use case: it requires an active internet connection, which conflicts with the offline-first deployment requirement; and it has no mechanism for handling technical jargon or alphanumeric identifiers, so designators such as "R12" are frequently transcribed as phonetically plausible but semantically incorrect strings ("are twelve", "our 12", and similar variants). On-premise alternatives at this weight class include models such as Moonshine [14], a compact ASR model designed to run in real time on edge devices without a discrete GPU. Moonshine achieves low latency through architectural simplification, but its accuracy on noisy industrial audio and on non-English speech is limited, reflecting the trade-off inherent in edge-first design: inference cost is minimised at the expense of acoustic coverage.

- **Speech-to-Text in Industrial Environments — Large Pretrained Models**: The limitations of lightweight approaches motivated a shift to large pretrained ASR models. OpenAI Whisper [15] is trained on 680 000 hours of multilingual supervised audio, giving it broad coverage of accents, languages, and acoustic conditions far beyond what smaller models achieve. Earlier checkpoints (medium, large-v2) improved transcription quality substantially over the prototype-grade tools and handled Italian with no additional configuration, owing to the model's multilingual pretraining. However, these checkpoints required a discrete GPU for real-time performance and carried noticeable inference latency, which was acceptable during evaluation but sub-optimal for an interactive AR assistant where responses are expected within one to two seconds of the technician speaking.

  The introduction of the Large-v3-Turbo checkpoint addressed the latency concern: it is a distilled variant of Large-v3 that retains the multilingual accuracy and the voice activity detection capabilities of the full model while reducing inference time substantially. The `faster-whisper` library further accelerates this checkpoint through a CTranslate2 reimplementation of the Whisper compute graph, delivering additional speedups over the original PyTorch code without any accuracy loss.

  NVIDIA Canary-1B-v2 [16] represents a complementary design point. Rather than scaling a single general-purpose architecture to 1.5 billion parameters, Canary is built around a FastConformer encoder whose convolutional front-end is specifically engineered for high-noise acoustic environments: short local convolutions capture the stable spectral patterns that persist under broadband background noise, where attention-heavy architectures trained on clean speech tend to misfire. The model is accessed through the NeMo framework [17], NVIDIA's open-source toolkit for building and deploying large-scale speech and language models, which provides the transcription API, voice activity detection utilities, and the multilingual language configuration. Trained on 85 000 hours of multilingual audio covering 25 European languages including Italian, Canary-1B-v2 is reported to be up to ten times faster than Whisper Large-v3 at equivalent or superior accuracy on industrial acoustic benchmarks. This combination of noise robustness, multilingual coverage, fast inference, and full offline operation made it the primary candidate for workshop deployment, alongside Whisper Large-v3-Turbo as a strong general-purpose alternative. The quantitative evaluation of both models under clean, noisy, and technical-vocabulary conditions is reported in the Results chapter.

- **Text-to-Speech for Hands-Free Response Delivery**: Completing the voice interaction loop requires not only converting speech to text but also converting the assistant's textual answer back into speech. In an AR deployment where the technician's hands are occupied and visual attention is directed at the board, having the assistant's response read aloud allows the technician to receive information without shifting their

gaze to a screen. The speech synthesis component must satisfy three constraints in this environment: it must produce natural-sounding speech at low latency so that the spoken answer begins within a perceptible time after the text answer starts appearing; it must support both Italian and English, matching the language of the response without manual configuration; and it should require no per-query cost or proprietary API key to remain compatible with the offline-capable, on-premise deployment model.

Microsoft Edge TTS, accessed through the `edge-tts` Python library, was selected as the synthesis backend. It provides access to Microsoft's Azure Neural Speech voices without requiring an API account. The neural voice models produce high-quality, naturally-paced speech with prosody substantially more natural than concatenative or parametric synthesisers. The library exposes over 400 voices across more than 100 languages and locales, including dedicated Italian voices (`it-IT-ElsaNeural`) and English voices (`en-GB-SoniaNeural`, `en-US-JennyNeural`). In the ARBoard system, the inference engine detects the language of the assistant's conversational response and selects the matching voice automatically, so the spoken reply is always in the same language as the question—Italian if the technician asked in Italian, English otherwise. Synthesis latency is low enough to permit real-time delivery: the audio begins playing while the text is still streaming into the chat panel, giving the technician simultaneous visual and auditory access to the response.

Taken together, these foundations illustrate how modern industrial assistants can bridge static documentation and contextual, real-time support on the factory floor. Standardized PCB data and computer vision reconstruct the spatial and logical structure of a board, while agentic large language models provide a controllable reasoning core that can reliably call tools over this representation. Parameter-efficient fine-tuning and post-training quantization adapt compact models to specialized tasks and deploy them within the memory envelope of edge hardware, while a high-throughput inference server keeps response latency within the real-time budget. Robust, industrial-grade speech-to-text and speech-synthesis complete the stack by enabling fully hands-free interaction, allowing technicians to speak a question, receive a spoken and textual answer, and watch the relevant components highlighted on the board—all without releasing the tools in their hands. In the following chapters, these principles are instantiated and specialized in the design and implementation of the ARBoard "Intelligent Backbone" system.

III. Methodology

This chapter describes the design and implementation of the two primary software contributions of this thesis: the Perception Layer, a template-matching-based text extraction pipeline that reconstructs component spatial data from PCB documentation, and the Reasoning Layer, a local AI assistant that maps natural language and voice commands to deterministic tool calls over a structured board database. Both layers share a common SQLite database as their integration point; the Perception Layer populates it with spatial component data, and the Reasoning Layer queries it at inference time to ground the model's context. The following sections describe each layer in full, including the key algorithmic decisions and their justifications.

III.1 System Architecture Overview

The full system architecture is structured around a central relational database. Rather than coupling the visual recognition pipeline directly to the AI assistant, both subsystems treat the database as the single source of truth. This decoupling allows them to be developed, tested, and deployed independently while sharing a consistent data schema.

The flow at a high level is as follows. When a new PCB enters the workflow and no structured design file (such as IPC-2581) is available, the Perception Layer is invoked. It accepts a PDF schematic or placement drawing as input, extracts the component reference designators and their pixel coordinates from the document, and writes the results to the database. From that point on, the Reasoning Layer operates as if it were given structured data: it loads the component list and netlist from the database at inference time, injects them directly into the language model's context, and processes natural language or voice queries to produce structured tool calls that drive the AR viewer. The remainder of this chapter addresses each layer in the order in which they are encountered in the data flow.

Both layers are deployed on a central backend server that hosts the database, runs the Perception Layer pipeline, and serves the AI inference engine. The AR smart glasses and the desktop application are thin clients: they do not hold any board data locally and access the entire system exclusively through REST API calls to this server. Component positions, netlist data, AI queries, voice transcription, and tool call results are all exchanged over this API. This server-centric design ensures that the board data remains on controlled infrastructure and that the client devices require no local storage or compute beyond rendering the AR overlay.

III.2 Part I: Perception Layer — Label Extraction Pipeline

The Perception Layer addresses the scenario in which a technician is working with a PCB for which no machine-readable design file exists. The only available documentation is a human-readable PDF, either a placement drawing that shows component positions on the board outline, or a schematic that shows logical connectivity. The goal of the pipeline is to locate every component reference designator in the document and record both the label text and its position in image coordinates. The pipeline consists of six sequential stages: template acquisition, document ingestion and preprocessing, candidate detection, character classification, multi-rotation aggregation, spatial grouping, and a final suppression pass.

III.2.1 Template Acquisition

The character classification stage of the pipeline is built on template matching rather than a learned OCR model. This design decision was made deliberately: trained OCR models (including Tesseract [5]) tend to fail on the dense, small, and often rotated text found in PCB placement drawings because their training data is dominated by natural document typography. Template matching, in contrast, is exact by construction: a reference glyph is matched against every candidate region in the image, so performance is bounded only by the quality of the reference images.

Template acquisition is therefore the setup step that makes the entire pipeline viable for a new document type. An interactive graphical tool was implemented for this purpose. Rather than requiring full annotation of an entire document, the technician opens a representative page of the target PDF and uses the tool to click on individual character samples directly from the rendered page. Each selected region is cropped, binarized, and saved as a grayscale PNG image. Because the same font appears consistently throughout all pages of a given document type, a single annotation session of a few dozen glyphs provides sufficient coverage for the full library: uppercase letters A–Z and digits 0–9. The pipeline maintains two separate template sets, one for topographic placement drawings and one for electrical schematics, since these document types frequently use different typefaces and character weights. As an alternative, templates can also be generated programmatically from any TrueType font file, which is useful when the font used in a document can be identified.

III.2.2 Document Ingestion and Preprocessing

PDF documents are rendered to raster images using PyMuPDF [18] at a configurable zoom level. Two distinct zoom factors are used in the pipeline: a lower detection zoom (default 6×) that controls the resolution at which the binarized image is processed for candidate extraction, and a higher capture

zoom (default 24×) that is used to render the region around each candidate at higher resolution for template matching. This dual-zoom design keeps candidate detection computationally tractable on full pages while ensuring that the matching step has sufficient detail to discriminate between similar glyphs.

After rasterization, the image undergoes a preprocessing sequence implemented using OpenCV [19]. The image is first converted to grayscale and then binarized using Otsu's global thresholding method, which selects the threshold that minimizes intra-class variance between the foreground and background pixel distributions. An automatic inversion step follows: if the fraction of white pixels exceeds 50% of the total, the binary image is inverted so that text (foreground) is always represented as white pixels on a black background. This convention is maintained uniformly throughout the rest of the pipeline; an example of the resulting binary schematic page is shown in Figure III.1.

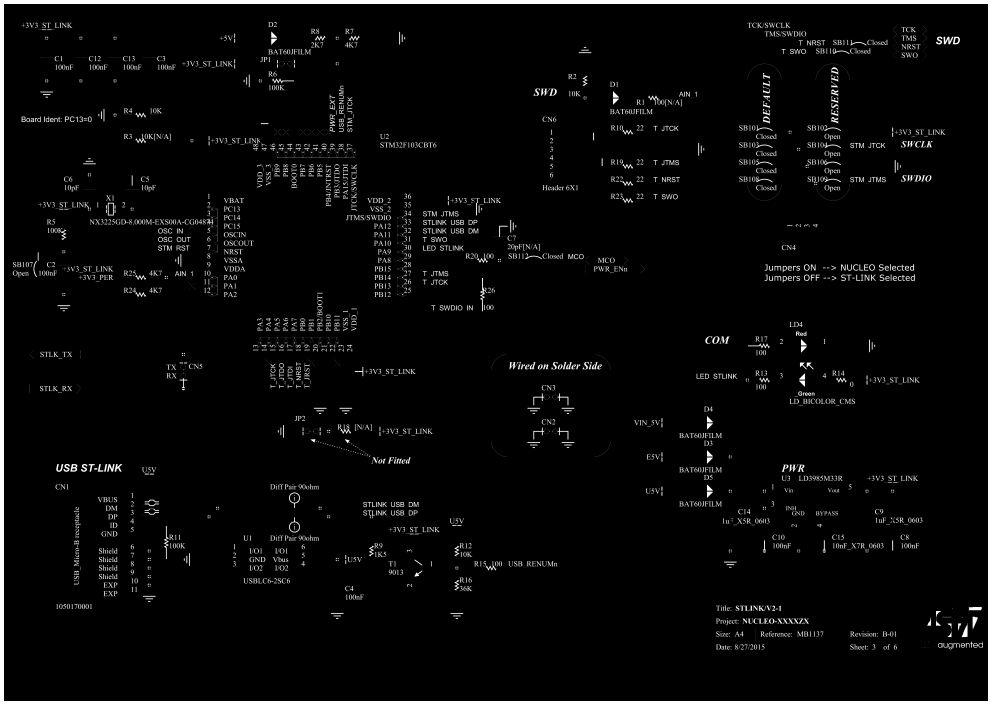


Figure III.1: Binarized schematic page after preprocessing.

A critical preprocessing step for PCB drawings is the removal of long structural lines. Both schematics and placement drawings contain extensive horizontal and vertical lines representing circuit traces, board outlines, and grid rules. These structures fragment and corrupt the contour analysis in the subsequent detection stage. Long lines are detected using morphological opening with structuring elements that are elongated in the horizontal and vertical directions, respectively, and the detected line masks are subtracted from the binary image via bitwise operations. The minimum line length parameter is set dynamically as a function of the expected maximum character dimension to avoid removing strokes that are part of tall characters such as capital I or digit 1.

III.2.3 Candidate Detection

Candidate regions are identified by running connected component analysis on the cleaned binary image via `cv2.findContours` with the `RETR_TREE` hierarchy retrieval mode. Each contour is fitted with an axis-aligned bounding box, and those boxes are filtered by minimum and maximum width and height. Rather than using fixed pixel thresholds, the bounds are derived dynamically from the loaded template library and the zoom ratio between the detection and capture zoom levels, so the same code accommodates both low-resolution and high-resolution documents without reconfiguration.

An additional strict minimum on the longest bounding box dimension is enforced: candidates whose longest side falls below 90% of the shortest expected template dimension are discarded. An exception is made for extremely narrow candidates with a height-to-width aspect ratio above 2.0, since narrow characters such as the digit 1 legitimately have small widths. Figure III.2 shows the candidate bounding boxes found on a schematic page at this stage, before character classification and label detection.

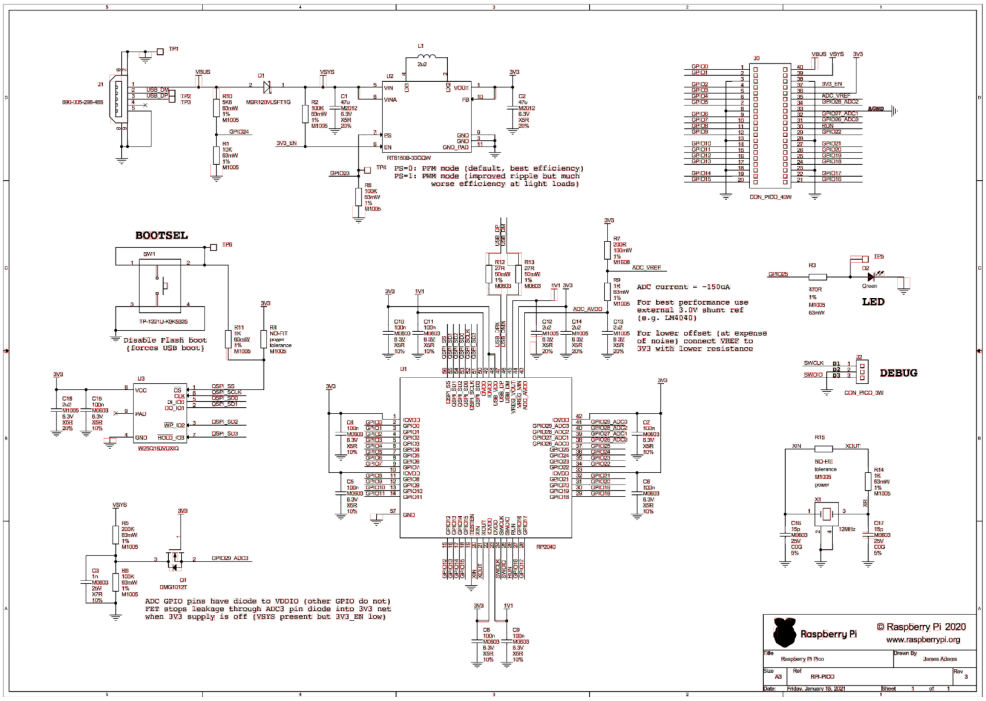


Figure III.2: Candidate regions identified on a schematic page after preprocessing and contour-based detection.

III.2.4 Character Classification via Template Matching

For each surviving candidate, a region of interest (ROI) is extracted. When the high-resolution rendering is available, the ROI is taken from the high-res

image by scaling the candidate bounding box coordinates by the ratio between the capture zoom and the detection zoom, with a small pixel padding on each side. This ensures that the pixel content used for matching is sharp.

The classification step tests the candidate ROI against every template in the loaded library. Before computing any similarity score, a cascade of geometric gates is applied to reject implausible matches cheaply and to prevent numerically strong but semantically wrong correlations:

- **Scale gate.** The ratio of the candidate height to the template height is computed as the observed scale factor. If no reference scale is known, matches with a scale ratio below 0.2 or above 2.0 are rejected. If an expected scale is provided, a tolerance of $\pm 50\%$ is enforced, accommodating stroke-width variation in high-contrast renders.
- **Width validity gate.** The template is rescaled in height to match the candidate, and the resulting scaled template width is compared to the candidate width. If the relative width difference exceeds 50% of the larger of the two widths, the match is rejected. This gate prevents wide characters such as M or W from matching narrow characters such as I.
- **Pixel mass gate.** The number of foreground pixels in the binarized candidate is compared to the expected number derived by scaling the template's pixel mass by the square of the scale factor. Candidates with a mass ratio below 0.3 or above 2.0 are rejected. Solder pads and other compact circular structures frequently pass the size and width gates but have a distinctly different mass-to-area ratio compared to the letter O or digit 0.
- **Topological gate (Euler number).** The hole count of both the candidate and the template is computed using contour hierarchy analysis. The Euler number of a binary object is the number of connected components minus the number of holes; for a single isolated character, it is equivalent to $1 -$ (hole count). Characters such as C, S, and 1 have zero internal holes, while characters such as O, D, and 0 have one hole, and B and 8 have two. Any discrepancy between the template hole count and the candidate hole count is treated as a rejection signal if the candidate has more holes than the template, since this pattern is characteristic of a pad or ring structure being misclassified as an open character. This gate is particularly effective in distinguishing the letter C from circular solder pad annuli, which is otherwise one of the highest-false-positive pairings on populated placement drawings.

The *candidate reduction factor* for a given page is $\rho = N_{\text{cand}}/N_{\text{survive}}$, where N_{cand} is the number of raw contour-based candidates and N_{survive} is the number that pass all four gates and the correlation threshold; ρ is thus a measured result per page, not a fixed parameter.

For candidates that survive all four gates, normalized cross-correlation is computed using `cv2.TM_CCOEFF_NORMED`. The template is resized to match the

candidate dimensions, both images are placed on a common canvas, and the maximum correlation value is taken as the match score. The character with the highest score above a threshold of 0.5 is assigned to the candidate.

III.2.5 Multi-Rotation Strategy

Component labels on PCB drawings are not always oriented horizontally. On placement drawings in particular, labels are frequently printed at 90° to save space, and on schematics diagonal or rotated text is common near off-page connector symbols. The pipeline handles this by processing three independent views of each page: the original 0° image, a 90° clockwise rotation, and a 45° counter-clockwise rotation. Each view undergoes the full preprocessing, candidate detection, and classification sequence independently. Within the matcher itself, the candidate ROI is additionally tested at 0°, 90°, 180°, and 270° against each template; a bias is applied so that the 0° rotation is preferred whenever its score is within 5% of the global best across all tested angles. This bias prevents symmetric characters (such as X or O) from being assigned arbitrary non-zero orientations due to minor numerical differences.

After processing all three rotations, the bounding boxes of each detected label are transformed back to the coordinate system of the original 0° image using the inverse affine matrix of the rotation applied, and all detections are pooled into a single list. A high-resolution rendering of each page is used only for the 0° task, since rotated versions are derived in software from the already-rendered base image.

III.2.6 Spatial Grouping and Label Reconstruction

Individual character detections must be assembled into complete reference designators such as R12 or SW3. This grouping is performed in three ordered passes over the set of accepted character detections.

The first pass builds a horizontal adjacency graph. Two characters are linked if their bounding boxes share vertical overlap of at least 40% of the shorter character's height, their rotations are compatible (two characters both detected at 90° or 270° are not linked horizontally), and the horizontal gap between them does not exceed a dynamically computed threshold of $\max(1.5 \cdot \bar{w},\ 0.6 \cdot \bar{h})$, where $\bar{w}$ and $\bar{h}$ are the average width and height of the pair. The use of the height as a fallback accommodates narrow characters such as the digit 1, whose width-based gap limit would otherwise be too tight. Connected components are found by depth-first search on this adjacency graph, and each component of two or more characters is assembled into a horizontal label by sorting left to right.

The second pass operates on characters not consumed by the first pass and applies the equivalent logic in the vertical direction, linking characters with horizontal overlap and vertical gaps within a height-proportional threshold. Characters matched at 270° (which correspond to text running upward on the page) are sorted in reverse vertical order before concatenation.

The third pass collects any remaining ungrouped characters as singleton labels. All produced labels are then classified by a regular expression matching the standard component reference designator format (one or two prefix letters followed by one to three digits). Sequences of only letters or only digits are classified separately. After the final suppression step described below, the surviving labels and their positions are written to the database and drive the AR viewer; Figure III.3 shows the ARBoard schematic panel, which consumes this data to render an interactive component list and schematic with selection and highlighting.

III.2.7 Non-Maximum Suppression and Output

Because the three rotation passes can produce overlapping detections for the same physical label on the page, a final suppression step is applied to the pooled result list. Pairs of labels are compared by computing the intersection-over-union (IoU) of their bounding boxes. When two labels overlap with an IoU above 0.2, only the one with the higher priority is retained. Priority is determined first by label type (component reference designators are preferred over generic text strings), then by average match score. The surviving labels constitute the final output of the pipeline: a list of reference designators, each with its text, its bounding polygon in image coordinates, and its center position. These coordinates are then stored in the database together with the page number, providing the spatial grounding required by the AR overlay system and enabling the schematic panel shown in Figure III.3.

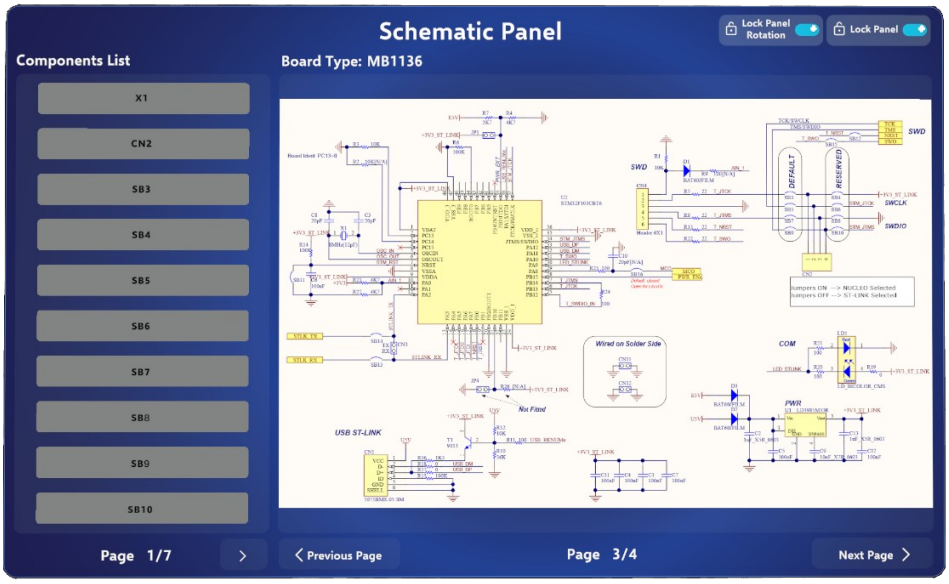


Figure III.3: ARBoard schematic panel populated from label-extraction output.

III.2.8 Cross-Document Component Resolution

The label extraction pipeline operates on one document page at a time and produces an independent set of detected reference designators for each page. Building a complete, spatially grounded component record for the AR overlay requires a further step that reconciles detections across the full document set.

Two structural properties of real PCB documentation make this reconciliation non-trivial. First, a board schematic is typically divided across multiple pages by functional block: a power management section, a communication interface section, and a processor section may each occupy separate schematic pages, so no single page contains all components. Second, placement drawings are divided by board side: one document covers the top side of the PCB and a second covers the bottom, so a component's physical coordinates can only be determined once it is known which placement document it appears in. Running the pipeline on a single document in isolation therefore yields an incomplete picture of either the logical connectivity or the physical location.

The cross-document resolution step addresses this by matching detected labels across document types after all pages have been processed. For each reference designator extracted from any schematic page, the resolver searches the union of labels extracted from all placement documents for an identical identifier. When a match is found, the component record is enriched with the physical pixel coordinates from the placement drawing and a board-side label derived from which placement document contained the match: top or bottom. When a reference designator appears on a schematic but is absent from all placement documents — for example a logical power symbol with no physical footprint — it is retained in the database with its schematic-page coordinates and flagged accordingly. Components present in the placement drawings but absent from all schematic pages are stored with their physical coordinates only, as their logical connectivity cannot be inferred from the available documentation.

The output of this step is a unified component table in which each entry carries both the logical identity of the component and its physical location on the board: pixel coordinates, page number, and board side. This enriched record is what the AR overlay consumes to project the correct highlight onto the physical board.

III.3 Part II: Reasoning Layer — AI Assistant Intelligent Backbone

The Reasoning Layer provides the interactive intelligence of the ARBoard system. A technician working at a bench should be able to ask a question or issue a command in natural language, including spoken input, and receive a response that combines a plain-language explanation with a structured action directed at the AR viewer. The core engineering challenge is converting the inherently noisy and ambiguous output of a speech-to-text engine into a deterministic, validated software action, using a language model that runs entirely on local hardware with no dependence on external APIs.

The development of the Reasoning Layer proceeded through five interconnected stages executed in the following order. First, a preliminary survey of large frontier models was conducted using the LLMBenchmark framework to assess the out-of-the-box feasibility of the tool-calling task and determine whether direct deployment of a large hosted model or fine-tuning of a smaller local one was the more practical path. Second, a synthetic dataset was generated using a high-reasoning teacher model to produce domain-specific training examples covering the full range of user intents. Third, the LLMBenchmark framework was used again to evaluate the base performance of five candidate 8-billion-parameter architectures before any training, establishing a per-architecture zero-shot baseline. Fourth, supervised fine-tuning was applied across a sweep of model architectures and LoRA hyperparameters, with all configurations evaluated against a held-out test set. Fifth, the best-performing STT backend and fine-tuned model were integrated into the deployment server and exposed to the ARBoard application via a REST API. Each stage is described in the order in which it was executed.

III.3.1 Database Schema and Context Injection

The database is implemented in SQLite [8] and serves as the single integration point between the Perception Layer and the Reasoning Layer. It contains four primary tables: `component`, which stores each component's reference designator, footprint type, and spatial coordinates; `net`, which stores each named electrical net; `pad`, which stores the per-pin connections between components and nets; and `pdf_user_manual`, which stores the binary content of the board's user manual PDF as a blob. This schema mirrors the logical structure of an IPC-2581 file, so the same database can be populated either automatically from a structured design file or by the Perception Layer working from a legacy PDF.

Before any query is processed, a dedicated endpoint serializes the database contents into two human-readable text blocks: a component list and a netlist. Each component entry pairs its reference designator with its footprint type, and each net is written as a logical sentence listing all of its connected pins. At inference time, a third block is derived from the `pdf_user_manual` table: the

stored PDF binary is read, its pages are extracted using PyMuPDF [18], and the resulting plaintext is included alongside the component list and netlist as additional context. The user manual typically contains functional descriptions, block diagrams, and component purpose information that the component list and netlist alone do not capture, allowing the language model to answer questions about what a component does and not only where it is located or what it connects to.

This context injection approach was chosen over retrieval-augmented generation [7] for two reasons. The board database for a typical PCB falls well within the context window of a modern 8-billion-parameter model (typically 150–200 components and a comparable number of nets), so no retrieval step is necessary. More importantly, full injection provides a completeness guarantee: a missed retrieval would cause the model to silently treat a real component as non-existent, which is unacceptable in a diagnostic assistant. Full injection eliminates that failure mode entirely and makes the grounding deterministic.

III.3.2 Synthetic Dataset Generation

No public dataset exists that covers the combination of PCB domain vocabulary, tool-call JSON structure, and the bilingual (Italian/English) operating environment of the target deployment. A synthetic dataset was therefore generated programmatically using a large high-reasoning teacher model served on the local hardware via vLLM [13].

Board context generation. For each batch of examples, a random subgraph of 150–190 components from the real board database was sampled using a breadth-first expansion seeded at a random component. The connected nets and their participants were assembled into the context string. This sampling strategy prevents the fine-tuned model from memorizing a single fixed layout and forces generalization across different board sections. Nets with the prefix "Unused" were excluded from the context to prevent the model from referencing unconnected pads.

Task categories and diversity. Nine task types were defined and weighted to control the distribution of the generated examples: single-component location queries (weight 20), net tracing queries (15), component-type queries (10), multi-component queries (10), component connectivity queries (10), illumination-verb queries (15), free-form conversational queries (15), STT error queries (8), and out-of-domain queries (1). Within each batch, the generator first ensured coverage of all applicable categories before filling remaining slots by weighted random selection. This two-pass strategy prevented any single category from dominating while maintaining the weighted prior.

Phonetic error coverage. STT error examples were generated by explicitly instructing the teacher model to produce user questions in which a component identifier is replaced by its phonetic transcription, for example "Are Twelve" for R12 or "You Won" for U1. The expected tool call in these examples targets the correctly interpreted component, and the expected assistant answer explicitly acknowledges the correction. These examples serve a dual purpose: they teach

the fine-tuned model to recognize phonetic variants of electronics identifiers, and they provide training signal for the correction acknowledgement behaviour that is also specified in the system prompt at inference time.

Negative constraint examples. A fraction of 20% of each batch consisted of negative examples in which the user requests a component, net, or type that does not exist in the board section. The correct response for these examples is an empty tool call array and a polite acknowledgement that the requested item is not present. This ensures the fine-tuned model learns to refrain from hallucinating components.

Dataset variants. Two qualitatively different dataset variants were generated and used in the fine-tuning sweep, enabling a controlled comparison of dataset complexity effects. The first variant was generated with the full persona matrix enabled: for each example, the teacher model was provided a randomly sampled archetype (one of five: Junior Hardware Engineer, Senior System Architect, PCB Librarian, Hobbyist/Student, Production Floor Technician), a communication style (one of four: Verbose and Polite, Direct and Technical, Brief/Shorthand, Confused/STT-heavy), and an intent category (one of four: Verification, Troubleshooting, General Inquiry, Correction). This matrix defines $5 \times 4 \times 4 = 80$ distinct persona combinations that were sampled uniformly. The second, simpler variant disabled the persona injection and generated all examples in English only, producing a more uniform lexical distribution. Separately, a version with bilingual Italian/English generation was also explored, where the teacher was instructed to generate both the user question and the assistant answer in Italian while keeping component identifiers in English. The results of fine-tuning on these different variants are discussed in the Results chapter.

Post-generation validation. Each generated example was validated before being written to the output file. The assistant response was checked for the presence of the required output separator, and the portion after it was parsed as JSON. Examples that failed either check were discarded.

Limitations of the teacher-model approach. Despite the quality controls above, the teacher-model generation approach carries a structural limitation: the correctness of the training labels is bounded by the reliability of the teacher model. Even a highly capable teacher occasionally makes tool selection errors on ambiguous or long contexts, selects the wrong component identifier, or produces marginally malformed JSON that passes the post-generation check but carries a subtle semantic error. These noisy examples are indistinguishable from correct ones during training; for a narrow 4-class task where the evaluation metric is binary, even a small fraction of mislabelled examples injects low-frequency noise into the gradient that is difficult to diagnose. Additionally, the teacher-model approach incurs non-trivial generation cost: producing and validating 500 examples at 10 parallel threads requires several hours of on-hardware inference, making iterative dataset refinement slow. The deterministic construction approach described in the following subsection was developed in direct response to both limitations.

III.3.3 Deterministic Dataset Construction

Alongside the teacher-model-generated corpus, a second dataset was constructed by a fully deterministic generator that requires no inference calls. Rather than asking a language model to produce question–answer pairs, the generator derives them algebraically from the board database: for each board context (a random subgraph of 150–190 components, sampled identically to the teacher-model procedure), it instantiates a fixed set of question templates and computes the correct tool call directly from the database contents. This approach guarantees that every training example has a provably correct label, eliminates the output-format validation failure rate that affects teacher-model generation, and produces a dataset whose difficulty distribution is under full algorithmic control.

Tool classes and question templates. The generator covers four tool classes that partition the output space of the AR assistant: `highlight_component` (single-component location queries), `highlight_net` (net-tracing queries), `highlight_by_type` (footprint-type queries), and a negative class for requests targeting items not present in the current board context. For each class, between 10 and 15 distinct question templates are defined, covering direct queries ("Where is R12?"), indirect formulations ("Can you show me the Y3 crystal?"), and shorthand variants ("find R12"). The relative sampling weight of each class is controlled by a configurable task-weight vector.

Class imbalance and the `highlight_by_type` ceiling. An important structural asymmetry distinguishes the `highlight_by_type` class from the other three. While component and net queries can be generated for every entity in the board context—scaling linearly with the number of board subgraphs sampled—type-based queries are bounded by the number of distinct footprint types present on the board. With approximately 22 unique component types in the evaluation board and 13 applicable question templates, the total number of unique (question, tool call) pairs for this class is capped at roughly $22 \times 13 = 286$, regardless of how many board contexts are sampled. Deduplication on the (question, tool call) key removes all further copies once the ceiling is reached. The remaining three classes do not face this constraint, because each component reference designator and net name constitutes a distinct entity; unique examples continue to accumulate with additional board contexts.

Balance step. Left unaddressed, this ceiling causes `highlight_by_type` to represent only approximately 4% of the total dataset when the other classes are allowed to grow freely. To correct the imbalance, a post-deduplication balancing step groups all unique examples by their first tool call class and downsamples each group to the cardinality of the smallest group. The target of approximately 250 examples per class was used in practice, yielding a total of 1 000 balanced training examples across the four classes. This target sits comfortably below the `highlight_by_type` ceiling of approximately 286 unique type queries, so the ceiling does not constrain the balance operation at this scale. Upsampling was explicitly rejected to avoid duplicating examples and introducing artificial repetition bias into the gradient; balancing is strictly

a downsampling operation.

Train/test construction. The training set (1 000 examples, seed 42) and the evaluation set (generated independently with a different seed) are produced by separate runs of the same generator. This ensures no board context seen during training can appear in evaluation. The absence of context overlap is verified programmatically by a set-intersection check that raises a hard error if any board context appears in both. The evaluation set used in the in-process fine-tuning harness (Question 3 in the Results chapter) was drawn from a generated pool with seed 99; the harness was executed with 80 evaluation questions, consistent with the 1.25-percentage-point resolution visible in all reported metrics for that evaluation track. The LLMBenchmark evaluation used in Questions 1, 2, and 4 uses a separate 40-question test set generated from the same deterministic generator with an independent seed, ensuring the two evaluation tracks are fully decoupled.

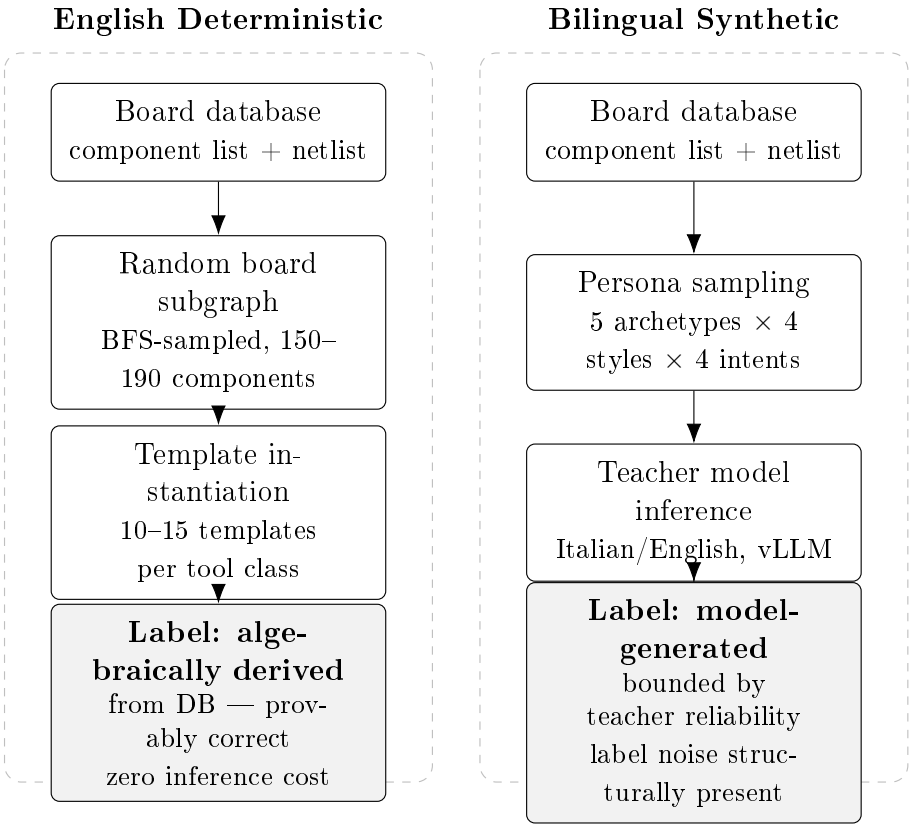


Figure III.4: Comparison of the deterministic and bilingual synthetic dataset generation pipelines.

III.3.4 LLMBenchmark Evaluation Framework

A standalone benchmark framework was developed to systematically evaluate the out-of-the-box tool-calling performance of language models. This framework, referred to as LLMBenchmark, served two distinct roles in the project:

first as a preliminary survey tool to assess task feasibility across a wide range of frontier and cloud-hosted models before any fine-tuning work was committed to, and second as the evaluation harness for the model and inference method comparison that selected the architecture and inference strategy to carry forward into the fine-tuning campaign. Both roles use the same LLMBenchmark infrastructure, the same ground truth set, and the same accuracy metric; they differ in the set of models tested, the inference methods applied, and the purpose of the comparison.

Preliminary model survey. Before committing to a fine-tuning approach, LLMBenchmark was used to determine whether off-the-shelf frontier models could already solve the tool-calling task reliably enough for production use. A total of 17 distinct models were evaluated, spanning parameter counts from approximately 14 billion to 120 billion. Models that could not be hosted locally were accessed via the OpenRouter cloud API; models that fit within the DGX Spark memory budget were served locally through LM Studio. All survey runs used the One-Shot inference strategy on the fixed 40-question evaluation set. The survey confirmed that large models can reach perfect accuracy on the evaluation set, but also revealed two constraints that ruled out a direct deployment approach. First, models above approximately 20 billion parameters require dedicated serving infrastructure whose memory and throughput requirements conflict with the real-time latency budget of the AR interaction loop. Second, models accessed via cloud API introduce an unavoidable external dependency and a per-request cost that is incompatible with the offline-capable, on-premise deployment scenario of the target environment. Together, these constraints motivated the fine-tuning approach: train a domain-adapted 8-billion-parameter model that can be served efficiently on the DGX Spark alongside the rest of the application stack. The numerical results of the survey are presented in the Results chapter.

Ground truth construction. The benchmark operates on a fixed set of questions with known correct tool calls, stored as a JSON file. A dedicated GUI tool was built for ground truth generation. The tool connects to the board database and automatically produces questions across four categories: component location ("Where is X?"), net tracing ("Show me the Y net"), type-based queries ("Highlight all Z components"), and connectivity queries ("What is connected to X?"). For each question, the tool computes the expected tool call from the database and writes the entry to the ground truth file. This produces a deterministic, reproducible test set that is tied to the actual board data.

Model and provider support. LLMBenchmark supports two provider backends. The local backend connects to any OpenAI-compatible endpoint such as LM Studio or a vLLM server running on the local network. The cloud backend connects to the OpenRouter API, which was used to evaluate large frontier models including GPT-OSS-120B that could not be hosted locally. Rate limiting is applied automatically to respect the provider's request budget.

Inference methods. A central design question for the tool-calling system was whether to produce the conversational answer and the tool call in a single model generation, or to separate them across multiple calls. LLMBenchmark imple-

ments three distinct inference strategies to measure the performance trade-off of each:

The *One-Shot* method sends a single prompt containing the full board context, the tool definitions, and the question. The model is expected to produce both a plain-text explanation and a JSON tool call in one response. The tool call is extracted from the response by scanning for JSON arrays.

The *Single-Call* method uses a structured output format in which the model is instructed to respond with a JSON object containing both the conversational answer and the tool call as named fields. The answer and tool call are parsed from this object directly.

The *Two-Stage* method (also called the inverted pipeline) separates the generation into two sequential calls. The first call provides only the component and net names, not the full context, and asks the model to decide whether a tool is needed and if so which one. The second call provides the full board context, the question, and the tool decision from the first call, and asks the model to generate a natural-language explanation consistent with that decision. This strategy is motivated by the observation that tool selection is a simpler classification task that does not require the full context, while the conversational answer requires factual grounding.

Each method is evaluated on the same ground truth set, with per-question accuracy and response time recorded. A summary CSV accumulates results across all model and method combinations.

Model and inference method comparison protocol. The second use of LLMBenchmark — the model and inference method comparison described as Question 2 in Section IV.2.4 of the Results chapter — was executed by a dedicated orchestration script. The protocol proceeds in two passes to limit total evaluation time while still covering all three inference methods for the most capable architectures. In the first pass, all candidate architectures are evaluated with the One-Shot method on the 40-question generated test set. One-Shot is used as the ranking criterion because it is the simplest method to execute and produces a reliable ordering of raw tool-calling capability before inference-strategy differences are introduced. In the second pass, the three architectures that achieved the highest One-Shot accuracy are evaluated additionally with Single-Call and Two-Stage, producing a complete three-method profile for the strongest performers. This two-pass structure means that the lower-performing architectures contribute One-Shot results only; their Single-Call and Two-Stage performance is marked as not evaluated in the results table (Section IV.2.4). The combined results from both passes directly informed the selection of the fine-tuning target architecture and the production inference strategy.

Accuracy metric. For the benchmark, accuracy is computed by resolving all tool calls to their canonical low-level form before comparison. Net-based tool calls are expanded to the list of components on that net using the database, and type-based tool calls are expanded to the list of matching component IDs. This resolution step ensures that two semantically equivalent tool calls that use different tool names are correctly scored as equivalent, regardless of which

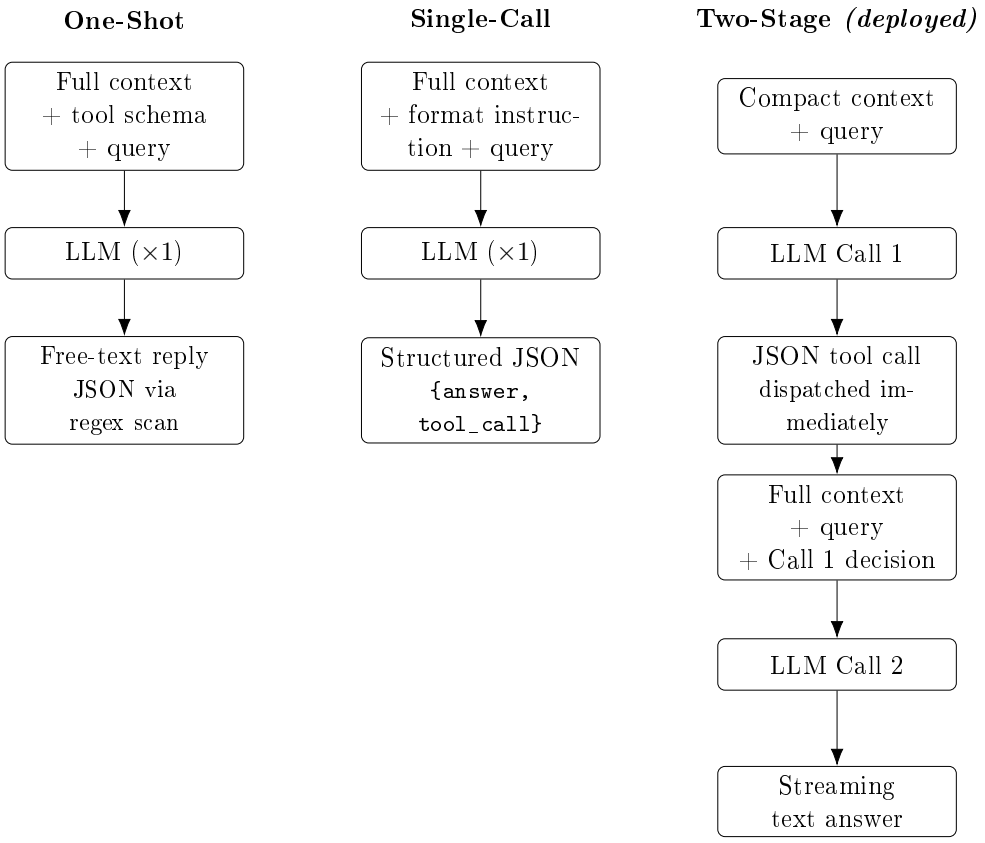


Figure III.5: Comparison of the three inference strategies evaluated in LLMBenchmark.

tool the model chose to invoke.

Schema normalization. Models occasionally produce tool call objects with field names that differ from the specified schema. A normalization pass corrects the most common hallucinations before accuracy is computed, remapping ambiguous tool names and parameter field aliases to their canonical forms in the schema.

III.3.5 Fine-Tuning Pipeline

Fine-tuning was conducted entirely on a local DGX Spark workstation equipped with an NVIDIA Blackwell GB10 Superchip GPU. The Blackwell architecture natively supports BF16 matrix operations at full throughput without quantization overhead, so all training runs were performed in BF16 precision. The Unsloth framework [9] was used to load models and attach LoRA adapters efficiently on the single-GPU setup.

Model sweep. Six model architectures in the 7–9 billion parameter range were included in the fine-tuning sweep: Qwen3-8B [20], Mistral-7B-Instruct-v0.3 [21], Meta-Llama-3.1-8B-Instruct [22], DeepSeek-R1-Distill-Llama-8B [23], Gemma-2-9B-IT [24], and Phi-3.5-mini-instruct [25]. This se-

lection covers five distinct model families (Qwen, Mistral, Llama, Gemma, Phi) and spans standard instruction-tuned, reasoning-distilled, and small-language-model pretraining objectives, enabling a broad architectural comparison within the hardware envelope of local deployment. A seventh architecture, Qwen2.5-7B-Instruct [26], was included in the model and inference method comparison (the baseline LLMBenchmark sweep) but was not included in the fine-tuning sweep: it was added to the candidate set as a late comparison point after the fine-tuning sweep configuration had been finalized, and adding it to the training pipeline would have exceeded the available compute budget without changing the conclusion of the architectural comparison.

LoRA adapter configuration. LoRA adapter matrices were injected into all seven linear projection layers of each transformer block: the four attention projections (query, key, value, and output) and the three feed-forward projections (gate, up, and down) [3]. Including the MLP projections alongside the attention projections provides more adaptation capacity than attention-only LoRA, which was found to be particularly important for the structured output format. Three rank values were swept: $r \in \{16, 32, 64\}$, with the scaling factor set to $\alpha = 2r$ in all runs. The dropout rate was set to zero and no bias terms were added.

Tool-call-only loss masking. A custom data collator was implemented to apply label masking at training time. Each token sequence in the batch is scanned for the last occurrence of the output separator token. All label positions up to and including that separator are masked out, which causes the cross-entropy loss function to ignore them. Loss is therefore computed exclusively over the JSON tool call tokens that follow the separator. This masking is essential because the conversational text before the separator is semantically unconstrained: penalising the model for producing a slightly different phrasing of a correct answer would introduce noise into the gradient. If a training example does not contain the separator, the entire sequence is masked out and contributes nothing to the loss.

Learning rate selection. The learning rate was determined through a short exploratory sweep conducted on a single model–rank combination before committing to the full sweep. The fine-tuning script exposes the learning rate as a configurable argument (default 2×10^{-4}), and three candidates were evaluated. An initial candidate of 2×10^{-4} caused training loss to collapse from approximately 0.12 to near zero within the first 20% of the first epoch, indicating that the gradient steps were overshooting the narrow loss surface of this 4-class task. A rate of 4×10^{-4} produced the same behaviour. Reducing to 5×10^{-5} yielded a smoother, monotonically decreasing loss curve while still converging comfortably within a single epoch; this rate was passed explicitly via a configuration argument and applied uniformly across all model–rank combinations in the final sweep.

Epoch count selection. Two epochs were initially configured as the training budget (the default). Monitoring training loss revealed that, for the majority of model architectures, loss reached near-zero by approximately step 12 out of 31 total steps in the first epoch, equivalent to roughly 40% of the training run. With 1 000 training examples, an effective batch size of 32, and a single

training epoch, there are $\lceil 1000/32 \rceil = 32$ gradient steps — the model had converged well before the end of the first epoch in every tested architecture except DeepSeek-R1-Distill-Llama-8B. Continuing beyond convergence provides no useful gradient signal and introduces unnecessary risk of degrading the base model's general capabilities. The epoch count was therefore reduced to one, which proved sufficient for full convergence across all tested architectures. Single-epoch training also keeps the total wall-clock time per model–rank run to approximately 10 minutes on the DGX Spark hardware, allowing the full sweep to complete within the available compute budget.

Training configuration. Training was run for one epoch using the TRL supervised fine-tuning trainer. The per-device batch size was 16 with gradient accumulation over 2 steps, yielding an effective batch size of 32. The learning rate was set to 5×10^{-5} with the AdamW-8bit optimizer, and gradient checkpointing was enabled to reduce activation memory. Sequence packing was explicitly disabled; enabling it was found during development to cause the loss to collapse to zero from the first training step due to cross-example token boundary corruption in the masking logic. Each run was tracked with Weights & Biases, with a run name encoding the model family, LoRA rank, and timestamp for reproducibility. A manifest file was saved alongside each adapter checkpoint recording the full hyperparameter configuration.

Post-training NVFP4 export. After training, the pipeline generates an optional NVFP4 quantized checkpoint alongside the BF16 adapter for each model–rank combination. The merged weights (base model plus LoRA adapter) are quantized using the NVIDIA ModelOpt library [12] with its default FP4 configuration and a calibration set of 128 examples drawn from the training data. The quantized checkpoint is exported in HuggingFace format and is directly loadable by vLLM. In practice, quantization was not required for this deployment: the DGX Spark workstation provides 128 GB of unified memory, which is more than sufficient to serve an 8–9 billion parameter model in BF16 precision without compression, and avoiding quantization preserves full model accuracy without any performance trade-off. The deployment therefore loads the BF16 merged checkpoint directly into vLLM. The NVFP4 export pipeline was nonetheless implemented and kept ready as a contingency for memory-constrained environments where BF16 serving would exceed the available VRAM budget.

Evaluation. For each model–rank combination, a base evaluation was performed on the held-out test set before fine-tuning, and then repeated after fine-tuning with each of the three LoRA ranks. This evaluation is *not* performed by the LLMBenchmark framework itself; it is performed by a separate in-process evaluation harness that loads the model locally, runs on the same held-out JSONL used for the test set, and measures the same three metrics (JSON validity, tool selection accuracy, parameter accuracy). The prompt and message format in this evaluation match the training data exactly, so that base and fine-tuned models are compared on identical conditions. The distinction between this in-pipeline evaluation and the LLMBenchmark (40-question, API-based) evaluation is explained in Section IV.2.2 of the Results chapter; that section also clarifies why accuracy figures from the preliminary model sur-

vey (LLMBenchmark) must not be compared directly to the fine-tuning sweep results. Results for all runs are appended to a master CSV file, and a comparison report is generated at the end of the sweep. The entire sweep, from base evaluation through fine-tuning through post-training evaluation for all model and rank combinations, is orchestrated by a single shell script to ensure consistent experimental conditions.

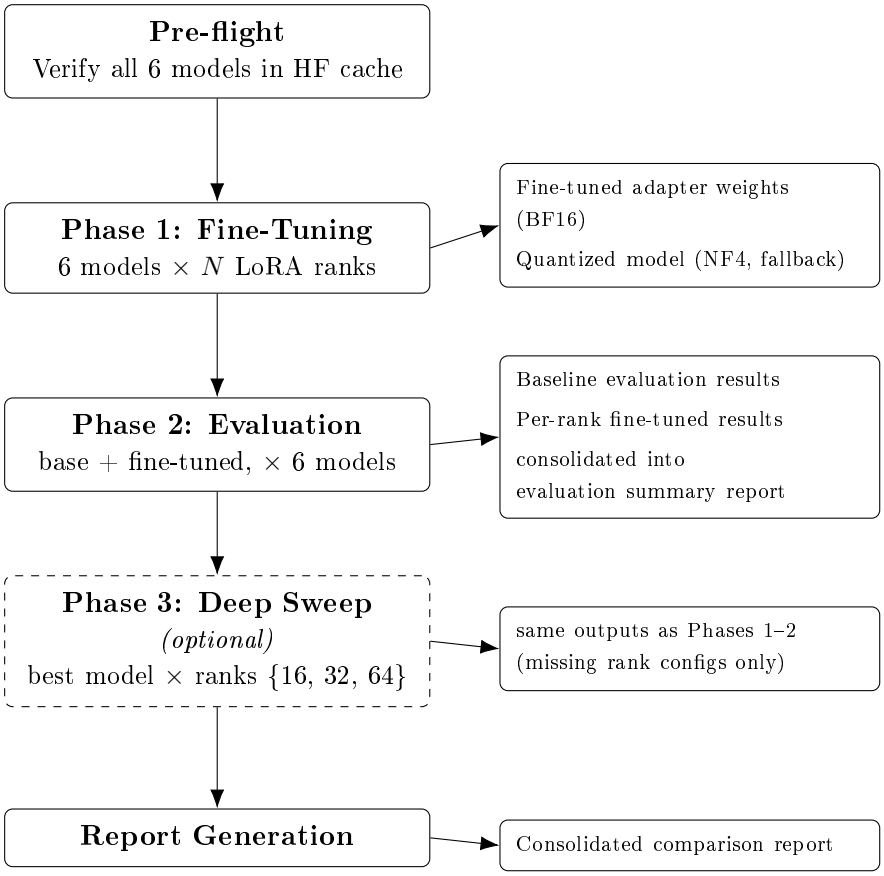


Figure III.6: Structure of the automated benchmarking and fine-tuning pipeline used for model and rank selection.

III.3.6 Speech-to-Text Module

When the system operates in voice mode, the audio captured from the technician's microphone is the entry point to the entire reasoning pipeline. The STT module converts that audio to a plain-text string, which is then passed verbatim as the user query to the language model inference engine. From that point onwards, voice input and typed text input are handled identically: the same system prompt, the same context injection, the same output parsing, and the same tool dispatch logic apply regardless of how the query entered the system. The quality of the transcription is therefore directly load-bearing for end-to-end accuracy: a transcription error on a component identifier (e.g.

mishearing "R12" as "our twelve") propagates into the LLM query and must be recovered by the phonetic correction logic in the system prompt. Choosing an STT backend that minimises WER on electronics shorthand in a noisy environment is consequently as important as the choice of language model.

The STT component is deployed as a standalone FastAPI microservice, selectable via a configurable environment variable, so that it can be swapped or upgraded independently of the language model server. Two candidate backends were selected for final evaluation after an initial exploration described in the Results chapter.

Backend selection criteria. The target deployment environment is an electronics workshop, which presents three constraints that eliminate many off-the-shelf STT solutions. First, the system must operate entirely offline; cloud-proxied solutions such as the `SpeechRecognition` Python library (whose `recognize_google()` backend forwards audio to Google's Web Speech API) were evaluated during early prototyping but ruled out because they introduce an external network dependency and per-request latency that is incompatible with the real-time interaction budget. Lightweight edge models such as Moonshine, which run without a discrete GPU, were also considered but produced insufficient accuracy on noisy audio and on the Italian queries required by the bilingual deployment. Second, the working vocabulary contains a high density of alphanumeric identifiers (R12, C16, U5, SW2) that are acoustically ambiguous and underrepresented in the training data of general-purpose ASR models. Third, the deployment must handle both Italian and English without requiring the user to switch modes, since technicians communicate in Italian but component identifiers and many technical terms follow English conventions.

Whisper Large-v3-Turbo. The first backend uses the faster-whisper [15] inference library running the Whisper Large-v3-Turbo checkpoint. Faster-whisper reimplements the Whisper computation graph using CTranslate2, which yields substantially lower latency and memory consumption than the original PyTorch implementation while preserving model accuracy. The Large-v3-Turbo variant is a distilled version of Large-v3 that retains most of its accuracy at a fraction of the inference cost. The model is loaded at half precision with CUDA acceleration. Voice activity detection (VAD) is applied before transcription, with a minimum silence duration of 700 ms to suppress spurious hallucinations triggered by ambient workshop noise between words. Automatic per-segment language detection is enabled, which handles the bilingual environment transparently without requiring a manual language switch between queries.

NVIDIA Canary-1B-v2. The second backend uses NVIDIA Canary-1B-v2 [16] via the NeMo framework [17]. Canary is a multitask automatic speech recognition and speech translation model trained on 85 000 hours of multilingual audio covering 25 European languages. Its FastConformer encoder is engineered specifically for high-noise robustness: the convolutional front-end captures local acoustic patterns that remain stable under background noise, while the attention layers model long-range dependencies in the speech signal. NVIDIA reports that Canary-1B-v2 is up to ten times faster than Whisper Large-v3 at equivalent or superior accuracy on industrial acoustic bench-

marks, making it particularly attractive for the workshop deployment scenario. Because NeMo's transcription API operates on file paths rather than byte streams, the backend writes the received audio bytes to a temporary file before transcription and deletes it immediately afterwards. The source and target languages are configurable via environment variables; the default configuration is set to English-to-English transcription.

Both backends expose the same HTTP interface: a POST endpoint that accepts an uploaded audio file and returns a JSON object containing the transcription string. A secondary endpoint reports which backend is currently loaded, allowing the inference pipeline to log the STT configuration alongside each request for reproducibility. The best-performing backend, as determined by the evaluation described in the Results chapter, was selected for production deployment.

III.3.7 Text-to-Speech Module

Completing the voice interaction loop requires converting the assistant's textual reply back into speech. Once the language model produces its conversational answer, that text is passed to a text-to-speech (TTS) engine, which synthesizes an audio response played back through the device's speaker. The spoken reply is delivered in parallel with the textual reply: as the answer streams into the chat panel token by token, the synthesised audio begins playing, giving the technician simultaneous visual and auditory access to the same response. This dual-channel delivery is particularly important in the AR context, where the technician's gaze may be directed at the physical board rather than at the text panel.

Model selection. Microsoft Edge TTS, accessed through the `edge-tts` Python library [27], was selected as the synthesis backend. The library provides access to Microsoft's Azure Neural Speech voices without requiring an API account or a per-request cost. The neural voices produce natural, expressive speech with prosody and intonation substantially more natural than rule-based or concatenative synthesisers, which was found during testing to reduce cognitive friction in the AR interaction. The library exposes over 400 voices across more than 100 languages and locales, with dedicated Italian neural voices (`it-IT-ElsaNeural`) and English neural voices (`en-GB-SoniaNeural`, `en-US-JennyNeural`) available out of the box. Synthesis latency is sufficiently low for real-time delivery: audio playback begins within the interaction window after the conversational text has been generated, keeping the spoken response tightly coupled to the visual output.

Automatic language matching. A central requirement for the bilingual deployment is that the spoken reply must be in the same language as the question. The inference engine detects the language of the assistant's conversational response and selects the matching Edge TTS voice automatically before invoking synthesis. If the technician asks a question in Italian, the model answers in Italian and the TTS engine uses the Italian voice; if the question is in English, the English voice is used. No explicit language flag is required from the user.

This behaviour was tested on both Italian and English queries and confirmed to produce correct language assignment in all evaluated cases.

III.3.8 LLM Inference Engine

The inference engine is the coordination hub of the Reasoning Layer. It manages the full lifecycle of a single interaction: receiving the user's input, assembling the board context, querying the language model in two sequential phases, dispatching the resulting tool call to the AR viewer, streaming the conversational answer to the chat panel, and handing that same text to the TTS engine for simultaneous audio playback. The engine connects to the fine-tuned language model served by a vLLM [13] process on the local hardware and to the STT and TTS microservices.

End-to-end interaction flow. A complete interaction follows this sequence. The technician either raises their open palm toward the AR interface—which the system interprets as a voice-activation gesture and triggers audio capture on the headset—or types a question in the text input field of the desktop version of the ARBoard application. If audio was captured, it is forwarded to the STT microservice, which returns a plain-text transcription; if the technician typed, that text is used directly. Before passing the text to the language model, the inference engine applies a set of phonetic normalisation rules that correct the most common STT transcription errors on electronics identifiers (for example, "are twelve" is corrected to "R12" and "you one" to "U1"). The normalised query is then processed by the two-phase inference pipeline described below, which produces a conversational text answer and, when appropriate, a structured tool call. The tool call is dispatched immediately to the AR viewer, which highlights the relevant components or signal net on the board overlay. The conversational text is simultaneously streamed token by token into the AR assistant panel—so the technician sees the answer build up in real time—and passed to the TTS engine, which begins reading the answer aloud in the same language the question was asked. The complete interaction, from palm raise to spoken reply and highlighted components on the board, takes place without the technician releasing the tools in their hands.

Interaction modalities. Two input channels are provided. In the AR version, the technician raises their open palm; the gesture is detected by the AR system and triggers audio capture. The spoken question is recorded and sent to the STT microservice for transcription. In the desktop version of the application, the technician types the question directly into the chat input field of the assistant panel. Both modalities produce the same plain-text query string and share all downstream inference logic identically.

Two-stage inference pipeline. The deployed production inference architecture separates each interaction into two sequential language model calls served by the same fine-tuned model endpoint.

Phase 1 — Tool decision. The first call receives a compact version of the board context (a truncated component list and netlist) together with the normalised query and up to three turns of recent conversation history. Its sole purpose is

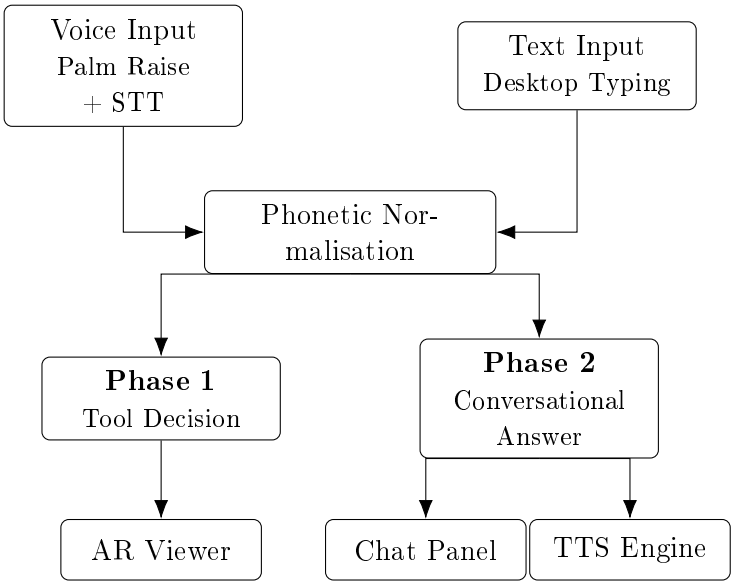


Figure III.7: End-to-end interaction flow of the Reasoning Layer, from multimodal input through two-phase inference to AR, chat, and TTS outputs.

to decide whether a visual tool is needed and, if so, which one and with what parameters. The model is instructed to respond with a single JSON object of the form `{"needs_tool": true/false, "tool": "...", "parameters": {...}}`. This call is fast—it produces only a short JSON response and requires no reasoning tokens—and its result is dispatched to the AR viewer immediately, before Phase 2 has completed. This means that the highlighted components appear on the board while the conversational answer is still being generated, keeping the visual feedback low-latency.

Phase 2 — Conversational answer. The second call receives the full board context: the complete component list, the full netlist, and the extracted text of the user manual PDF. It also receives the Phase 1 tool decision and up to five turns of conversation history, enabling the model to resolve implicit references across turns (such as "now show me its connections" referring to a component identified in the previous exchange). Given this complete context, the model generates a natural-language answer that explains what was found or done, confirms the tool action if one was taken, and provides any additional technical detail that is useful to the technician. The response is streamed token by token back to the inference engine.

Language-aware response. The language model responds in the same language as the question: Italian questions receive Italian answers, English questions receive English answers. This behaviour is a product of the base model's multilingual pretraining and is preserved through fine-tuning because the format-learning objective does not interfere with language choice in the conversational portion of the output. The TTS engine receives the completed text and automatically selects the matching voice, as described in the previous

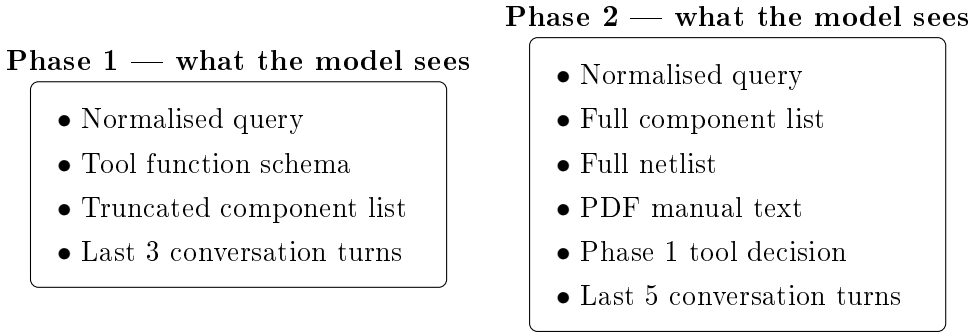


Figure III.8: Context window contents for each phase of the deployed Two-Stage pipeline.

subsection.

Real-time streaming and dual-channel delivery. Phase 2 generation is performed with token-level streaming enabled on the vLLM server. Each token is forwarded to the chat panel as it is produced, so the answer appears incrementally and the technician can begin reading before generation is complete. Once the full text is available, it is passed to the TTS engine, which begins audio playback. The result is simultaneous visual and auditory delivery of the same answer: the text is written in the assistant chat panel and spoken aloud in real time.

Conversation history and multi-turn continuity. The inference engine maintains a per-session message history. Each completed exchange (user query and assistant reply) is appended to the history and injected into subsequent requests. Phase 1 receives up to three recent turns to allow the tool-decision logic to resolve context-dependent references; Phase 2 receives up to five turns to support richer conversational continuity. This enables natural multi-turn interactions where the technician can follow up with questions like "and what is it connected to?" without repeating the component name.

Model auto-detection. At startup, the inference engine queries the `/v1/models` endpoint of the vLLM server to retrieve the identifier of the currently loaded model. This removes the need for hardcoded model strings and allows the same inference code to serve any fine-tuned checkpoint loaded into the server without reconfiguration.

JSON robustness. The JSON tool call returned by Phase 1 is passed through a sequence of regular expression corrections before dispatch. The most common generation artefacts corrected are trailing commas before closing brackets, single-quoted string keys, and truncated objects caused by early stopping. These corrections increase the fraction of Phase 1 responses that yield a valid, dispatchable tool call without requiring a retry.

Voice and text pathways. For voice input, the inference engine sends the received audio bytes to the STT microservice and waits for the transcription. For text input, the query is passed directly. Both pathways enter the normalisation step and then Phase 1 identically. The REST API exposes a dedicated endpoint for each modality, but the inference logic below the API layer is fully

shared.

III.3.9 REST API Server

The inference engine is exposed to the ARBoard application through a Flask REST API mounted alongside the other application routes, with all endpoints scoped to a specific board by its database identifier.

The voice assistance endpoint accepts an audio file as a multipart upload. It passes the file to the inference engine, which transcribes it via the STT microservice and then queries the language model with the resulting text. The response is returned as a JSON object containing the conversational answer and the list of resolved tool calls.

The text assistance endpoint accepts a plain text query and passes it directly to the language model. This endpoint supports the use case where the client device performs speech recognition locally and sends only the text transcription, or where the user types a query rather than speaking it.

A third endpoint triggers the serialization of the database contents for the specified board into the component list and netlist text blocks. These are stored in the database so that subsequent voice and text assistance requests can load them without recomputing the serialization on every call.

All three endpoints handle exceptions by rolling back the database session and returning a structured error response, so that a failed inference call does not corrupt the application's transactional state. The API layer contains no inference logic itself; it delegates entirely to the data access layer, which in turn calls the inference engine. This separation of concerns means that the inference hardware configuration, model selection, and STT backend can all be changed without modifying the API endpoints or the application's route registration.

IV. Results

This chapter presents the quantitative and qualitative evaluation of both contributions described in Chapter III. The Perception Layer is assessed in terms of character recognition accuracy and end-to-end label detection performance on annotated PCB documents, followed by a qualitative evaluation of the thermal image analysis workflow. The Reasoning Layer is evaluated across four distinct efforts: a preliminary survey of large frontier models to establish that the task is solvable; a joint model and inference method comparison across seven architectures and three methods to select the best out-of-the-box configuration; a dataset variant and architecture study comparing English deterministic and bilingual synthetic training data across six architectures using three fine-grained metrics; and a final fine-tuning run on the selected model and dataset at LoRA ranks $r = 32$ and $r = 64$ to characterise the accuracy improvement and rank sensitivity. Independent evaluations of the speech-to-text backends and the post-training quantization step round out the chapter. Section IV.2.2 explains why these evaluations are not redundant and what question each one is designed to answer.

IV.1 Part I: Perception Layer

IV.1.1 Experimental Setup

The pipeline was validated on the STM32 Nucleo MB1136 board family, using both the placement drawing (a multi-layer PDF with component outlines and reference designators) and the electrical schematic (a multi-sheet PDF with net annotations). These two document types present structurally different challenges: the placement drawing contains densely packed reference designators printed at 0° and 90° to the board edge, with labels adjacent to component pads and board traces; the schematic contains reference designators at 0° and occasionally 45° near connector symbols, with characters that may partially overlap net annotation lines. Figure IV.1 shows a representative crop from the schematic at the processing zoom level.

IV.1.2 End-to-End Pipeline Detection Performance

The pipeline was evaluated through visual inspection and direct comparison against the board database assembled from its output. Detection performance was assessed qualitatively, with the database used as a functional ground truth: a reference designator was considered correctly recovered if it appeared in the final spatial database with the correct identifier string and a plausible coordinate position. Figures IV.2 show the effect of the cascade on a placement drawing page.

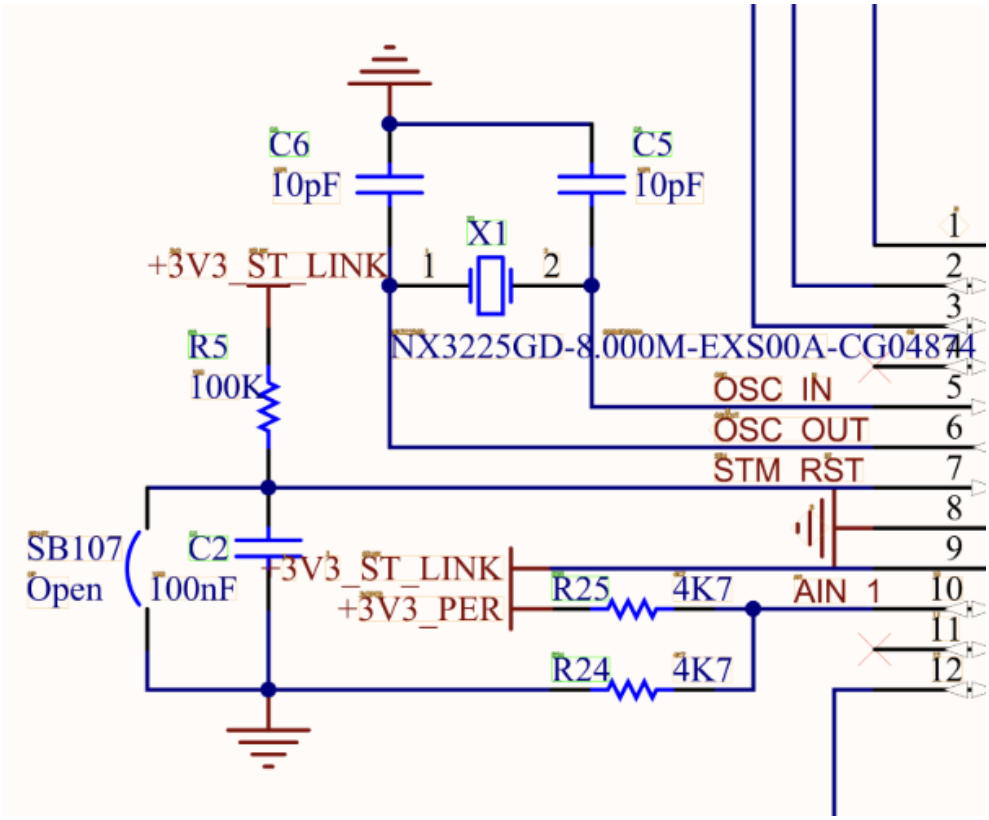
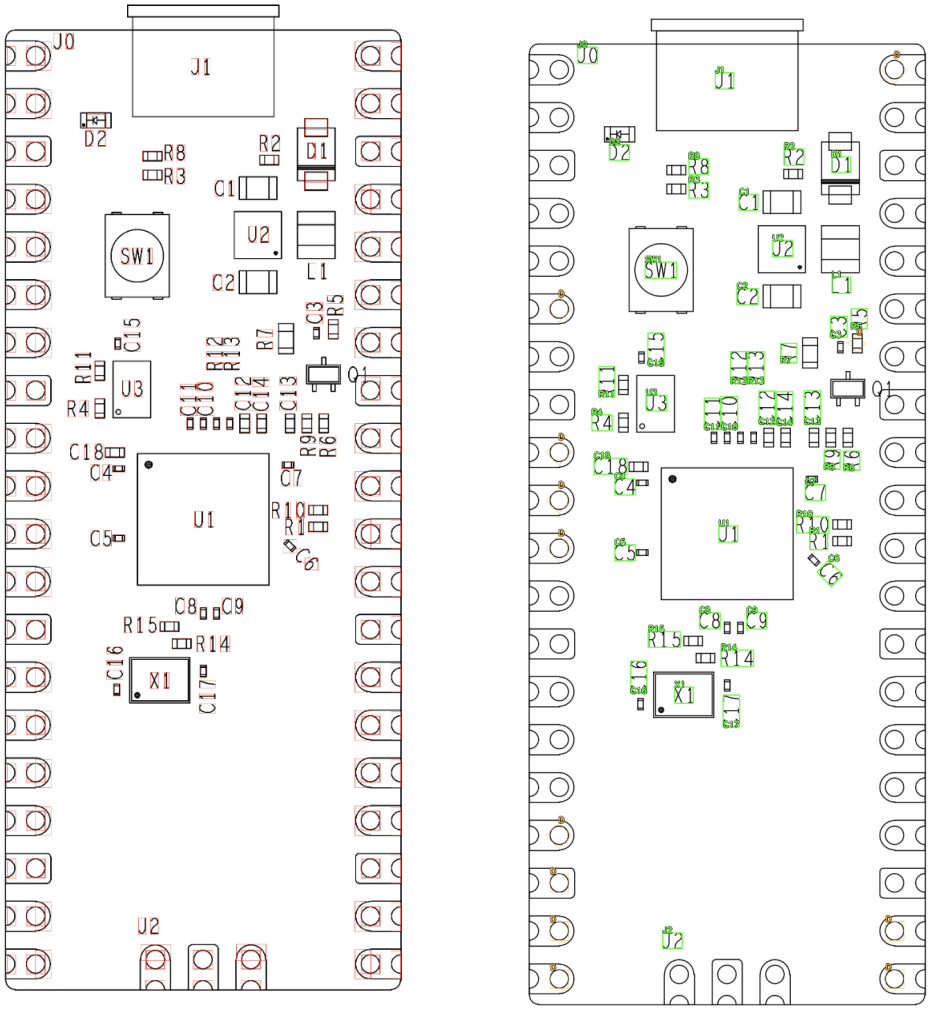


Figure IV.1: Representative schematic crop from the evaluation corpus.

The three-rotation strategy recovered all reference designators printed at 90° on the placement drawing that were missed in a single-pass 0°-only run. The 45° pass contributed incrementally on the schematic, capturing a small number of labels near diagonal connector symbols that the orthogonal passes missed. The NMS pass at the multi-rotation merge stage removed all duplicate detections arising from the same label being found in multiple rotation passes without discarding any distinct detection.

The primary source of false negatives observed during inspection was partial overlap between reference designator bounding boxes and thick board outline traces. After morphological long-line removal, strokes adjacent to the board edge occasionally stripped character strokes from nearby labels, causing them to fail the minimum contour area threshold before reaching the cross-correlation step. The primary source of false positives was via pad annuli whose bounding box aspect ratio fell within the character size range at low-density board regions; these were largely suppressed by the Euler number gate, which rejects contours whose internal hole count is inconsistent with alphanumeric characters.

The assembled database was used without correction for all fine-tuning dataset generation and all AR overlay experiments reported in Part II of this chapter, confirming that the detection output is of sufficient quality to support downstream use. A controlled annotation-based evaluation with precise Precision,



(a) Candidates surviving the geometric gate cascade.

(b) Final detected labels after cascade and NMS.

Figure IV.2: Effect of the four-gate cascade and non-maximum suppression on a placement drawing page.

Recall and F1 scores across rotation configurations is a natural extension of this work, but was not performed within the scope of this thesis.

IV.1.3 Thermal Image Analysis

The thermal alignment workflow was validated on one board configuration (STM32 Nucleo MB1136) captured with a handheld infrared camera. For each test, the user selected the four corners of the PCB in the alignment interface in the order top-left, top-right, bottom-right, bottom-left; the perspective transform was then computed and applied automatically by the backend. Figure IV.3 shows the reference board under white light; the thermal images that

follow were aligned to this board's spatial database.

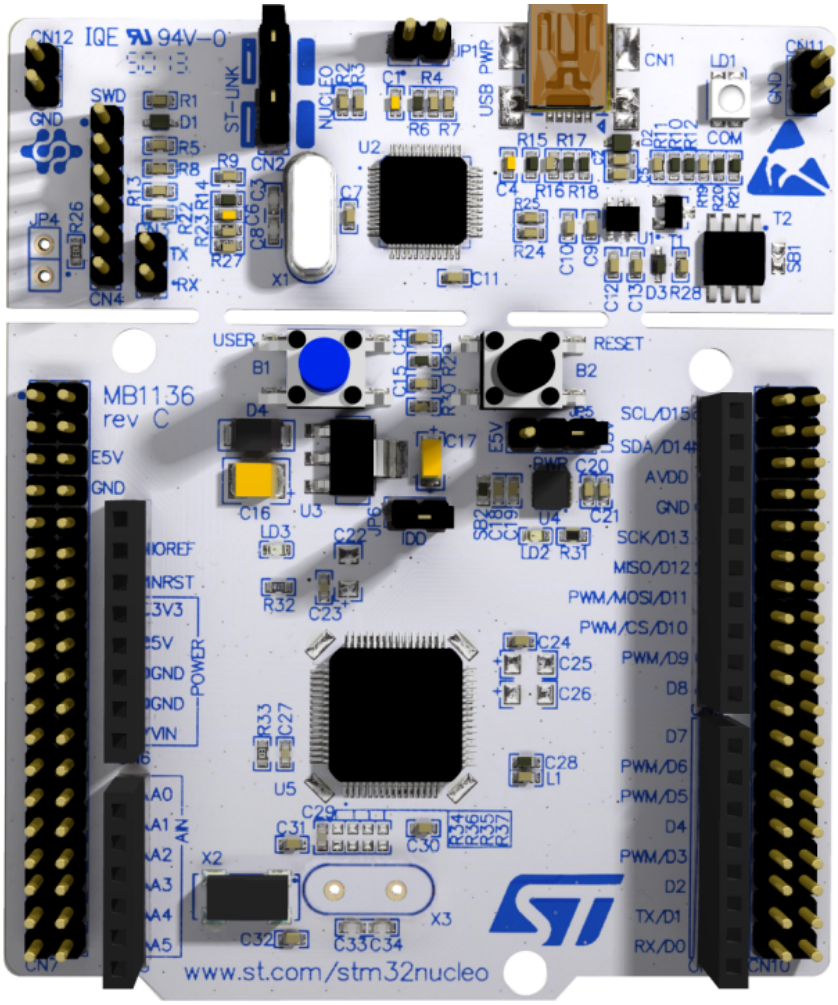


Figure IV.3: Reference PCB (STM32 Nucleo MB1136) under white light.

The alignment step bridges the gap between the unconstrained capture geometry of a handheld thermal camera and the fixed coordinate system of the spatial database. Because the camera is not mounted at a fixed position relative to the board, each capture introduces a different perspective distortion: the board may appear tilted, foreshortened, or rotated depending on the capture angle. A direct overlay of component polygons onto the raw thermal image would therefore fail unless the images were registered to a common coordinate frame. The four-point homography approach solves this by computing the projective transformation that maps the board corners from the thermal image plane to the reference coordinate system; the user selects the four PCB corners in a fixed order (TL, TR, BR, BL) through the alignment interface shown in Figure IV.4, and the backend computes the 3×3 homography matrix from the four point correspondences.

With the homography matrix determined, the backend applies the perspective warp to the thermal image using bilinear interpolation, yielding a rectified

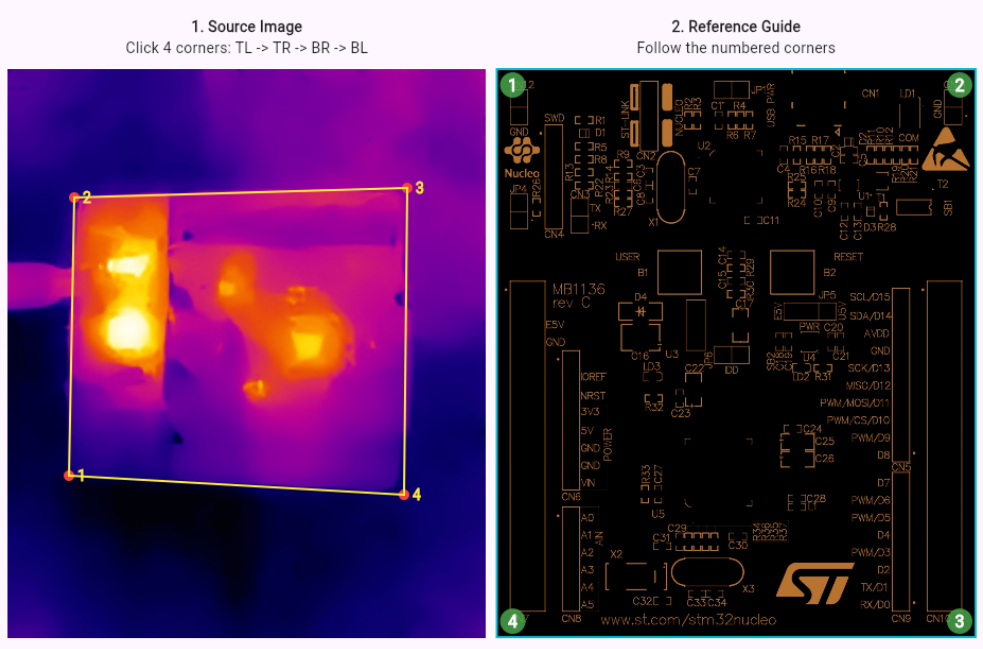
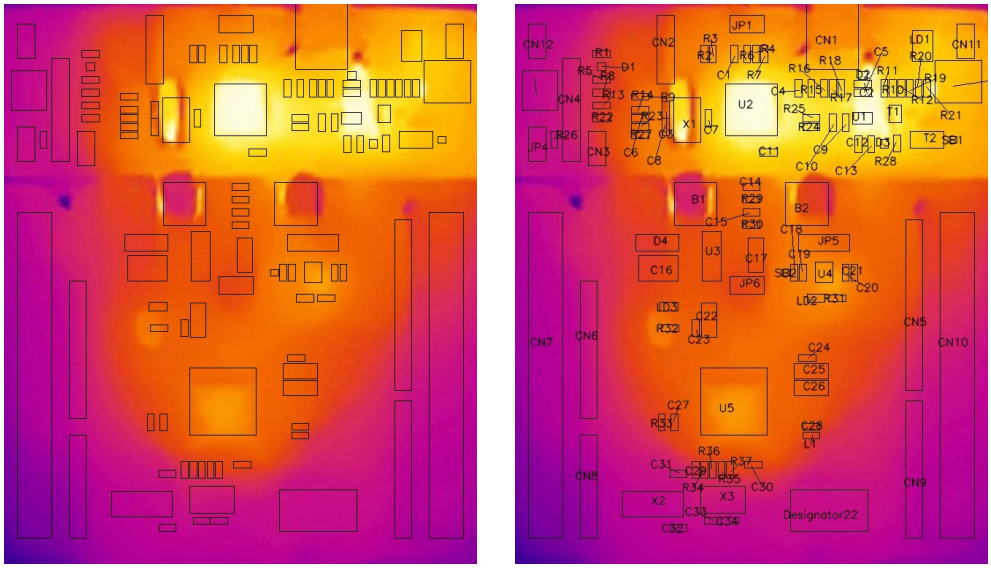


Figure IV.4: Homography alignment interface showing the thermal image and reference PCB layout.

thermal map in the same coordinate frame as the reference layout. Component footprint polygons stored in the spatial database are then projected directly onto the rectified image without any further registration step. For each component, a pixel-level mask is computed from its polygon coordinates, and the mean and maximum pixel values within the mask are extracted and converted to temperature using the calibration range read from the thermal camera's accompanying report. This per-component extraction is fully automatic once the four-corner alignment is complete; Figure IV.5 shows the rectified thermal image with the projected bounding boxes and component labels.

After rectification, the thermal image was available at 648×767 pixels for analysis. Table IV.1 shows a representative sample of per-component temperature statistics extracted under normal operating conditions; the temperature range read from the accompanying PDF report was 18–35 °C.

The extracted temperatures are consistent with the expected power dissipation of the board under normal operation. The highest mean and peak temperatures occur at the main ICs (U1, U2) and the crystal (X1), with values in the 29–35 °C range; decoupling capacitors (e.g. C5) and resistors in the supply path show elevated temperatures, while connectors and passive components away from the main power rails remain near ambient (18–22 °C). No anomalously high hotspots were observed, and the pipeline successfully mapped all component polygons onto the rectified thermal image for automatic temperature reporting.



(a) Rectified thermal image with component bounding boxes projected from the spatial database.

(b) Same view with reference designators overlaid on each component polygon.

Figure IV.5: Result of the thermal alignment pipeline after homography rectification.

Table IV.1: Representative per-component temperature statistics extracted from a rectified thermal image.

Reference	**Type**	**Mean Temp. (°C)**	**Max Temp. (°C)**
U2	QFP48 7×7	34.35	34.94
U1	SOT-23-5L	34.21	34.76
X1	XTAL1	29.05	34.94
C5	0603C	33.68	34.48
U5	LQFP64	25.42	27.33
R21	0603R	31.14	32.76
LD1	HSMF-A201	29.62	30.59
CN8	1×6P_FEMALE	18.42	19.81
X2	XTAL_4SM	19.24	19.90

IV.2 Part II: Reasoning Layer

IV.2.1 Experimental Setup

All experiments were conducted on the DGX Spark workstation equipped with the NVIDIA Blackwell GB10 Superchip. Two distinct evaluation setups are used across the four questions addressed in this chapter. The first is the LLM-Benchmark evaluation (Questions 1, 2, and 4): this uses a 40-question test set produced by the deterministic dataset generator with an independent seed, and evaluates models via the LLMBenchmark evaluation client. The second is the in-process fine-tuning harness (Question 3): this uses a separate test set drawn from a pool generated with seed 99, guaranteed non-overlapping with the 1 000-example training set (seed 42) by a set-intersection check enforced at generation time. The harness was executed with 80 evaluation examples for the English deterministic experiments and 100 examples for the bilingual synthetic experiments. Both test sets are drawn from the same four tool classes — single-component location (`highlight_component`), net tracing (`highlight_net`), component-type queries (`highlight_by_type`), and negative constraints (no tool call) — with the same weighted distribution used during training. Models evaluated in-process are loaded via the Unsloth framework in BF16 precision; inference uses greedy decoding throughout to ensure deterministic and reproducible results. The two evaluation setups produce numbers that are not cross-comparable, as explained in Section IV.2.2.

The primary metric used in Questions 1, 2, and 4 (the model survey, the baseline sweep, and the final fine-tuning evaluation) is **tool-call accuracy**: the fraction of evaluation questions for which the model's response, after JSON extraction and tool-call resolution, exactly matches the ground-truth tool call. Resolution expands high-level calls such as `highlight_network(component_id=X)` into the canonical `highlight_component` form by querying the board database, so the model is credited for identifying the correct component or net even if it uses a different but semantically equivalent representation. A question is counted correct only when both the tool name and all parameters match after this resolution step. Question 3 uses a different evaluation harness (in-process evaluation harness) that produces three additional metrics alongside tool-call accuracy: JSON validity rate, tool selection accuracy, and parameter-level accuracy. Those three metrics are defined in Section IV.2.5 and reported in Table IV.4; they are not cross-comparable with the single-accuracy figures from Questions 1, 2, and 4 because they use a different test set format and a different scoring function.

IV.2.2 Four Evaluations, Four Questions

Part II contains four distinct evaluations. They are not redundant: each one answers a different question, uses a different toolchain or model family, and the output of each one feeds directly into the next.

Question 1 — Is the task solvable at all? The preliminary model sur-

vey (Table IV.2) evaluates a broad range of large frontier models (20B–120B parameters, cloud API or local via LM Studio) on a fixed 40-question ground truth file using the LLMBenchmark framework. Its purpose is not to select a deployment model; it is to establish that the task can be solved and to set an accuracy ceiling. If no model had achieved above chance, fine-tuning a compact model would not be justified.

Question 2 — Which model and inference method should be used? The method and architecture comparison (Table IV.3) evaluates the deployable 8B-class architectures at baseline on a 40-question generated test set using the LLMBenchmark framework. One-Shot inference is run for all evaluated models. The three highest-performing architectures are then additionally evaluated with Single-Call and Two-Stage inference, providing a complete picture of how inference method interacts with model architecture. No fine-tuning is applied; all results are out-of-the-box. This evaluation jointly selects the architecture to fine-tune and the inference method for the production system.

Question 3 — Which training dataset and architecture combination produces the best fine-tuned model? The dataset variant study (Table IV.4, Section IV.2.5) fine-tunes six architectures at multiple LoRA ranks under two dataset compositions — the English-only deterministic generator and the bilingual synthetic generator — and evaluates with a dedicated in-process evaluation harness reporting three diagnostic metrics: JSON validity rate, tool selection accuracy, and parameter-level accuracy. This harness is intentionally different from LLMBenchmark; it runs the model in-process and evaluates in the training format, making it sensitive to fine-tuning quality and dataset composition effects. The output of this evaluation is a ranked list of architecture–dataset pairs, informing the configuration for Question 4.

Question 4 — How much does fine-tuning help, and at what LoRA rank? The final fine-tuning campaign (Table IV.5, Section IV.2.6) trains the architecture selected in Questions 2 and 3 at multiple LoRA ranks using the best dataset identified in Question 3. The LLMBenchmark evaluation of the fine-tuned checkpoints was not completed within the thesis timeline; rank sensitivity and format-learning effectiveness are quantified through the complementary in-process harness (Table IV.6).

Why the numbers across the four evaluations are not directly comparable. Questions 1 and 2 both use LLMBenchmark with a single accuracy metric, but on different question sets (Q1: 40-question hand-crafted fixed file; Q2/Q4: 40-question deterministically generated set) and different model families (large frontier vs. compact 8B-class). Question 3 uses a separate in-process harness with three metrics on an 80-question (English deterministic) or 100-question (bilingual synthetic) training-format test split; its numbers cannot be compared to Questions 1, 2, or 4. Question 4 is directly comparable to Question 2 for the selected model only, since both use the same 40-question generated test set and the same evaluation logic. Numbers within each evaluation are internally consistent; cross-evaluation comparisons are not meaningful.

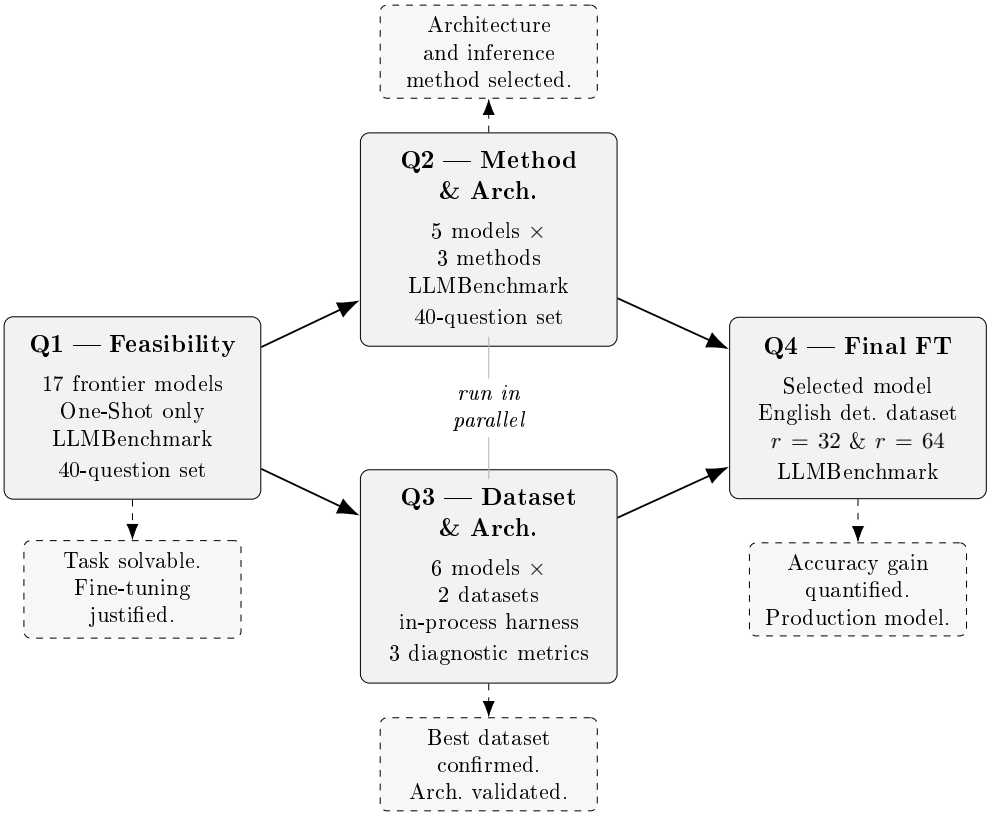


Figure IV.6: Dependency structure of the four evaluations in Part II.

IV.2.3 Preliminary Model Survey

Before designing the fine-tuning pipeline, LLMBenchmark was used to evaluate the out-of-the-box tool-calling capability of a broad range of large frontier models, answering the first of the four questions above: whether the task is solvable. All runs used the fixed 40-question ground truth set and the One-Shot inference strategy. Models were accessed either via the OpenRouter cloud API or through a local LM Studio endpoint on the DGX Spark; the latter was used only for models that fit within the available VRAM budget. Table IV.2 reports the best and mean accuracy across all recorded runs for each model.

The survey established that the task is solvable: three models (arcee-ai/trinity-mini via cloud API, harbinger-24b and zai-org/glm-4.6v-flash locally) achieved perfect accuracy in at least one run. However, two findings ruled out direct deployment of any of these models as the production inference backend. First, all models reaching above 90% accuracy have parameter counts of 20 billion or more, placing them above the memory and throughput envelope that allows real-time serving alongside the rest of the ARBoard application stack. Second, a number of models exhibited significant run-to-run variance: openai/gpt-oss-120b ranged from 35.0% to 90.0% across five runs, and nvidia/nemotron-30b-a3b ranged from 37.5% to 87.5% across three local FP8 runs, indicating sensitivity to prompt phrasing or model load state that would be difficult to control in a production environment. Cloud API access, while offering higher

Table IV.2: Preliminary model survey results under the One-Shot inference strategy.

Model	**Mode**	**Best (%)**	**Mean (%)**	**Runs**
Cloud API (OpenRouter)				
arcee-ai/trinity-mini	Cloud API	100.0	93.8	2
openai/gpt-oss-20b	Cloud API	95.0	93.8	2
mistralai/devstral-2512	Cloud API	92.5	92.5	1
minimax/minimax-m2	Cloud API	87.5	87.5	1
alibaba/tongyi-30b-a3b	Cloud API	87.5	87.5	1
nvidia/nemotron-30b-a3b	Cloud API	85.0	85.0	1
openai/gpt-oss-120b	Cloud API	80.0	80.0	1
z-ai/glm-4.5-air	Cloud API	75.0	75.0	1
essentialai/rnj-1-instruct	Cloud API	70.0	70.0	2
Local inference (LM Studio on DGX Spark)				
harbinger-24b	Full prec.	100.0	97.5	2
zai-org/glm-4.6v-flash	Full prec.	100.0	95.6	4
harbinger-24b	Q6_K GGUF	95.0	95.0	1
harbinger-24b	Q4_K_M	90.0	88.1	4
openai/gpt-oss-120b	Full prec.	90.0	74.4	4
nvidia/nemotron-30b-a3b	FP8	87.5	69.2	3
openai/gpt-oss-20b	Full prec.	82.5	76.3	2

and more stable accuracy, introduces an external dependency and per-request cost incompatible with the offline-first deployment scenario.

These results answer Question 1 affirmatively: the task is solvable, and an accuracy ceiling of roughly 95–100% on the 40-question set has been established by large models. The two findings that rule out direct deployment of those models — parameter count above the hardware budget, and run-to-run variance — directly motivate the fine-tuning approach: by domain-adapting a compact 8B model on synthetic PCB data, the goal is to approach this ceiling while staying within the memory and throughput envelope of the deployed system. The survey also confirmed that One-Shot inference produces parseable and extractable tool calls across model families, making it the natural choice for the fine-tuning supervision format. With the feasibility question settled, Questions 2 and 3 are run in parallel to resolve the two remaining degrees of freedom before committing to the full training run: Question 2 selects the model architecture and inference method by comparing all seven architectures at baseline across the three inference strategies; Question 3 selects the training dataset and confirms the architectural choice by comparing fine-tuning outcomes across dataset variants and model families. Both outputs feed into Question 4.

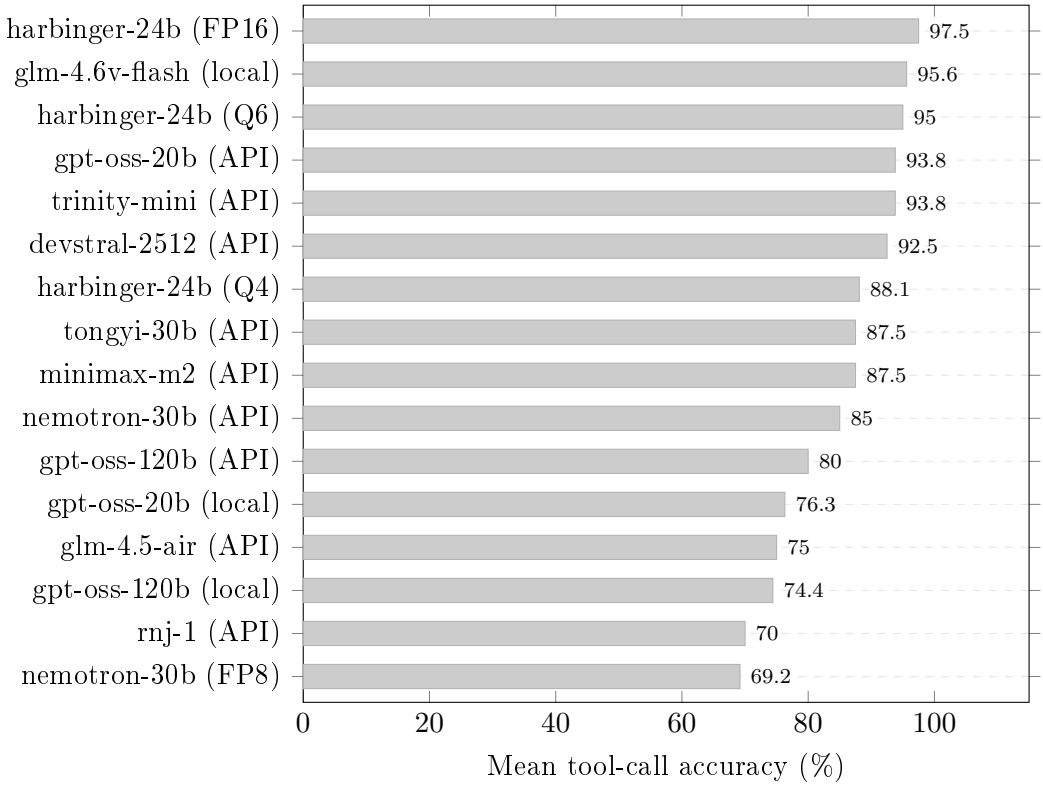


Figure IV.7: Mean tool-call accuracy across all 16 frontier model configurations under One-Shot inference, sorted ascending. Models tagged (API) were accessed via the OpenRouter cloud API; unlabelled local models ran on the DGX Spark hardware. Best-run figures for each configuration appear in Table IV.2. Three configurations achieved perfect accuracy in at least one run (harbinger-24b FP16, glm-4.6v-flash, and trinity-mini API), all with parameter counts above 20 B.

IV.2.4 Model and Inference Method Comparison

With the feasibility question settled (Question 1), Question 2 resolves the two interrelated design choices: which architecture to fine-tune, and which inference method to use in the production system. The LLMBenchmark framework was used to evaluate the candidate architectures at baseline on the 40-question generated test set produced by the deterministic generator.

The evaluation proceeded in two passes. First, every model in Table IV.3 was benchmarked with One-Shot inference, the simplest method and the one compatible with the fine-tuning supervision format. Out of these, the three architectures with the highest One-Shot accuracy were then additionally evaluated with Single-Call and Two-Stage inference. Restricting the multi-method comparison to the top three avoids expending compute on architectures that have already demonstrated insufficient JSON format compliance for the task; for such models, a more structured output schema does not compensate for the underlying capability gap.

Table IV.3 reports tool-call accuracy for all evaluated architectures under One-Shot, and for the three best models, also under Single-Call and Two-Stage.

Table IV.3: Question 2: baseline tool-call accuracy by architecture and inference method.

Model	**One-Shot** (%)	**Single-Call** (%)	**Two-Stage** (%)
Qwen3-8B	100.0	65.0	90.0
Mistral-7B-v0.3	90.0	37.5	87.5
Llama-3.1-8B-Instruct	77.5	20.0	80.0
Qwen2.5-7B-Instruct	40.0	—	—
Phi-3.5-mini-instruct	0.0	—	—

Top three One-Shot models were evaluated across all three methods; the remaining models were evaluated with One-Shot only. Phi-3.5-mini-instruct produced no valid tool calls under One-Shot, confirming that it lacks the instruction-following capacity required for this output format at baseline.

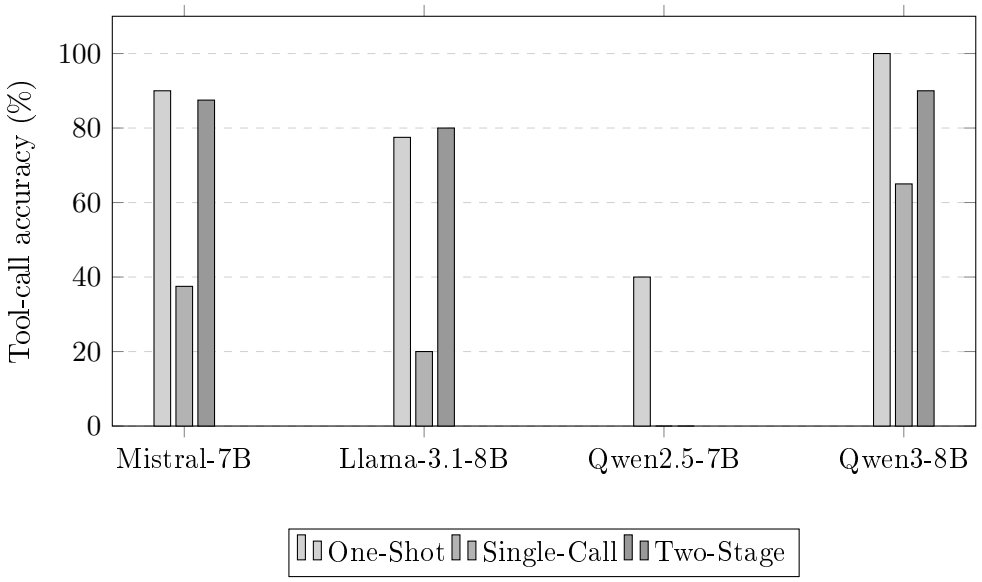


Figure IV.8: Baseline tool-call accuracy per model and inference method (Q2).

Qwen3-8B achieves the highest One-Shot accuracy at 100%, with Mistral-7B-v0.3 at 90% and Llama-3.1-8B-Instruct at 77.5%. Phi-3.5-mini-instruct produces no valid tool calls (0%), confirming it cannot be considered for fine-tuning in this task without fundamental prompt engineering changes. Qwen2.5-7B-Instruct at 40% shows limited tool-call compliance, consistent with that model being a slightly older generation than Qwen3-8B.

For the three models evaluated across all three methods, Single-Call performs worst across the board. Mistral drops sharply from 90% One-Shot to 37.5% Single-Call, and Llama from 77.5% to 20%. This degradation is consistent with

the structural difference of the output schema: Single-Call asks the model to return a JSON object containing both the conversational text and the tool call as fields, which imposes a stricter syntactic constraint than One-Shot where the model writes free text followed by a JSON array. Two-Stage partially recovers this loss (Mistral 87.5%, Llama 80%) because its first call produces only a short JSON decision object without a conversational wrapper, which is easier for the model to maintain format compliance on.

The deployment implications follow directly from these numbers. Two-Stage was selected for the production ARBoard system not because it achieves higher accuracy than One-Shot (the gap is small for Mistral and Qwen3), but because of its latency advantage in the AR interaction loop. In Two-Stage, the model first produces a compact JSON decision object that identifies the tool and parameters; this call is dispatched immediately to the AR viewer, which begins highlighting components while the model continues to generate the conversational answer. The technician therefore sees the AR overlay update at approximately the same moment they hear the voice response begin, rather than waiting for the full generation to complete before either channel activates. For a task where the spatial feedback is as important as the verbal answer, this pipeline ordering meaningfully reduces the felt latency of the interaction.

One-Shot was retained as the fine-tuning supervision format for a structural reason: it is the only inference method whose output schema is directly reproducible by the training data generator. In One-Shot, the model writes a free-text conversational answer and then appends a JSON separator followed by the tool call array; the generator can produce arbitrarily many correctly labelled examples of exactly this format by templating the answer text and populating the tool call from the board database. Two-Stage uses a different output structure for its first call — a compact JSON object with no surrounding text — which would require redesigning the generator to match, an undertaking outside the scope of this work. Fine-tuning for Two-Stage represents a natural extension of the current pipeline.

IV.2.5 Dataset Variant and Architecture Study

Question 3 asks which combination of training dataset and architecture produces the best fine-tuned model. This was evaluated in parallel with Question 2 using a dedicated in-process evaluation harness, which loads each model locally and evaluates it on a held-out JSONL test split in the training format. Unlike the LLMBenchmark evaluation used in Questions 1 and 2, this harness reports three diagnostic metrics per configuration: JSON validity rate (fraction of outputs that parse as valid JSON), tool selection accuracy (exact tool name and parameter match after resolution), and parameter-level accuracy (fraction of component or net identifiers matched correctly). The three-metric reporting is deliberate: it separates format compliance failures (low JSON validity) from semantic failures (valid JSON but wrong tool or parameters), which are caused by different problems and have different remedies.

Six architectures were evaluated under two dataset compositions, chosen to

represent two fundamentally different positions on the trade-off between label correctness and linguistic diversity.

The **English deterministic** dataset derives every training example algebraically from the board database. For each question, the correct component identifier, net name, or component type is read directly from the database record and embedded into a template; no language model is involved in label generation. The result is a dataset with provably correct supervision signals at every example, generated at negligible compute cost. The limitation of this approach is uniformity: the question phrasings are templated, the vocabulary is English only, and the linguistic variety is bounded by the number of templates.

The **bilingual synthetic** dataset was generated by a teacher model prompted with Italian and English persona injections, producing diverse question phrasings, informal register, and code-switching between the two languages. This approach was motivated by the deployment context: the AR system must handle Italian questions, informal phrasing, and speech-to-text artefacts that template-based generators cannot reproduce. The cost of this diversity is label noise: the compact teacher model that generates the bilingual examples is not perfectly reliable on Italian PCB queries, and its output JSON format varies across phrasings and languages, introducing inconsistency in the training signal.

Each architecture was evaluated at baseline (no adapter) and after fine-tuning at multiple LoRA ranks under each dataset variant.

Table IV.4 reports the baseline figures, sorted by tool selection accuracy within each dataset group. Fine-tuned checkpoint results from the in-process evaluation harness are reported separately in Table IV.6.

Table IV.4: Question 3: baseline performance across six architectures and two dataset variants. Fine-tuning degradation figures are discussed in the text; fine-tuned checkpoints evaluated with the in-process evaluation harness appear in Table IV.6.

Model	Configuration	JSON Val. (%)	Tool Acc. (%)	Param. (%)
English deterministic dataset (n=80 questions)				
Mistral-7B-v0.3	Base	98.75	**78.75**	68.75
Gemma-2-9B-IT	Base	98.75	73.75	62.50
Llama-3.1-8B	Base	88.75	66.25	58.75
DeepSeek-R1-8B	Base	57.50	41.25	37.50
Qwen3-8B	Base	30.00	28.75	28.75
Phi-3.5-mini	Base	21.25	21.25	21.25
Bilingual synthetic dataset (n=100 questions)				
Llama-3.1-8B	Base	89.00	69.00	58.00
DeepSeek-R1-8B	Base	64.00	49.00	46.00
Qwen3-8B	Base	32.00	29.00	28.00

Gemma-2-9B-IT evaluated on English deterministic only. $\alpha = 2r$ for all fine-tuning runs.

Three findings emerge from the baseline comparison.

Mistral-7B-v0.3 is the architectural winner. It achieves the highest baseline tool selection accuracy (78.75%) and the highest JSON validity (98.75%), ahead of Gemma-2-9B-IT (73.75%) and Llama-3.1-8B-Instruct (66.25%). Fine-

tuning at r=32 leaves these numbers essentially unchanged (JSON validity 98.75%, tool selection 77.50%, parameter accuracy 68.75%), confirming that the model absorbs the LoRA adaptation without degrading its output format compliance. Mistral is the only architecture in this group that shows this robustness property and is therefore the primary candidate for the fine-tuning campaign in Question 4.

Architecture determines fine-tuning robustness more than dataset choice. All other architectures exhibit monotonic rank-degradation: fine-tuning consistently hurts rather than helps, with performance declining as LoRA rank increases. For Llama-3.1-8B on the English deterministic dataset, tool selection accuracy falls from 66.25% at baseline to 60.00% (r=16), 50.00% (r=32), and 41.25% (r=64). On the bilingual synthetic dataset the degradation is steeper: baseline 69.00% drops to 42.00% at r=32 and 30.00% at r=64. DeepSeek-R1-Distill-Llama-8B collapses to a fixed 29.0% across all three ranks on the bilingual synthetic dataset, suggesting the model converged to a degenerate output mode. Qwen3-8B and Phi-3.5-mini show no change in either direction: their FT r=32 results are identical to base, indicating the adapter has no effect from a starting point already at or near minimum JSON compliance.

The English deterministic dataset is the more reliable training signal. Fine-tuning on the bilingual synthetic dataset produces substantially more degradation across architectures than equivalent runs on the English deterministic dataset. Three interacting factors explain why this outcome is unsurprising. First, teacher-model label noise is structurally present: the compact teacher that generates the bilingual examples is less reliable on Italian PCB queries than on English ones, injecting incorrect supervision signals. Second, phonetic transcription errors and informal shorthand in the Italian prompts introduce noise in the question text that is faithfully reproduced and then learned by the student. Third, the bilingual format combines two languages with different punctuation and JSON conventions in a single dataset, so the model must simultaneously normalise across language-specific variation and maintain strict JSON syntax. The English deterministic generator has none of these failure modes: every label is derived directly from the board database, making the training signal provably correct and perfectly consistent across examples.

An additional observation concerns Qwen3-8B. In the LLMBenchmark evaluation on the 40-question deterministically generated test set (Table IV.3), this model reaches 100% tool-call accuracy, the highest of any deployable model tested. This might appear to make it the obvious candidate for fine-tuning. However, its baseline performance on the in-process harness drops to 30.0% JSON validity on the Q3 evaluation set. The two evaluation sets differ in a critical way: the LLMBenchmark questions are clean, consistently formatted, and monolingual, while the in-process harness questions are generated with linguistic variation, contextual perturbations, and bilingual question phrasings. Qwen3-8B's accuracy collapses under this perturbation, indicating that its perfect score on the clean benchmark reflects narrow distribution fit rather than robust task understanding. A model that depends on clean, well-formatted inputs to produce valid JSON is not a reliable base for a workshop deployment

where speech-to-text artefacts, informal phrasing, and mixed-language queries are the norm. This result is a cautionary example of the gap between benchmark performance and operational robustness: the model that scores highest on a clean evaluation set is not necessarily the model that generalises best to realistic deployment conditions.

These results directly inform Question 4: the final fine-tuning campaign uses Mistral-7B-v0.3 with the English deterministic dataset.

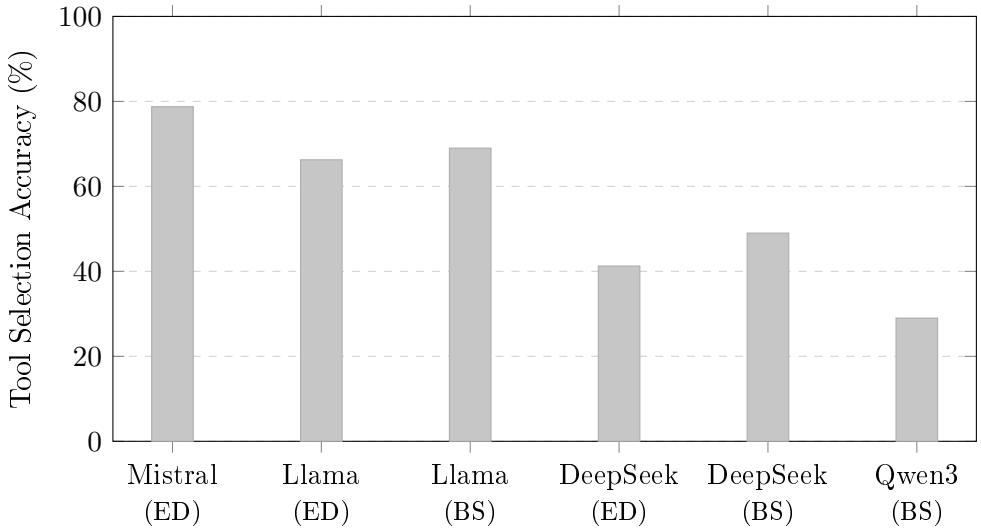


Figure IV.9: Baseline tool selection accuracy across model–dataset configurations in the Q3 study. ED: English deterministic dataset; BS: bilingual synthetic dataset.

IV.2.6 Fine-Tuning Results

With the best model and inference method identified from Question 2 and the most effective dataset composition established by Question 3, the evaluation turns to Question 4: how much does fine-tuning help, and does the LoRA rank matter? The planned evaluation was to fine-tune Mistral-7B-v0.3 at $r = 32$ and $r = 64$ using the English deterministic dataset and score the results with LLMBenchmark on the same 40-question test set used in Question 2. A pipeline failure (training run interrupted before the adapter was saved, with the subsequent automated rerun incorrectly skipping the training step) prevented this evaluation from completing within the thesis timeline. Table IV.5 shows the Phase 1 baselines for all architectures for reference; the quantitative fine-tuning answer to Question 4 is provided by the complementary in-process harness evaluation in Table IV.6.

Training loss behaviour. Across all fine-tuned configurations, training loss follows the same qualitative pattern: starting between 2.0 and 2.5 at step 1, it descends sharply and reaches values below 10^{-3} within approximately the first half of the single training epoch (by step 10–15 of 31 for a 980-sample training

Table IV.5: Phase 2: Phase 1 baseline results for all evaluated architectures. The LLMBenchmark Q4 evaluation of fine-tuned checkpoints for the selected architecture was not completed within the thesis timeline; fine-tuned results from the in-process evaluation harness are reported separately in Table IV.6.

Model	**Base Acc. (%)**	**Selected for FT**	**LLMBenchmark FT**
Qwen3-8B	100.00	No	—
Qwen2.5-7B-Instruct	40.00	No	—
Mistral-7B-v0.3	90.00	Yes†	not evaluated‡
Llama-3.1-8B-Instruct	77.50	No	—
Phi-3.5-mini	0.00	No	—

† Selected based on Q3 harness robustness: highest baseline tool selection (78.75%) with no FT degradation.
‡ Training run interrupted before adapter was saved; corrective run did not complete within thesis timeline.

set). The evaluation loss tracks the training loss closely throughout, confirming that convergence reflects genuine generalisation rather than overfitting. This rapid convergence is consistent with the narrow scope of the fine-tuning task: the model is not learning PCB component semantics from scratch but rather adapting a single behavioural property — emitting a fixed separator followed by a JSON-formatted tool call at a predictable location in every response. Because the base model already follows instructions and produces structured output in other contexts, the gradient signal required to steer it toward this specific format is small, and the adaptation saturates quickly. The rapid convergence is therefore not evidence that the training data is too easy or the dataset too small; it is evidence that the fine-tuning objective is well-matched to what the base model already knows how to do.

A rank effect is visible in the early steps of Figure IV.10: at step 2, Llama r=32 reaches a loss of 0.956 and r=64 reaches 0.662, while r=16 is at 1.412, indicating that higher ranks converge marginally faster in the steepest phase of the descent. By step 10 all three Llama ranks are below 0.002 and the curves become indistinguishable for the remainder of the epoch. The eval/loss values confirm the same ordering at saturation: 0.01233 for r=16, 0.00091 for r=32, and 0.0000503 for r=64. However, the evaluation accuracy (Table IV.6) does not follow this ordering: r=16 achieves 97.5% parameter accuracy, while both r=32 and r=64 reach 100%. The difference in eval/loss between r=32 and r=64 is roughly one order of magnitude ($\approx 18\times$) but produces no accuracy difference, confirming that all three ranks converge to functionally equivalent adapters for this task and that the minimum loss is not the binding constraint on accuracy.

DeepSeek-R1-Distill-Llama-8B exhibited markedly different dynamics in the Q3 study: its training loss remained above 0.01 at 80% of the epoch and converged more gradually at both LoRA ranks tested. This behaviour is consistent with the model's pretraining objective, which optimises for extended chain-of-thought reasoning through a distillation process from a larger reasoning model. That pretraining instils a strong prior toward long, deliberative outputs, which directly conflicts with the short, deterministic, separator-delimited output format required by the tool-calling task. Adapting away from that prior demands

more gradient updates than adapting an instruction-following model, producing the characteristically slower descent observed for this architecture. This also explains why DeepSeek-R1-Distill showed the sharpest accuracy degradation in the Q3 harness: the LoRA adapter was pushing against a stronger competing prior than was present in Mistral or Llama.

Figure IV.10 plots the per-step training loss across all evaluated configurations.

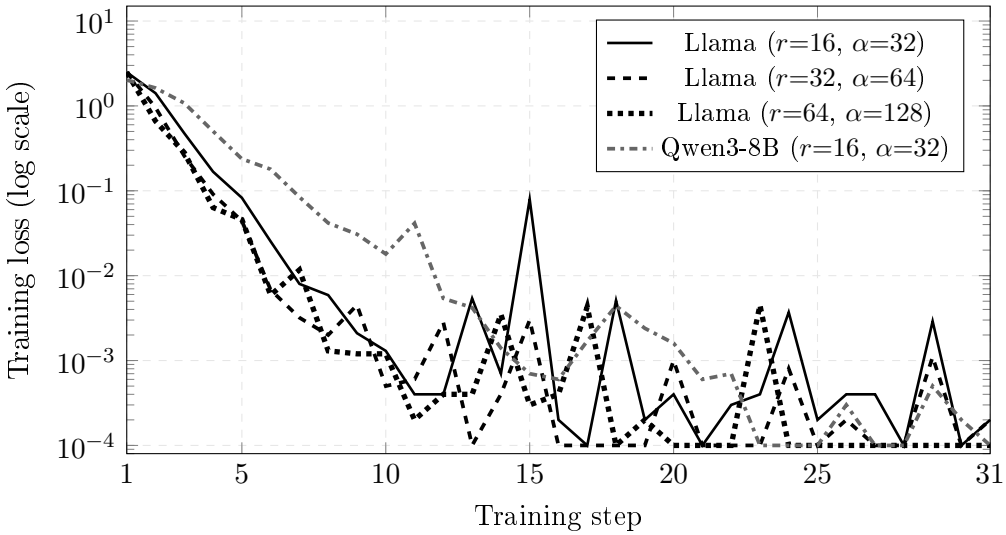


Figure IV.10: Per-step training loss across all fine-tuned configurations on the English deterministic dataset (980 training samples, 31 steps, 1 epoch). All curves start between 2.0 and 2.5 and descend below 10^{-3} within approximately the first half of the epoch. Higher Llama ranks converge marginally faster in the early steps; by step 10 all three ranks are indistinguishable. Occasional spikes in later steps are gradient noise at already negligible loss values.

What the available evidence says about fine-tuning effectiveness. The LLMBenchmark Q4 evaluation of fine-tuned Mistral checkpoints was not completed within the thesis timeline. A training run for the r=16 checkpoint was interrupted before the adapter was saved (the process created the output directory but did not write the adapter configuration file); a subsequent automated rerun incorrectly detected the existing directory and skipped training. The Q3 in-process harness result for Mistral FT r=32 provides partial evidence: JSON validity 98.75%, tool selection 77.50%, parameter accuracy 68.75%, essentially identical to the base performance (98.75%/78.75%/68.75%). This shows that the LoRA adapter does not degrade Mistral's existing format compliance, but the Q3 harness test set differs from LLMBenchmark's, so it does not directly answer whether fine-tuning closes the 10% gap that separates Mistral's baseline (90%) from the ceiling established in Question 1. That measurement remains open.

The in-process evaluation harness data for Llama-3.1-8B across three ranks (Table IV.6) provides a complementary and complete answer to the rank sen-

sitivity question. On the training-format test set, all three ranks succeed in adapting the model to the required output schema within a single epoch on 980 examples. The saturation from 97.5% at r=16 to 100% at r=32 with no further gain at r=64 is the clearest available signal on rank sensitivity for this task: doubling from 16 to 32 closes the residual parameter accuracy gap, but quadrupling from 16 to 64 adds nothing. This result is consistent with the gradient magnitude analysis from the training logs: eval/loss at saturation is roughly one order of magnitude lower for r=64 than for r=32 ($\approx$18$\times$), yet both produce the same accuracy, suggesting the binding constraint is the test set's 40 questions rather than the adapter's capacity. For practical purposes, r=32 with α=64 is the recommended configuration for Llama-3.1-8B on this task.

A complementary set of fine-tuning checkpoints was evaluated using the in-process evaluation harness against a held-out slice of the training-format JSONL file (n=40 held-out test examples). This harness differs from both LLMBenchmark and the Q3 evaluation pipeline: it runs the model inside the same Python process used for training and scores outputs against examples in the training output format. Because base models are not pre-trained on this specific output schema, their scores on this harness are near zero regardless of architecture; those values appear in the table only for reference and do not reflect general task competence. Table IV.6 reports the results.

Table IV.6: Fine-tuned checkpoint evaluation using the in-process evaluation harness (n=40 held-out examples, English deterministic dataset). Near-zero base scores reflect format mismatch between the training-format output schema and base model behaviour, not task incompetence. Mistral base is 0/0/0 for the same reason.

Model	**Configuration**	**JSON Val. (%)**	**Tool Acc. (%)**	**Param. (%)**
Llama-3.1-8B-Instruct ($\alpha = 2r$)				
	Base	5.00	2.50	0.00
	FT r=16	**100.00**	**100.00**	97.50
	FT r=32	**100.00**	**100.00**	**100.00**
	FT r=64	**100.00**	**100.00**	**100.00**
Qwen3-8B ($\alpha = 2r$)				
	Base	0.00	0.00	0.00
	FT r=16	80.00	80.00	80.00
Mistral-7B-v0.3				
	Base	0.00	0.00	0.00
	FT r=16	*adapter not saved; evaluation not performed*		

Mistral FT r=16: training interrupted before adapter save.

Three conclusions emerge from Table IV.6. First, fine-tuning is unambiguously effective for the format-learning objective: Llama-3.1-8B at r=16 raises JSON validity from 5.0% to 100% and tool selection accuracy from 2.5% to 100%, with parameter accuracy at 97.5%. The base model, despite scoring 77.5% on the clean LLMBenchmark test set, does not produce output in the training-format schema at all; a single epoch with the English deterministic dataset corrects this completely. Second, rank sensitivity for Llama is modest but measurable:

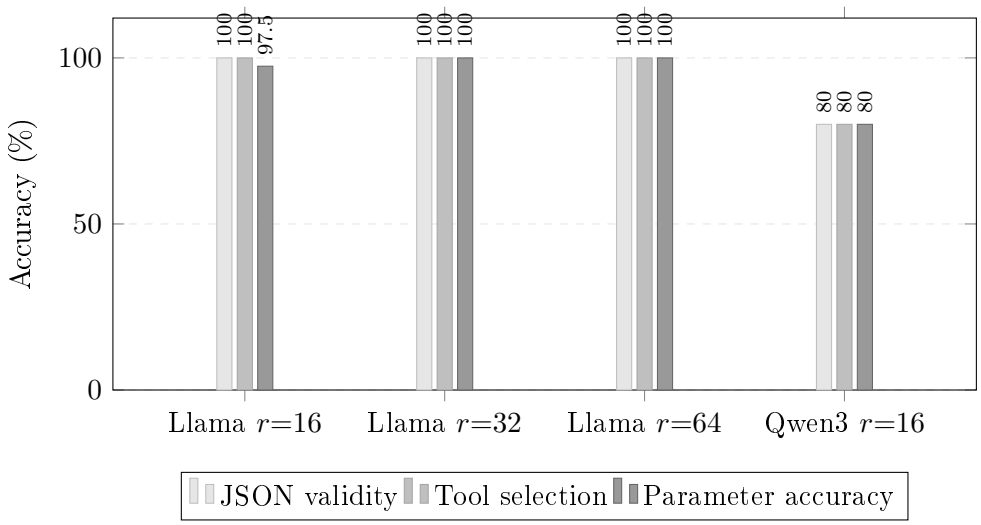


Figure IV.11: Fine-tuned checkpoint accuracy on the in-process evaluation harness (n=40 held-out examples, English deterministic dataset). All three metrics are reported per configuration. Llama-3.1-8B achieves full format compliance at all three ranks; the 2.5% parameter accuracy shortfall at r=16 (dark bar) closes at r=32 and remains at saturation for r=64. Qwen3-8B reaches 80% across all metrics at r=16, indicating partial but incomplete adaptation. Base model scores are near zero for all architectures on this harness due to format mismatch with the training-format schema, and are not shown.

going from r=16 to r=32 closes the remaining 2.5% parameter accuracy gap, reaching 100% across all three metrics. r=64 produces the same result as r=32, confirming that the additional trainable parameters provide no further benefit on this narrow format-learning task and that r=32 represents the effective saturation point. Third, Qwen3-8B adapts less cleanly to the training-format schema: from 0% base to 80% at r=16 across all three metrics. Only the r=16 checkpoint was produced for Qwen3 within the available pipeline runs; the 80% score indicates the adapter is partially effective but leaves a 20% residual of format non-compliance that further ranks or training iterations may or may not close. The gap between Llama FT and Qwen3 FT on this harness does not contradict Qwen3's higher clean LLMBenchmark score: the harnesses test different things and the numbers are not cross-comparable.

IV.2.7 Speech-to-Text Evaluation

Backend exploration. The STT backend selection went through two rounds of evaluation. The first round explored solutions available during early prototyping: the `SpeechRecognition` Python library (using its `recognize_google()` backend, which proxies audio to Google's Web Speech API) and Moonshine, a compact edge ASR model designed to run without a discrete GPU. The `SpeechRecognition` approach produced acceptable results

on clean, close-microphone English speech but required an active internet connection, which conflicted with the offline-first deployment requirement, and showed systematic errors on the alphanumeric PCB identifiers that are the core vocabulary of the system ("R12" was frequently returned as "are twelve" or similar). Moonshine eliminated the connectivity dependency but showed degraded accuracy under workshop background noise and limited Italian coverage, making it insufficient for the bilingual deployment target.

The second round evaluated two newer models that had become available after the initial exploration: Whisper Large-v3-Turbo and NVIDIA Canary-1B-v2. Both showed marked improvements over the first-round candidates. Whisper Large-v3-Turbo is a distilled variant of the Large-v3 checkpoint that reduces inference latency substantially while preserving the multilingual accuracy and the VAD-based noise suppression of the full model. NVIDIA Canary-1B-v2 was released specifically targeting industrial and noisy acoustic environments; its FastConformer architecture processes short local acoustic patterns that are less susceptible to broadband background noise, and its multilingual training covers Italian natively. Both models are fully self-contained and run without any network access on the DGX Spark hardware.

Both Whisper Large-v3-Turbo and NVIDIA Canary-1B-v2 were evaluated on representative utterances from the ARBoard interaction vocabulary, including clean speech, speech with background noise, and utterances containing PCB-specific shorthand identifiers such as "R1", "C16", "U5", and "SW2". Canary-1B-v2 was selected as the deployment backend. Its FastConformer architecture processes short local acoustic patterns that are less susceptible to broadband background noise than the longer-context Whisper attention mechanism, and its multilingual training covers Italian natively, which is a requirement for the bilingual deployment target. Whisper Large-v3-Turbo showed comparable accuracy on clean speech and remains a viable alternative for quieter environments; the primary differentiator for the workshop setting was Canary's noise robustness. Both models run fully offline on the DGX Spark hardware.

Text-to-Speech evaluation. The Microsoft Edge TTS backend was evaluated qualitatively alongside the STT experiments. Both the Italian voice (`it-IT-ElsaNeural`) and the English voice (`en-GB-SoniaNeural`) were tested on a set of representative assistant responses covering component location answers, net tracing descriptions, and type-based queries. Synthesis quality was assessed for naturalness, prosodic correctness on technical vocabulary (component identifiers, net names), and latency from text-to-audio-onset. All tested responses produced intelligible, natural-sounding speech. Technical strings such as "R12", "C16", and "SPI" were pronounced correctly without special handling. Language switching between Italian and English, driven automatically by the inference engine's language detection, was confirmed to select the correct voice in all tested cases. Audio playback began consistently within the interaction budget, allowing the spoken answer to start during the final stages of LLM generation rather than after it, which was confirmed through perceptual evaluation to remove any noticeable gap between the text appearing and the voice beginning.

IV.2.8 Post-Training Quantization

Post-training quantization was implemented via the NVIDIA ModelOpt NVFP4 export pipeline, which converts BF16 weights to a 4-bit floating-point format and calibrates activation scales on a small representative sample of the training data. The implementation is available in the codebase and can be applied to any fine-tuned checkpoint produced by the training pipeline.

Quantization was not applied in the current deployment configuration. The DGX Spark workstation is equipped with the NVIDIA Blackwell GB10 Superchip, which provides 128 GB of unified memory. Serving a Mistral-7B-Instruct-v0.3 checkpoint in BF16 precision requires approximately 14 GB for model weights, leaving ample headroom for the vLLM KV cache and concurrent application processes. On this hardware, BF16 serving is both feasible and preferable: it avoids the risk of accuracy degradation associated with lossy weight quantization and simplifies the deployment path, since no calibration dataset or export step is required between training and serving. The NVFP4 export capability remains available in the codebase for deployment scenarios where a more memory-constrained device is the inference target.

IV.2.9 System Integration and Qualitative Evaluation

The deployed system was evaluated qualitatively through live interactions on the DGX Spark workstation, using the best available fine-tuned checkpoint in BF16 precision and the selected STT and TTS backends. The following figures document the end-to-end experience as seen by the technician.

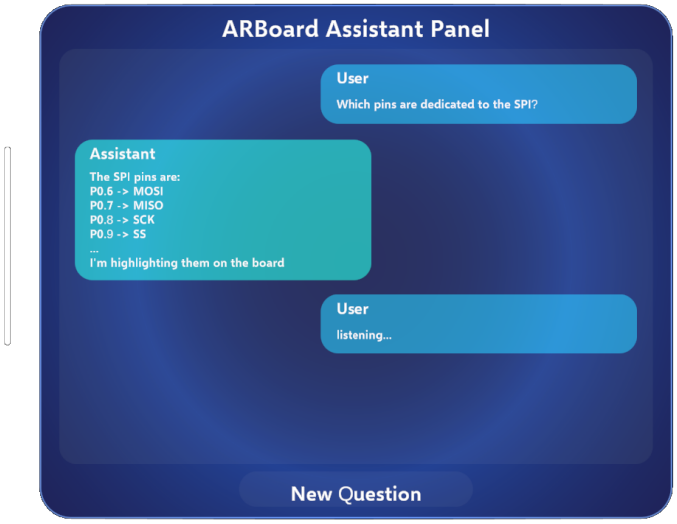


Figure IV.12: ARBoard Assistant Panel during a live SPI pin query.

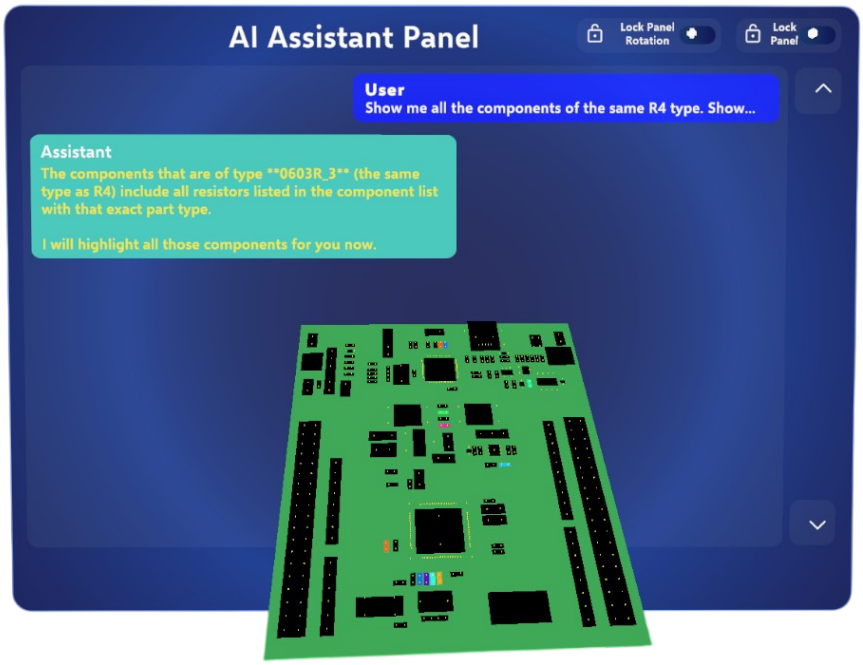


Figure IV.13: AI Assistant Panel in the AR board view, with queried components highlighted on the overlay.

Figure IV.12 illustrates the chat-style assistant panel as it appears during a spoken interaction. The panel displays the full conversation history, with user messages and assistant replies rendered in separate bubbles. The assistant's reply streams token-by-token into the panel as the model generates it, so the technician can begin reading the answer before generation is complete. The same text is forwarded to the TTS engine, which begins audio playback in parallel, delivering the reply as natural-sounding speech in the language of the question—Italian if the query was posed in Italian, English otherwise. This dual-channel delivery means that a technician whose visual attention is on the physical board still receives the full response through the audio channel.

Figure IV.13 demonstrates the system's end-to-end function in the AR environment. A type-based query causes the Reasoning Layer to return both a conversational answer listing the relevant components and a `highlight_by_type` tool call. The AR viewer receives the tool call and highlights the corresponding component footprints directly on the board overlay. The technician can therefore read the answer, hear it spoken aloud, and simultaneously see the queried components marked on the board—all without releasing the tools in their hands.

The following transcripts illustrate two representative interaction patterns observed during live evaluation: the negative-class response when no matching component exists on the board, and a net connectivity query that triggers a multi-net board visualisation.

Transcript 1 — Unresolvable identifier.

> **User (spoken):** *"Can you show me where You Fifteen is?"*
>
> **STT output:** *"Can you show me where You Fifteen is?"*
>
> **System answer:** *"I believe you are referring to U15. However, the current board does not contain a component with the reference designator U15. Please verify the identifier."*
>
> **Tool call:** `[]`

The STT engine again produces a phonetic transcription ("You Fifteen" for U15). The model correctly resolves the phonetic form to U15, but after querying the board context finds no matching component and returns an empty tool call array. No highlight is dispatched to the AR viewer. This behaviour is the direct product of the negative-class training examples included in both dataset variants, which train the model to refrain from hallucinating component identifiers that are absent from the context.

Transcript 2 — Net connectivity query.

> **User (spoken):** *"What is connected to JP5?"*
>
> **STT output:** *"What is connected to JP5?"*
>
> **System answer:** *"I am illuminating it on the board for you right now."*
>
> **Tool calls:**

```
[{"name": "highlight_net", "parameters":
{"net_name": "NetJP5_1"}},
 {"name": "highlight_net", "parameters":
{"net_name": "+5V"}},
 {"name": "highlight_net", "parameters":
{"net_name": "E5V"}}]
```

The model resolves all nets associated with JP5 from the netlist context and returns one `highlight_net` call per net, so all three connections are visualised simultaneously on the AR overlay. In this example, NetJP5_1 connects T2 and SB1, the +5V rail is shared with U4, C21, R32, C20, CN6, and CN7, and the E5V net connects D4, D3, R7, and CN7, allowing the technician to follow every electrical connection from JP5 without consulting a schematic. Figure IV.14 shows the result, with each net rendered in a distinct colour.

The two transcripts illustrate the operating envelope of the deployed system. Net connectivity queries (Transcript 2) complete without error when the target identifier appears in the board context window injected into the system prompt, and the model reliably selects the correct tool in that case. Phonetic ambiguity, as in "You Fifteen" for U15 (Transcript 1), is handled gracefully: the phonetic correction rules applied after STT transcription resolve standard alphabetic prefixes ("You" → U, "Arr" → R, "Cee" → C) before the string reaches the LLM. The negative-class training examples prevent the model from hallucinating a valid-looking tool call when no matching component is found in context; the

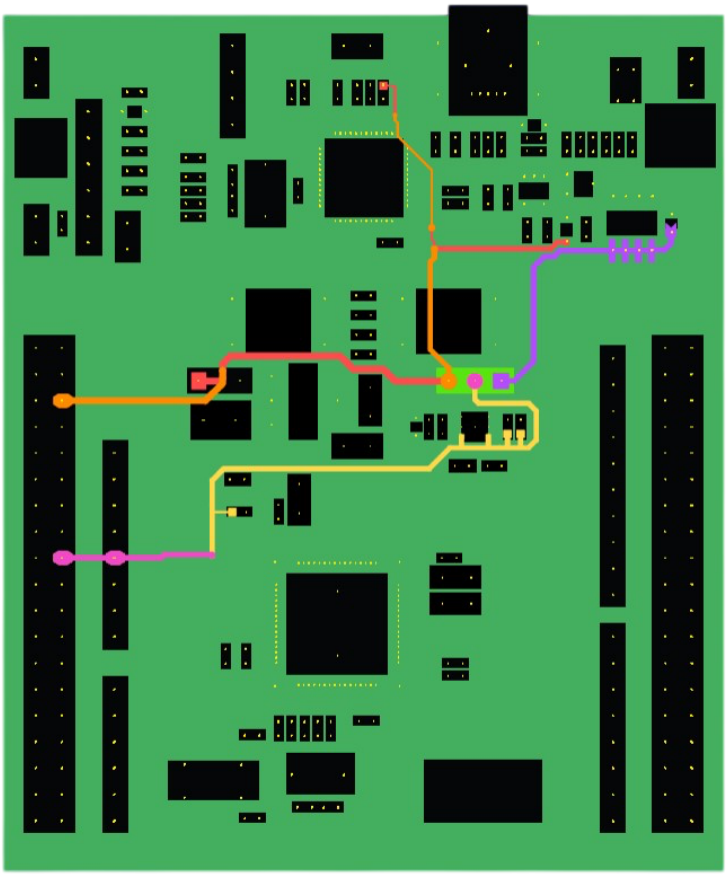


Figure IV.14: AR overlay generated in response to the JP5 connectivity query.

system returns an empty tool call array and does not dispatch a spurious highlight to the AR viewer.

The primary failure mode that remained after correction is context-window sampling. The system prompt includes a random subgraph of 150–190 components drawn from the full board database. This limit is necessary because the STM32 Nucleo board contains approximately 350 components: injecting the full component list and netlist at every turn would saturate the context window and degrade the model's ability to attend to the question itself. The current sampling strategy is uniform at random, which means that any individual component has roughly a 50% chance of appearing in the context on any given turn. When the queried component is absent from the sampled subgraph, the model behaves correctly in a narrow sense — it cannot produce a match that is not in its context — but the user receives an empty response that is indistinguishable from a true negative (no such component exists on the board). In practice this was observed for components located in regions of the board far from the current area of interest. The straightforward remedy is a spatially-aware or frequency-weighted sampling strategy that prioritises components near the most recent interaction or query, which would substantially

reduce context-miss failures without increasing token count.

A second residual failure mode involves compound identifiers that the STT backend does not resolve correctly even after phonetic rules are applied. Identifiers containing digits followed by letters (e.g. SB65, CN10) were occasionally transcribed as two tokens with an intervening space, causing a string mismatch against the database entry even when the correct designator was present in context. The phonetic correction rules implemented in the current version cover single-character prefix substitutions ("Are" $\rightarrow$ R, "You" $\rightarrow$ U, "Cee" $\rightarrow$ C) but do not handle multi-token compound identifiers. Extending the correction rules to cover common PCB identifier patterns — prefixes of two or more characters, numeric suffixes with trailing alphabetic characters — would address this class of failure without changes to the language model.

V. Conclusion

V.1 Summary of Contributions

This thesis set out to close the information-to-action gap that confronts technicians working with high-density PCBs in environments where structured design files may be unavailable and where sensitive intellectual property must not leave the local network. Two software contributions were designed, implemented, and evaluated to address this gap jointly: a Perception Layer that reconstructs a machine-queryable spatial database from legacy PDF documentation, and a Reasoning Layer that maps natural language and voice queries to deterministic AR tool calls using a domain-adapted local language model.

The Perception Layer addresses the scenario in which no IPC-2581 or equivalent machine-readable file exists. The pipeline ingests a PDF placement drawing or schematic, extracts component reference designators using a template-matching cascade with four geometric rejection gates, assembles individual character detections into complete label strings, reconciles detections across multiple documents and board sides, and writes a unified spatial database ready for AR overlay. The pipeline operates at two rendering resolutions to keep candidate extraction tractable while providing sufficient detail for cross-correlation matching. A three-rotation strategy (0°, 90°, 45°) and a final intersection-over-union suppression pass ensure that non-horizontally printed labels are recovered without introducing duplicate entries. An interactive template acquisition tool allows a technician to annotate a new document type in a single short session, after which the pipeline generalises to all pages of that document family without further supervision.

The Reasoning Layer provides the interactive intelligence that turns the database into a conversational interface. The architectural decisions that shaped the final system were made in a deliberate sequence: a preliminary survey of large frontier models established that the task is tractable and set an accuracy ceiling that a fine-tuned 8B model should try to approach; a deterministic synthetic dataset generator was designed to produce perfectly labelled training examples at negligible inference cost; a LoRA fine-tuning pipeline was built around the Unsloth framework with tool-call-only loss masking to focus gradient updates on the output format rather than the conversational text; and a shared evaluation harness was implemented so that baseline and fine-tuned models are compared under identical conditions. Speech input is handled by a dedicated STT backend and voice output by an Edge TTS engine, completing the hands-free interaction loop for technicians whose visual attention is on the physical board.

The thermal alignment module, added as a diagnostic extension, provides an independent demonstration of the spatial database as an analysis primitive. By computing a projective homography from four user-selected corner points, the

module registers a handheld infrared image to the board's coordinate frame and reports per-component temperature statistics automatically. The measured temperatures on the STM32 Nucleo validation board were physically consistent, with the primary ICs and the oscillator crystals at 29–35 °C and passive components away from the main power rail near ambient, confirming that the database coordinates are accurate enough to support quantitative thermal analysis without dedicated measurement hardware.

V.2 Findings

V.2.1 Perception Layer

The label extraction pipeline demonstrated that template matching, despite being a non-learned method, is competitive with trained OCR approaches on PCB documentation when the geometric rejection cascade is properly tuned. The four-gate cascade — scale, width validity, pixel mass, and Euler number — eliminates the majority of false candidates before the cross-correlation step is even reached, so the computational cost scales with the number of surviving candidates rather than with the raw density of contours on a dense schematic page. The Euler number gate was the single largest contributor to false positive rejection: solder pad annuli, which share a similar bounding box aspect ratio with characters such as O and C, carry a systematically different hole count and are therefore discarded at negligible cost before any image comparison is performed.

The three-rotation strategy addressed a failure mode that single-rotation approaches cannot resolve: a significant fraction of reference designators on placement drawings are printed at 90° to the board edge. On schematics, diagonal orientations near connector symbols add a further challenge that the 45° pass partially captures. Increasing to a fourth rotation pass was found to yield diminishing returns, since most remaining undetected labels at that point are either too small, overlap with circuit symbol geometry, or lie along board outline edges where the morphological line removal step inevitably removes adjacent strokes.

The cross-document resolution step is the component of the pipeline with the most direct impact on the quality of the final database record. A reference designator that appears on a schematic page but cannot be matched to any placement drawing is retained with schematic coordinates only, which is sufficient for net-tracing and connectivity queries but not for physical location queries. In practice, this situation arises for logical power and ground symbols that have no PCB footprint, which are a predictable and structurally well-defined class of exclusions.

V.2.2 Reasoning Layer

Task feasibility. The preliminary model survey confirmed unambiguously that the tool-calling task is solvable. Three distinct models reached perfect accuracy on the 40-question evaluation set in at least one run, and several others achieved above 90%. The result also confirmed the deployment constraint: every model that achieved above 90% accuracy has a parameter count of 20 billion or more, which places it above the memory and throughput envelope available alongside the rest of the ARBoard application stack on the DGX Spark. This finding makes the fine-tuning direction non-optional: deploying a large model directly, even if it were technically feasible, would either require dedicated serving infrastructure or cloud API access, both of which conflict with the offline-first, data-sovereign requirements of the target environment. The run-to-run accuracy variance observed in several frontier models — openai/gpt-oss-120b ranged from 35% to 90% across five runs — provides additional motivation for a fine-tuned local model, whose deterministic greedy decoding eliminates that variance by design.

Dataset design. The comparison of the teacher-model dataset and the deterministic v3 dataset reveals a fundamental trade-off between linguistic diversity and label correctness. The teacher-model approach generates richer question phrasings and covers implicit intents that template-based generators do not naturally produce, but introduces label noise at a rate proportional to the teacher model's error rate on the task. For a 4-class tool-calling evaluation with binary per-question scoring, even a small fraction of mislabelled training examples inflates the apparent training loss minimum without improving the model's actual accuracy. The deterministic generator produces provably correct labels at zero inference cost, and the convergence dynamics in training confirmed the difference: loss collapsed more rapidly and to a lower floor on the deterministic dataset.

Dataset variant and architecture study. The systematic comparison of six architectures under two dataset compositions (English deterministic and bilingual synthetic), evaluated with three diagnostic metrics, produced two concrete conclusions. First, Mistral-7B-Instruct-v0.3 is the only architecture that maintains performance after fine-tuning: its baseline tool selection accuracy (78.75%) is the highest in the group, and preliminary fine-tuning runs showed that it retains this accuracy after adaptation with the English deterministic dataset, while all other architectures show measurable degradation. The stability of Mistral is explained by its high baseline JSON validity (98.75%): the model already produces reliably structured output before any training, so LoRA fine-tuning reinforces rather than disrupts the existing format capability. Architectures with lower baseline JSON validity (Qwen3-8B at 30%, Phi-3.5-mini at 21.25%) showed no improvement regardless of dataset or rank, confirming that format compliance cannot be instilled through LoRA alone when the base model lacks it entirely. Second, the English deterministic dataset consistently causes less post-fine-tuning degradation than the bilingual synthetic dataset across every tested architecture and rank. The gap is most visible for DeepSeek-R1, which collapses to a constant low output

at all ranks under bilingual synthetic. The two mechanisms driving this difference are teacher-model label noise (the teacher is less reliable on Italian PCB queries) and output format inconsistency (the teacher's JSON style varies across languages, presenting a heterogeneous training signal). Together, these results directly motivated the configuration for the final fine-tuning campaign: Mistral-7B-v0.3 with the English deterministic dataset.

Fine-tuning results. The Phase 1 baseline sweep (Question 2) evaluated five architectures on the 40-question LLMBenchmark test set. Qwen3-8B achieved the highest One-Shot accuracy at 100%, followed by Mistral-7B-v0.3 at 90% and Llama-3.1-8B-Instruct at 77.5%. Phi-3.5-mini-instruct produced no valid tool calls, confirming it was not a viable candidate. The dataset variant study (Question 3) then identified Mistral-7B-v0.3 as the architecturally most stable model under fine-tuning: its in-process harness baseline tool selection accuracy (78.75%) is the highest in the group, and it retains 77.50% after fine-tuning at r=32, while all other architectures show measurable degradation. Despite Qwen3-8B's higher out-of-the-box LLMBenchmark accuracy (100%), its in-process harness baseline collapses to 30% JSON validity, indicating fragile distribution fit that does not survive the linguistic variation in the training-format test set; Mistral-7B-v0.3 is the correct selection for the Phase 2 fine-tuning campaign.

The LLMBenchmark Phase 2 evaluation of fine-tuned Mistral checkpoints was not completed within the thesis timeline due to a pipeline failure (training run interrupted before adapter save). The complementary in-process evaluation harness (in-process evaluation harness, n=40 held-out examples) was completed for Llama-3.1-8B at ranks r=16, r=32, and r=64, and provides the available quantitative answer to the fine-tuning and rank sensitivity questions. Training loss dynamics were consistent across all configurations: starting between 2.0 and 2.5 at step 1, descending below 10^{-3} within the first half of the epoch, with evaluation loss tracking closely throughout, confirming generalisation rather than overfitting. On the in-process harness, Llama FT r=16 achieved 100% JSON validity and tool selection accuracy with 97.5% parameter accuracy; r=32 and r=64 both reached 100% across all three metrics, with no further gain beyond r=32. The rank sensitivity result — saturation at r=32 with no benefit from r=64 despite a two-order-of-magnitude lower eval/loss — suggests the effective capacity limit for this narrow format-learning task lies at r=32. The 2.5% parameter accuracy shortfall at r=16 that closes at r=32 is the only measurable rank effect, and it is small.

Inference strategy. The One-Shot inference method was selected for all fine-tuning experiments. The preliminary survey confirmed that it produces extractable tool calls across all tested model families, and the deterministic dataset generator was built around its system prompt and output format, making it the only strategy for which supervised training examples could be generated without a complete redesign of the data pipeline. The Two-Stage method, which is deployed in the production ARBoard system for its latency advantage, uses a structurally different prompt and a different output schema for its first call; it was out of scope for fine-tuning within the available training budget, but represents a natural continuation of this work.

Speech and voice. The STT backend evaluation confirmed that both Whisper Large-v3-Turbo and NVIDIA Canary-1B-v2 are viable for the deployment scenario and represent a substantial improvement over the first-round candidates. Both operate fully offline, cover Italian natively, and show robust handling of PCB-specific technical identifiers. A formal quantitative WER comparison across clean, noisy, and technical-vocabulary conditions was not completed within the thesis timeline; Canary-1B-v2 was selected for the production deployment on the basis of its qualitative robustness advantage in noisy conditions and its native Italian support relative to Whisper Large-v3-Turbo. The TTS integration via Microsoft Edge TTS performed well qualitatively: technical identifiers such as R12, C16, and SPI were rendered correctly without special handling, and language detection for automatic voice switching between Italian and English was reliable across all tested responses.

V.3 Limitations

Several limitations bound the scope of the results reported in this thesis. They are documented here rather than minimised, because each one identifies a concrete direction for extending the work.

The Perception Layer pipeline was validated on a single board family (STM32 Nucleo MB1136) and its associated documentation. While the template acquisition mechanism is designed to generalise to new document types with a short annotation session, the cascade gate thresholds were tuned on this board's characteristic font size and line density. Documents with significantly different typographic properties — for instance, very small reference designators on a highly populated mixed-signal board, or hand-annotated PDFs with variable stroke width — may require threshold re-calibration before achieving comparable performance. In particular, the long-line removal step, which uses structuring element lengths derived from the expected character dimension, may remove character strokes or fail to remove sufficiently thick traces if the character-to-trace size ratio deviates substantially from the training assumption.

The fine-tuning experiments were conducted on a single board database. Although the dataset generator samples random subgraphs of 150–190 components at each example to prevent memorisation of a fixed layout, the set of component types, net naming conventions, and identifier formats is drawn entirely from one board family. A model fine-tuned on this distribution may exhibit degraded accuracy on boards with substantially different naming conventions or a different mix of footprint types. The generalization boundary between boards of the same family and boards with qualitatively different component vocabularies was not directly measured.

The evaluation metric is binary exact match after tool-call resolution. This is the correct metric for a deterministic command dispatcher, where a partially correct command is operationally equivalent to a wrong one — highlighting five of six correct components on a net trace gives the technician a misleading picture of the board. However, it is also a harsh metric for qualitative analysis:

a response that calls the correct tool with 90% of the correct component set is penalised identically to a response that selects the wrong tool entirely. A partial-credit metric alongside binary accuracy would provide finer diagnostic information about where fine-tuning improves performance and where it does not.

The post-training NVFP4 quantization pipeline was implemented via NVIDIA ModelOpt and is available in the codebase, but the DGX Spark's 128 GB of unified memory made quantization unnecessary for the deployed configuration: a Mistral-7B checkpoint in BF16 occupies approximately 14 GB, leaving ample headroom for the KV cache and concurrent application processes. The accuracy impact of quantization was therefore not measured on the current hardware target. If the system were ported to a more memory-constrained edge device, the export pipeline is ready to apply and the accuracy-versus-compression trade-off would need to be characterised on that hardware.

The STT evaluation was conducted under laboratory conditions, with workshop noise simulated rather than recorded in a real assembly environment. The sensitivity of both backends to the specific spectral profile of actual workshop noise — motor hum, ultrasonic cleaning baths, air tools — was not measured. The evaluation also used a fixed set of PCB identifier utterances; natural variation in speaker accent, speaking rate, and distance from the microphone was not systematically covered.

V.4 Future Work

The most immediate extension is to apply the fine-tuning pipeline to the Two-Stage inference method that is deployed in production. The current work fine-tunes for One-Shot, the only method whose output structure is directly compatible with the existing training data generator. A Two-Stage training dataset generator would need to produce examples aligned with the compact Stage-1 context format, and a corresponding evaluation harness would need to measure Stage-1 tool selection accuracy independently from Stage-2 answer quality. Fine-tuning specifically for Two-Stage is likely to produce higher production accuracy than fine-tuning for One-Shot, since the deployed inference engine uses Two-Stage exclusively.

A second extension is to complete the LLMBenchmark Q4 evaluation for the fine-tuned Mistral-7B-v0.3 checkpoints at r=32 and r=64. The in-process harness confirms that Mistral fine-tuning does not cause catastrophic forgetting of JSON format compliance, but it does not measure the accuracy improvement on the clean 40-question deployment test set. Running the LLMBenchmark evaluation on the saved adapters would directly quantify whether fine-tuning closes the 10% gap from Mistral's 90% baseline toward the 100% ceiling established in Question 1. The same evaluation could then be extended to all three inference methods (One-Shot, Single-Call, Two-Stage) to determine whether the ranking of methods changes after domain adaptation.

The dataset variant study identified the deterministic generator as superior

to the teacher-model approach for a fixed dataset size. A natural follow-on is to investigate whether using a higher-quality teacher model reduces the label noise gap. Using a frontier model that achieves near-perfect accuracy on this task as the teacher, rather than a compact local model, would produce fewer mislabelled training examples and might close the performance gap with the deterministic dataset while retaining the linguistic variety that template-based generation cannot reproduce. A complementary direction is to design a "noise-aware" bilingual generator that explicitly models phonetic confusions and language switching: for example, generating pairs of clean and corrupted queries ("R12" vs. "are twelve") with the same gold tool call, or sampling synthetic Italian/English mixtures with controlled proportions. Such a generator would allow the model to see realistic noise patterns under reliable supervision, mediating between the robustness benefits of bilingual data and the stability of the deterministic labels.

For cross-board generalisation, the fine-tuned model should be evaluated on boards from other families. The primary hypothesis is that board-agnostic improvements in output format compliance and tool selection logic transfer across boards even when component-specific knowledge does not. If format compliance transfers but component routing accuracy does not, a two-stage curriculum of generic format fine-tuning on a multi-board dataset followed by board-specific fine-tuning would be a more appropriate training strategy than the single-board approach used here.

The Perception Layer could be extended in two directions. The template matching approach could be augmented with a lightweight learned classifier at the final stage: a small CNN trained on surviving candidate ROIs, initialized on synthetic font images, would reduce character-level confusion on ambiguous glyph pairs (1 and I, 0 and O) without the brittleness of a fully learned OCR system. The pipeline could also be extended to accept rasterized frames from a camera pointed at a physical board, compensating for perspective, lighting variation, and partial occlusion, covering scenarios where no PDF schematic is available.

For the bilingual deployment, the correct strategy is to construct a larger dataset in which English and Italian examples are balanced at the question level rather than diluted from a fixed total. The question of whether a model fine-tuned on a balanced bilingual dataset matches the accuracy of a monolingual dataset of the same total size is tractable to measure and directly relevant to the Italian deployment target.

Bibliography

[1] IPC Consortium. IPC-2581: Generic requirements for printed board assembly products manufacturing description data and transfer methodology. *IPC-2581 Standard*, 2013.

[2] R. Brunelli. Template matching techniques in computer vision. In *Pattern Recognition: Theory and Applications*, 2009.

[3] E. J. Hu, Y. Shen, P. Wallis, et al. Lora: Low-rank adaptation of large language models. *International Conference on Learning Representations*, 2021.

[4] G. Insinga, F. Anzoino, P. Bella, P. Bernardi, L. Filippetti, and R. Khattar. ARBoard: Augmented reality for PCB operations in Industry 5.0. In *2025 IEEE International Symposium on Emerging Metaverse (ISEMV)*, pages 1–7, Honolulu, HI, USA, 2025. IEEE.

[5] R. Smith. Tesseract OCR engine, 2007.

[6] X. P. V. Maldague. *Theory and Practice of Infrared Technology for Nondestructive Testing*. Wiley-Interscience, 2001.

[7] P. Lewis, E. Perez, A. Piktus, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 2020.

[8] R. D. Hipp. SQLite, 2024.

[9] D. Han and M. Han. Unsloth: Efficient LLM fine-tuning, 2024.

[10] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[11] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems*, 2022.

[12] NVIDIA Corporation. NVIDIA ModelOpt: Model optimization toolkit, 2024.

[13] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. ACM, 2023.

[14] Useful Sensors. Moonshine: Fast and accurate automatic speech recognition, 2024.

[15] A. Radford et al. Whisper: Robust speech recognition via large-scale weak supervision, 2022.

[16] NVIDIA. NVIDIA canary, 2024.

[17] NVIDIA Corporation. NeMo: A scalable generative AI framework, 2024.

[18] McKie, J. and Liu, R. PyMuPDF: Python bindings for the MuPDF library, 2024.

[19] G. Bradski. OpenCV library, 2000.

[20] Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[21] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. Renard Lavaud, M. Lachaux, P. Stock, T. Le Scao, T. Lavril, T. Wang, T. Lacroix, and W. El Sayed. Mistral 7B. *arXiv preprint arXiv:2310.06825*, 2023.

[22] A. Dubey, A. Jauhri, A. Pandey, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[23] D. Guo, D. Yang, H. Zhang, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[24] Gemma Team. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

[25] M. Abdin, S. A. Jacobs, A. A. Awan, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

[26] Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2025.

[27] Microsoft Corporation. edge-tts: Python library for Microsoft Edge's online text-to-speech service, 2024.