Project Synthesis: Engineering a Robust Search Classification Pipeline for an Offline Retrieval-Augmented Chatbot

Designing a privacy-first, confidence-aware, and fault-tolerant AI system through calibrated LLM labelers, distributed Docker infrastructure, and BERT fine-tuning.

0.0 Executive summary

This project set out to design and implement a **reliable, private, and fully offline chatbot system** capable of answering user queries without relying on the internet. Unlike typical cloud-based models, this system guarantees **privacy** by operating entirely on local hardware while still maintaining **accuracy and efficiency**.

The solution combined three core innovations:

1. **Dynamic Search Classification** — A custom classifier decides when external lookup is needed, ensuring the chatbot balances **speed with factual accuracy**.
2. **Confidence Calibration Pipeline** — A novel multi-stage process transformed unreliable outputs from lightweight models into **trustworthy, reusable training data**.
3. **Distributed Labeling & Fine-Tuned BERT** — A fault-tolerant infrastructure labeled and audited over 200,000 examples at scale, enabling the development of a **production-grade BERT classifier** that achieved ~90% accuracy in its first training epoch.

Together, these components created a **blueprint for building data-centric AI systems** that are not only performant but also **robust, reusable across domains, and resilient under real-world constraints**.

1.0 Project Mandate and Strategic Context

This project set out to engineer a **reliable, private, and fully offline chatbot** built on local large language models (LLMs). It was conceived as a direct response to the *“catastrophic failure”* of an earlier system that depended on live web-scraping — an approach that proved brittle and unreliable once external dependencies broke.

The strategic imperative was clear: create a **self-contained system** that guarantees consistency and privacy by eliminating reliance on external networks. However, local LLMs — especially smaller, quantized models that can run on consumer hardware — face two core challenges:

- They are prone to **hallucinations** when pushed beyond their training scope.
- Their knowledge is **static and quickly outdated**, as it is locked at the point of pretraining.

To address this, the solution was to augment a local LLM with an **offline Wikipedia knowledge base** (over six million articles in the English XML dump, ~100 GB). This retrieval layer grounds the model's outputs in factual data, enabling accurate, privacy-preserving responses.

From the outset, the project defined five guiding objectives:

- **Accurate Information Retrieval:** Efficiently search the local Wikipedia index to extract relevant facts for any given query.
- **Effective Question Answering:** Use a lightweight LLM to understand the user's intent, synthesize retrieved content, and deliver clear, accurate answers.
- **Resource Efficiency:** Ensure the system runs on a typical laptop CPU (no GPU required), with reasonable performance (targeting ~1 minute per query for full retrieval + answer).
- **Dynamic Decision-Making:** Develop a classifier to decide, on a per-query basis, whether a Wikipedia lookup is necessary — optimizing both speed and relevance. (This classifier became the central focus of the project and the narrative core of this report.)
- **Large Context Handling:** Build mechanisms, such as summarization, to condense long Wikipedia passages so they fit within the LLM's limited context window.

At its core, the **query classifier** functions as the system's **gatekeeper** — balancing efficiency, accuracy, and reliability. Engineering this component successfully was key to unlocking the project's mandate of **privacy-first, production-grade performance.**

2.0 System Architecture: The Query Classification Pipeline

The strategic core of the chatbot's architecture is the query classification module. This component acts as an intelligent "gatekeeper," deciding precisely when to leverage the offline Wikipedia knowledge base. By dynamically routing queries, it ensures that simple questions are answered instantly using the LLM's internal knowledge, while complex or fact-specific questions trigger a retrieval process to guarantee accuracy. This prevents unnecessary search latency and avoids cluttering answers with extraneous information. Which can especially cause small parameter models to hallucinate or "lose the plot".

The complete system operates through a modular, multi-stage pipeline that processes each user query from input to final answer:

1. **User Query Input:** The process begins when a user submits a question in natural language.

2. **Input Preprocessing:** User input is preprocessed to cut down noise and help with classification. Pulling our relevant entities to aid with search classification as well as downstream model prompting.
3. **Query Classification Module:** The system's first step is to analyze the query and make a binary decision: is a Wikipedia lookup required? This module outputs its decision along with a confidence score.

***Note: The following is a description of future blueprint and has not yet been implemented but is important to understand certain design decisions.**

4. **Wikipedia Search & Retrieval Module:** If the classifier deems a search necessary, the query is passed to a local search index built from an offline Wikipedia dump. The module retrieves the most relevant article text.
5. **Context Management Module:** If the retrieved text is too long for the main LLM's context window, a lightweight summarization model condenses it to its essential points.
6. **LLM Answer Generation Module:** The main LLM receives the original user question, along with the retrieved and summarized context (if any), and generates a comprehensive, contextually-grounded final answer.

2.1 Early Iterations and Roadblocks

The initial implementation of the Query Classification Module proved to be a significant engineering hurdle, revealing issues of model overconfidence, inconsistency and crucially performance. This necessitated a multi-phased effort to build a classifier robust enough for production use and fast enough to be negligible for input processing.

The previous project used a large language model as a classifier, hoping to rely on the pre-trained encoded data as enough to classify searches with given example examples within the system prompted within building a prompt, however that seemed to fall apart quickly, and the performance was significantly worse than expected. As such the goal of the search classification aimed to optimize for speed as much as possible. As such dependency parsing was chosen.

In the earliest stages of development, the system experimented with dependency parsing using spacy as a lightweight way to extract semantic structure from queries. Dependency parsing maps words in a sentence to their grammatical roles — such as subjects, objects, and modifiers — creating a directed graph of relationships between terms. For example, in the question “What is the capital of France?”, the parser identifies “capital” as the subject, “is” as the root verb, and “France” as the object.

The motivation was to heuristically derive intent from these grammatical relationships without requiring a full transformer model. For very short, single-sentence inputs, this worked reasonably well. However, the approach quickly broke down with longer or multi-sentence queries, where parsing inconsistencies and user input variability caused unstable results.

As a result, dependency parsing was phased out after these initial trials. It was ultimately replaced by entity recognition and tagging as the main structured feature source, since entity spans provided more reliable signals for distinguishing between general reasoning questions and those requiring factual lookup.

The benefits of using dependency parsing would have been near instant (in terms of modern compute) classification as it did not rely on any advanced encoding as was based simply or rules based on the sentence structure. That did not work so more traditional ML methods where explored.

2.2 Traditional ML Baselines

We next revisited proven text classifiers such as **k-Nearest Neighbors, Naive Bayes, and Logistic Regression**. In prior biomedical applications, Logistic Regression with TF-IDF features achieved 97.3% accuracy on long articles. But in this project, queries averaged just 9–19 words. Sparse token-based features failed to capture user intent, making these methods underpowered for short, conversational text.

This pointed us toward transformer architectures, which excel at capturing semantic meaning rather than relying solely on token frequency.

2.3 Expansion into Transformers

Transformers are highly effective for semantic understanding in NLP because they process entire input sequences simultaneously, rather than token by token. This parallel processing, enabled by their self-attention mechanism, allows them to weigh the importance of all other words in a sentence when encoding each word. This global context capture enables models to grasp nuanced relationships and dependencies between words, making them adept at understanding the true meaning and intent of a query, which is crucial for tasks like classification. One of the main hold backs against going the transformer route from the getgo was the need for finetuning data. The hope was the pertained “encoded” data of an LLM would be sufficient to yield accurate classification results, it however was not.

Because of this, further research into transformer architectures for binary classification was conducted, focusing on the following criteria:

- **Speed**: How efficiently the model performs inference, especially on CPU-bound or edge hardware.
- **Accuracy**: Ability to capture nuance in short queries and produce reliable, calibrated predictions.
- **Compatibility**: Suitability for integration into a lightweight, offline pipeline with confidence calibration.

Transformer Options Considered

1. GPT-style (Decoder-only) Models

- **Speed**: Typically slower for classification tasks due to their left-to-right architecture.
- **Accuracy**: Strong generative abilities, but less efficient for classification where bidirectional context is valuable.
- **Compatibility**: Require more memory and compute; not optimal for short binary decisions.

2. XLNet

- **Speed**: Comparable to BERT, but with higher training complexity.
- **Accuracy**: Outperforms BERT in some benchmarks by using a permutation-based training objective that captures bidirectional context without masking.
- **Compatibility**: Computationally heavier; inference latency is higher, making it less practical for CPU-only environments.

3. ELECTRA

- **Speed**: More efficient than BERT in pretraining, but inference speed is similar.
- **Accuracy**: Excellent performance on binary classification due to its replaced-token detection objective, which trains the model to distinguish fine-grained differences.
- **Compatibility**: Strong contender, but less mature ecosystem and fewer calibration studies compared to BERT.

4. T5 (Text-to-Text Transfer Transformer)

- **Speed**: Slower, as all tasks are reframed into text generation.
- **Accuracy**: Highly flexible and accurate across tasks, but overkill for simple classification.
- **Compatibility**: Requires more compute and memory; inefficient for constrained setups.

5. DeBERTa (Disentangled Attention)

- **Speed**: Slower than BERT due to more complex attention mechanics.
- **Accuracy**: State-of-the-art for many classification benchmarks. Captures subtle semantic distinctions extremely well.
- **Compatibility**: Best for GPU-backed systems, less ideal for CPU-only deployments.

6. BERT (Bidirectional Encoder Representations from Transformers)

- **Speed**: Balanced runtime efficiency; slower than distilled variants but fast enough for CPU inference on short queries.
- **Accuracy**: Strong semantic understanding of short text and robust performance on classification tasks. Handles nuanced intent better than lightweight models.
- **Compatibility**: Mature ecosystem with extensive tooling and calibration research. Easy to fine-tune with entity tags and soft labels, making it highly adaptable for production pipelines.

Why BERT Was Selected

Although architectures like XLNet, ELECTRA, and DeBERTa can edge out BERT in absolute accuracy, they impose heavier computational costs or are less stable in calibration. Decoder-only models (e.g., GPT-style) are strong generators but inefficient classifiers. T5 is highly general but wasteful for binary outputs.

BERT strikes the right balance:

- Strong semantic accuracy on short queries.
- Moderate inference cost suitable for CPU-bound systems.
- Rich ecosystem and tooling, making it easy to integrate entity tagging, calibration, and confidence-aware training.

And over the previous LLM based classifier

1. **Scalability and Data Utilization:** The BERT classifier was designed to be fine-tuned on the massive dataset of 200,000 labeled examples generated by the LLM labeling system. BERT and its variants (RoBERTa, DeBERTa, etc.) are highly effective models that generalize well when fine-tuned on large amounts of labeled data.
2. **Suitability for Short Queries:** The training data indicated that most user queries are short texts (average word count for "Search (1)" is about 9 words, and "No Search (0)" is about 19 words). Furthermore, most queries are short and BERT-suitable (typically under 128 words/tokens).
3. **Stability and Calibration:** The LLM's primary weakness was its unstable and overconfident raw confidence scores. The LLM's purpose was not necessarily to have perfectly accurate confidence scores itself, but to establish the *relation between confidence scores* to provide high-quality training data for BERT. Fine-tuning BERT on this calibrated data allowed the final classification model to achieve robust semantic understanding without needing real-time confidence extraction via complex LLM prompting.
4. **High Throughput:** While the LLM was slow for classification, the BERT classifier, running on a GPU node in the distributed labeling setup, demonstrated efficient throughput (e.g., processing 812 questions in approximately 4 minutes on a single RTX 3060 GPU). This efficiency is critical for production use.

For these reasons, BERT was selected as the final classifier, combining proven performance with practical deployability.

2.4 Processing input for BERT classification

Before user queries are passed into the BERT classifier, they undergo a carefully designed preprocessing pipeline. This step ensures that the input is clean, consistent, and structured in a way that maximizes classification accuracy while retaining the nuances needed for semantic interpretation.

Normalization

All text is converted to lowercase to remove case sensitivity issues. Filler words such as "hey," "like," or "um" are stripped, along with most punctuation except for question marks, which often signal query intent. This process reduces noise without losing semantic content. All abbreviations are mapped to a CSV that expands them out, ie LOL to laughing out loud.

Entity Recognition and Markup

Named entities such as organizations, products, and geopolitical entities are extracted using a lightweight spacy recognition model. These entities are wrapped in <ENT> tags and re-inserted into the query, giving the classifier explicit cues about important spans of text. For example:

"Who runs Tesla?" → "who runs tesla?<ENT> Tesla Company<ENT>"

This improves the model's ability to distinguish between factual lookups requiring a search and more general reasoning tasks. All entity recognition is done before normalisation to ensure lowercasing names does not lead to lost context. Entities are append first based on there index within the input. So if a company comes first than a place that is referenced first.

Sequence Length Management

Since BERT has a maximum input length of 512 tokens (roughly 650 characters), special handling is required for longer queries. Instead of discarding or truncating them, the system is designed to fall back to a lightweight LLM module. This fallback is not yet implemented but is planned as part of the broader architecture to handle out-of-bound inputs gracefully.

Why Not Lemmatization

Unlike traditional NLP pipelines, lemmatization is avoided here. Surface form distinctions (e.g., "running" vs. "runs") can signal differences in query intent. By preserving these variations, the classifier retains subtle cues that are useful in deciding whether an external search is required.

3.0 Phase 1: Data sources and Labelling Strategy

Because BERT was chosen as the final classifier, the new challenge of creating and labeling training data was on the horizon. Unlike lightweight LLM prototypes that could rely on few-shot prompting, BERT requires a large, consistently labeled dataset to adapt its pre-trained representations to the specific task of determining search necessity.

This stage of the project focused on building a scalable and reliable pipeline for generating high-quality training data. Two core requirements guided this effort:

- Volume: Tens of thousands of examples were needed to capture the diversity of user queries across domains and contexts.
- Quality: Labels had to not only specify whether a search was required but also encode calibrated confidence scores, allowing BERT to learn from both discrete classifications and the relative certainty of those decisions.

Meeting these requirements meant moving beyond manual annotation. Instead, lightweight LLMs were leveraged as automated labelers, supported by a calibration pipeline to correct their systematic overconfidence. The outputs formed the backbone of the dataset used to fine-tune BERT, transforming noisy heuristic labels into a structured, production-ready resource. But firstly the issue of collecting all the data needed to be solved.

3.1 Data Sources and Collection Strategy

Building a high-quality dataset required combining **diverse external corpora** with **raw real-world queries** to balance scale, realism, and robustness. This approach ensured the classifier was exposed to both clean, curated examples and noisy, human-like inputs that reflected actual usage conditions.

External Domain-Specific Sources

- **Wikipedia Question Sets** – Approximately 30,000 annotated questions were extracted, cleaned, and filtered, with only the raw question text retained for consistency. While not tailored to search classification, they provided a broad factual baseline.
- **Medical Journals** – Selected datasets of medical questions were integrated. These were **hardcoded to “search”**, since answering them reliably requires factual retrieval beyond what any lightweight LLM can provide.
- **Stack Overflow** – Technical Q&A data was incorporated to cover programming-specific inputs, a domain where entity grounding and external factual checks are especially critical.

Exported User History

The most novel data source came from **personal ChatGPT user history**. By exporting and parsing the raw JSON schema, hundreds of thousands of natural queries were collected. Unlike curated datasets, these queries included:

- **Spelling mistakes, typos, and informal phrasing**
- **Fragmented or conversational question styles**
- **A wide variety of entities and intent signals**

This user-generated data injected realism into the dataset, ensuring the classifier would not only perform well on clean benchmarks but also on the messier queries users actually produce.

Data Cleaning and Processing

Custom pipelines were built for **CSV parsing** and **JSON extraction**, designed to stream data in manageable batches rather than load everything into memory. This lightweight but robust approach allowed scaling across very large files without requiring specialized infrastructure. These pipelines were intentionally domain-specific and single-use — simple tools designed for efficiency, not generality.

Domain-Specific Labeling Rules

To reduce ambiguity and improve robustness, certain classes of questions were **explicitly hardcoded**:

- **Medical questions → Always "search"** (external retrieval required).
- **Mathematics questions → Always "no search"** (logic-based, self-contained).

Outcome

By combining **curated sources**, **domain-specific hard rules**, and **raw real-world queries**, the project produced a dataset of roughly **320,000 examples** ready for labeling. This mixture provided both **breadth** (coverage across domains) and **authenticity** (reflecting how users actually ask questions), giving the downstream BERT model a strong foundation for fine-tuning.

3.2 Selecting LLMS for data labeling.

The first engineering challenge was to validate whether a **lightweight LLM-based approach** could reliably classify queries on CPU-only hardware. More importantly, this stage tested whether such models could be scaled to automatically label the entire dataset for BERT fine-tuning, eliminating the need for manual annotation.

The objective was to build a **lightweight but accurate search-needed classifier** that could:

- Run efficiently on CPU-bound environments (e.g., 4–6 cores, optionally with GPU support).
- Produce consistent JSON outputs in a strict schema (`{"search_needed": 0|1, "confidence": float}`) for downstream BERT fine-tuning.
- Handle short, independent user queries without context memory.

- Flag ambiguous classifications: confidences below 0.5–0.4 were marked for manual review.

Constraints:

- Input length capped at 512 tokens (~650 characters).
- Low tolerance for formatting errors or hallucinations.
- Preference for models optimized for classification rather than open-ended chat, to avoid verbose or inconsistent outputs.

To meet these requirements, the **Ollama runtime** (a wrapper around Llama.cpp) was selected as the execution environment. Ollama provided consistent cross-platform support (critical for distributed compute on Linux/CUDA nodes) and simplified deployment significantly.

Models Selected for Evaluation:

Model	Parameters	Notes
Qwen2.5:0.5b-instruct	0.5B	Smallest footprint; strong contextual performance for its size.
Granite3.1-MoE:1b	1B	Mixture-of-experts variant; efficient but uneven across domains.
Granite3.3:2b	2B	Higher baseline accuracy but slower latency.
Llama3.2:1b	1B	Stable general-purpose model; moderate latency.
Phi-4 mini	3.8B	Most capable; significantly slower, high memory cost.
Falcon3:1b	1B	Balanced performance; smaller ecosystem support.

All models were quantized to 4 bits for efficiency.

This initial evaluation phase established the trade-offs between **latency, accuracy, and robustness**, setting the stage for calibration and distributed labeling.

3.3 Testing Methodology

To fairly evaluate candidate models for the labeling pipeline, a structured test suite was developed around the gold standard class of 60 labeled questions. This suite ensured that each model was benchmarked consistently, producing not only accuracy results but also efficiency and stability metrics that reflect real-world usage.

Test Suite Execution

All candidate models were run against the full set of labeled questions. Each output was compared against the gold standard, with results logged both globally and by domain. This allowed direct comparison across models and domains such as programming, math, medical, mental health, and general knowledge.

Metrics Collected per Run

- **Latency (wall-clock time)**: Measured end-to-end generation time, including Ollama's serving overhead.
- **CPU Time**: Captured the active compute time required, highlighting whether bottlenecks were caused by raw compute versus memory bandwidth limitations.
- **Accuracy (Discrepancies)**: Logged as mismatches between the model output and the gold standard label. For search classification only not confidence score float.
- **Confidence Statistics**: Tracked the average and variance of model-reported confidence scores, revealing the systematic overconfidence problem common in small LLMs.
- **Error Counts**: Recorded JSON parse failures, since the labeling pipeline required outputs to conform to a strict schema. In later stages this was reduced to 0.
- **Domain-Level Performance**: Discrepancies were broken down by domain to identify which models generalized well and which were specialized.

Automation and Logging

All results were written to CSV logs in real time. This enabled automated aggregation and post-hoc analysis without rerunning benchmarks. From these logs, visualizations were generated.

Outcome

This methodology provided a multi-dimensional view of performance, balancing speed, accuracy, and reliability. By automating logging and visualization, the test suite created a repeatable, data-driven foundation for selecting the optimal model for the labeling pipeline.

3.4 Building a robust labeler

Prompting

Started with generic prompts requiring strict JSON outputs as in earlier iterations model drift was a significant issue. Then they were iteratively refined to emphasize "no reasoning, JSON only", reducing model drift. Added domain-specific few-shot examples (medical, programming, general, mental health). Later introduced balanced two-shot prompts based on input domain, (1 yes, 1 no), selected by question length, to prevent bias toward trivial or complex examples. Bellow is a summary of the findings

The **system prompt** defined how each lightweight LLM interpreted and responded to incoming queries. Because the classifier's reliability hinged on consistent behavior, prompt engineering became a critical component of the pipeline.

Core Elements of the Prompt
All prompt variants tested included:

- **Role framing**: Affirmations positioning the model as "highly qualified" and "highly accurate," reinforcing the expectation of precision.
- **Heuristic rules**: Explicit guidelines describing when a query should trigger a search and when it should not.
- **Entity awareness**: Examples demonstrating both raw queries and queries with `<ENT>` tags, ensuring the model recognized entity spans as key features.
- **Rationale for decisions**: Instructions clarifying the *why* behind classification, not just the binary outcome.
- **Pre-question prompting**: Each user query was introduced with a framing instruction, guiding the model to respond within the strict schema.

Shot Variations

- **One-shot prompting** (one example of each class: *search* and *no search*) yielded the best results. Outputs were more homogeneous and aligned closely with gold-standard labels.
- **Two-shot prompting** (two examples per class) increased verbosity and inconsistencies, with diminishing returns.

Confidence Calibration via Prompting
Raw outputs across all models were systematically overconfident, with reported confidences consistently above 90%—even for incorrect classifications. To address this, two interventions were tested:

1. **Artificially lowering example confidences** in the prompt (e.g., showing exemplar outputs with 0.65–0.75 confidence instead of >0.9).
2. **Framing the model as “accurate but unsure”** in its self-assessment, nudging it toward more realistic probabilities.

These adjustments produced a healthier distribution of confidence scores, averaging around **72% with high variance**—a desirable outcome that better reflected classification uncertainty. This diversity of confidence values laid the groundwork for the post-processing calibration pipeline that followed.

Options Sweeps

In addition to evaluating model architecture, a critical part of the methodology was testing how different inference settings affected performance. Small LLMs are particularly sensitive to sampling parameters, and slight changes in temperature or top-k (in layman's terms how “creative” model responses are) filtering can cause large swings in both accuracy and stability. To account for this, an options sweep was designed to benchmark models under two contrasting parameter regimes: Conservative and Liberal.

Experimental Setup

Each model was run for 10 trials across the gold-standard test class, split evenly between the two regimes (5 conservative, 5 liberal). Random seeds were fixed at 42 for reproducibility. Constants across all runs included:

- repeat_penalty=1.1 to discourage repetitive outputs. (context was **NOT** preserved across classifications)
- num_predict=22 to keep responses short and schema-constrained.
- format="json" and raw=True to enforce strict adherence to the output schema required by the labeling pipeline.

Parameter Regimes

- **Conservative Regime:**
  - temperature=0.0–0.3
  - top_k=1–5
  - top_p=0.5–1.0

This regime heavily constrained sampling, encouraging deterministic outputs. The aim was to maximize consistency with the JSON schema and reduce random drift, at the cost of flexibility in borderline classification cases.

- **Liberal Regime:**
  - temperature=0.4–1.0
  - top_k=10–60
  - top_p=0.1–0.5

This regime allowed for more diverse generations, encouraging the models to explore alternative token paths. The goal was to test whether a looser configuration could improve semantic reasoning and reduce discrepancies on ambiguous queries, even if it increased the risk of malformed outputs.

3.5 Testing results and conclusions

The evaluation produced several key insights into model performance, surfacing trade-offs between **accuracy, speed, and reliability**. The charts below summarize aggregated results across all test runs. Before interpreting the figures, it is important to establish context and clarify definitions:

- **Discrepancies (vs. gold labels):**
  A discrepancy is recorded whenever a model's classification ("search needed" or "no search") differs from the gold standard labels. Confidence scores are *not* factored into this metric; only the correctness of the binary decision is measured here.

- **Domain coverage:**
  Not all domains tested will appear in the final production labeling pipeline. For example, **mathematics queries** are always hardcoded as *no search* because they can be resolved with pure logical reasoning. However, evaluating models across diverse domains provided a broader understanding of their *generalization ability* and revealed how they behaved outside of their "expected" workload.

- **Confidence interpretation:**
  Raw confidence values should be viewed with caution. Smaller LLMs tend to dramatically overstate certainty due to their limited representational nuance. In practice, we observed models regularly assigning >90% confidence to incorrect outputs. This overconfidence is *not corrected* in the figures below—subsequent calibration stages address this issue separately.

Taken together, the results illustrate how different architectures balance **latency vs. accuracy**, how their overconfidence patterns vary by domain, and why calibration and auditing were non-negotiable steps in making the system production-ready.

Figure 1 — Average Discrepancies per Model (Lower is better).

This chart compares the average number of discrepancies for each model. Lower bars indicate higher accuracy.

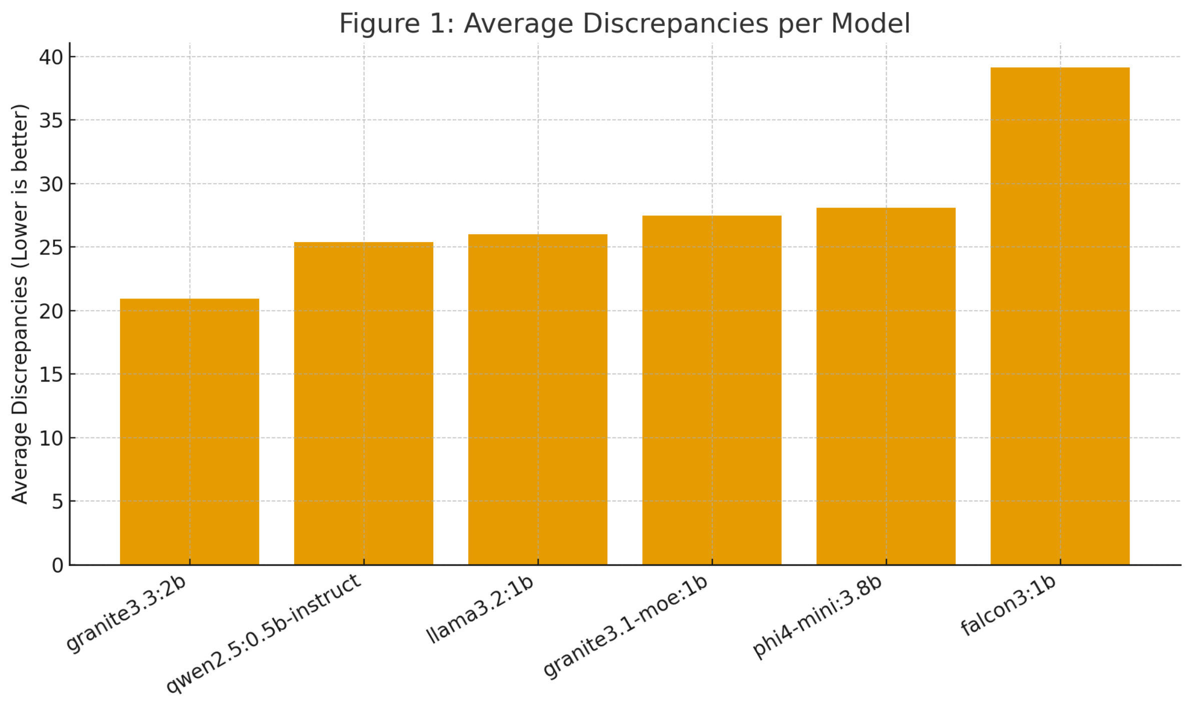
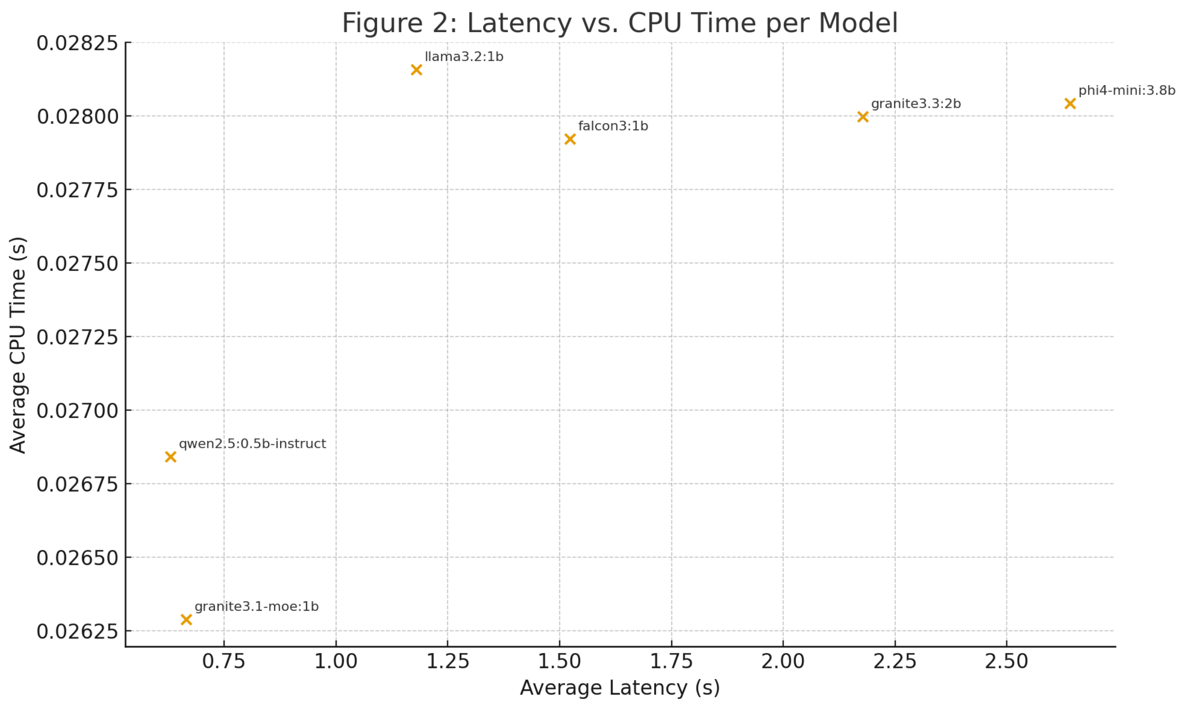


Figure 2 — Latency vs. CPU Time per Model (bottom-left is most efficient).

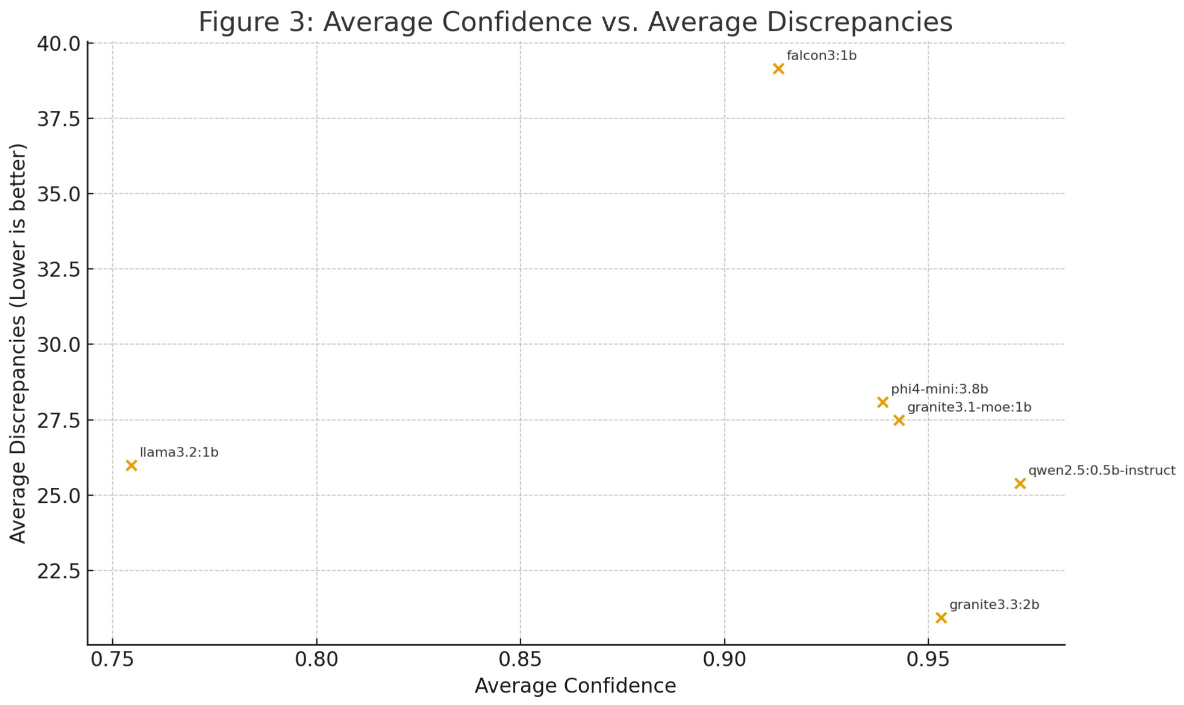
This plot visualizes the trade-off between latency (speed) and CPU time for each model. Models in the bottom-left are the most efficient.



*side note: CPU bound LLMS are often memory bandwidth constrained measuring latency (time to generate fundamentally) vs CPU time (time spent computing when on CPU like here) gives us a rough idea of how adding more compute would effect our results higher cpu time vs lower latency indicates benefits with adding more compute while higher latency to cpu time indicates less benefits in adding more compute (more/faster cores) to get beyond this gpu processing or faster memory would be necessary.

Figure 3 — Average Confidence vs. Average Discrepancies (bottom-right is ideal).

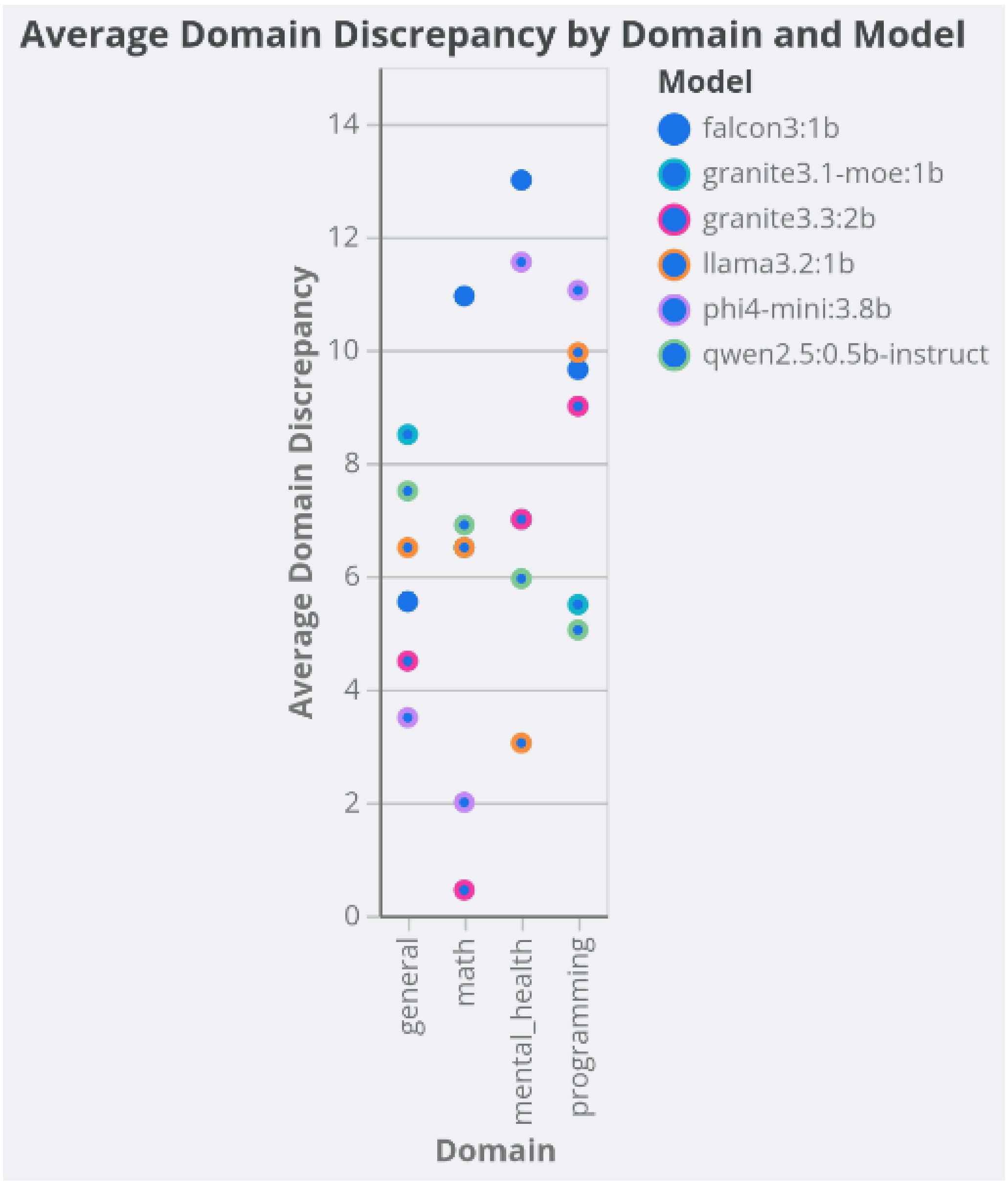
This scatter plot shows the relationship between a model's self-reported confidence and its actual accuracy (discrepancies). An ideal model would appear in the top-left quadrant.

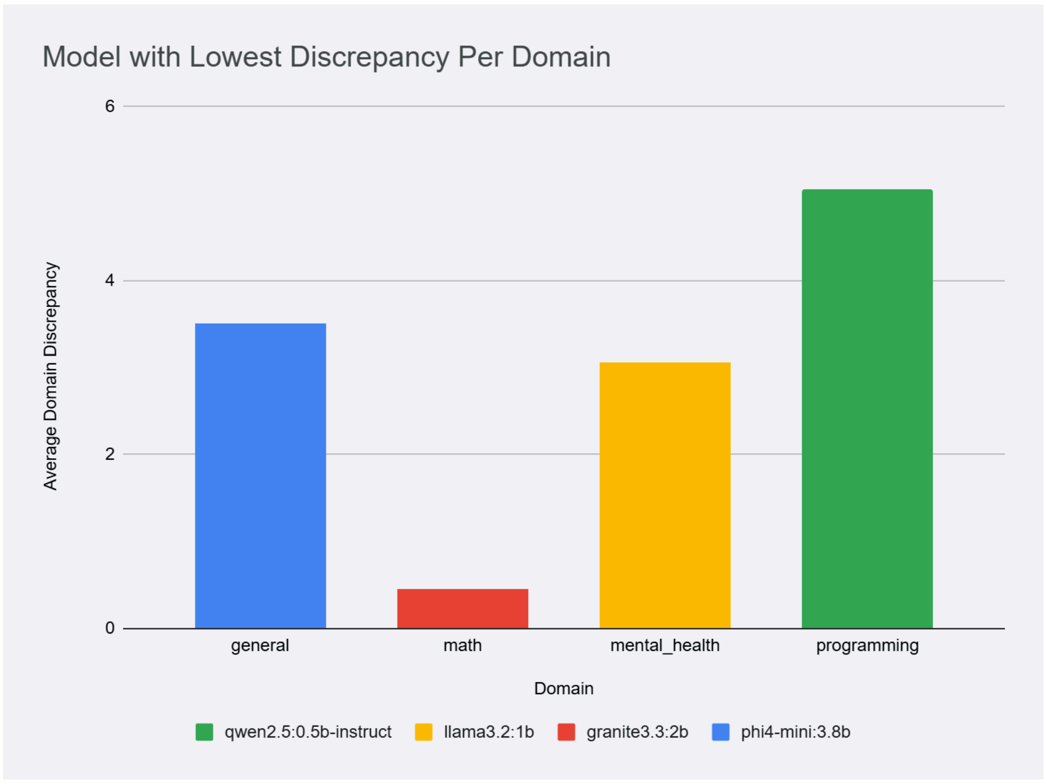


Note* while higher discrepancies are worse because this is a natural language task there is no fundamentally correct answer so lower discrepancies is better but no discrepancies is not necessarily best.

Figure 4 — Domain-Level Discrepancy Rate per Model (stacked by domain).

This stacked bar chart breaks down the discrepancy rate for each model across different domains (e.g., math, programming). This helps identify strengths and weaknesses in specific areas.





The table includes a summary of the findings and a more detailed explanation bellow

Model	Parameters	Avg. Latency (s)	CPU Time Profile	Confidence Behavior	Avg. Discrepancies	Domain Strengths	Domain Weaknesses
Qwen2.5:0.5b-instruct	0.5B	~0.6s	Stable, low CPU	Overconfident (~0.9–1.0) but consistent	~25	Strong in programming & general knowledge	Weaker in math, some edge cases in health
Granite3.3:2b	2.0B	~2.2s	High CPU usage	Inflated confidence (~0.95)	~21	Best in math domain, stable overall	Latency too high for CPU-only scaling
Phi-4 mini:3.8B	3.8B	~2.6s	Very high CPU	Overconfident, clustered high	~27	Strongest semantic reasoning, nuanced queries	Prohibitively slow, high memory demand
Llama3.2:1b	1.0B	~1.2s	Moderate CPU	Moderately overconfident	~29	Best in mental health queries	Less consistent in math & programming
Falcon3.1b	1.0B	~1.5s	Spiky CPU demand	Overconfident, unreliable	~39	None notable	Highest discrepancies, poor schema adherence
Granite3.1-moe:1b	1.0B	~1.1s	Low CPU overhead	Unstable, inconsistent	~32	Occasional strength in short factual queries	Unstable results across domains

The testing campaign provided a comprehensive picture of how different lightweight LLMs performed as automated labelers, revealing both their strengths and limitations. The figures and logged metrics allow several key takeaways to be drawn.

1. Discrepancies and Domain Specialization

The discrepancy analysis showed that no single model was universally dominant across all domains. Granite3.3:2b excelled in mathematics, Llama3.2:1b showed the strongest performance in mental health, and Qwen2.5:0.5b-instruct consistently led in general knowledge and programming. This reinforced that domain context plays a significant role in classification reliability, but also highlighted Qwen2.5's versatility as the most balanced model overall.

2. Latency vs. CPU Trade-offs

Latency and CPU usage data made clear that model size had a direct impact on usability in CPU-bound environments. Larger models like Phi-4 mini (3.8B) and Granite3.3:2b were capable but too slow, often taking over two seconds per query. By contrast, Qwen2.5 consistently achieved sub-second latency (~0.6s), which made it uniquely suitable for scaling to production workloads.

3. Confidence Calibration Needs

Scatter plots of confidence vs. discrepancies highlighted a recurring issue: nearly all models were systematically overconfident, with reported confidence scores clustering near 0.9–1.0 regardless of accuracy. This confirmed that raw confidence values could not be trusted as-is, and justified the development of the shrinkage and calibration layers applied later in the pipeline.

4. Error Patterns and Stability

Error counts revealed that schema compliance was tightly linked to inference settings. Conservative regimes reduced JSON parse failures but sometimes at the cost of reduced flexibility in borderline queries. Liberal regimes enabled more nuanced answers but introduced higher error rates. These findings underscored the importance of parameter sweeps and informed the hybrid inference strategy adopted later.

5. Overall Model Selection

Taken together, the figures made the trade-offs clear: Granite3.3:2b offered strong stability but prohibitive latency; Phi-4 mini delivered semantic power but was too slow for scale; Falcon models underperformed across domains; and Qwen2.5:0.5b-instruct provided the best overall balance of accuracy, latency, and schema reliability. For these reasons, Qwen2.5 was selected as the production labeling model.

Final Note

The combination of discrepancy analysis, latency profiling, confidence calibration, and error tracking provided a multi-dimensional view of performance. By interpreting results holistically, the project was able to move beyond superficial metrics and identify a labeling solution that was both practical and robust for powering downstream BERT fine-tuning.

Conclusion

The project successfully identified an optimal model and a robust evaluation framework for the search classification task. Qwen2.5:0.5b-instruct emerged as the top choice due to its excellent balance of speed and accuracy. The iterative process of prompt engineering, parameter tuning, and detailed performance analysis proved crucial in achieving the desired outcome. The developed evaluation harness provides a solid foundation for future model comparisons and optimizations.

***Note**: With some tweaking (namely to how the confidence was determined) to the system prompt as well as the prompt presented to the model Qwen2.5:0.5b-instruct discrepancies where reduced to 15/60 as opposed to the 25/60 in the graphs above.*

The most critical challenge discovered during this phase was **systematic model overconfidence**. Raw confidence scores from these lightweight models were not reliable indicators of true accuracy, frequently averaging over 90% even for incorrect classifications. This finding made it clear that a simple threshold-based decision system would be unreliable and that a dedicated, post-hoc calibration effort was essential before the classifier could be trusted.

4.0 Phase 2: Addressing Systematic Overconfidence via a Multi-Stage Calibration Pipeline

A major obstacle in deploying lightweight LLMs as automated labelers was their tendency toward systematic overconfidence. Even when a model produced the wrong classification, it often reported confidence scores above 90%, making raw outputs misleading and untrustworthy for downstream use. Since the ultimate goal was to generate a high-quality, calibrated dataset for BERT fine-tuning, solving this problem became essential.

Initial Mitigation

Prompt engineering was the first line of defense. Few-shot examples were adjusted to encourage more cautious scoring, especially for nuanced queries. While this reduced some extremes, it did not sufficiently address the broader inflation problem—scores still clustered unrealistically high, regardless of actual accuracy.

The Calibration Pipeline

To resolve this, a multi-stage calibration pipeline was introduced. Its purpose was not only to rescale outputs but also to create a dataset where confidence values reflected relative difficulty and uncertainty, giving BERT richer supervision during fine-tuning.

1. **Shrinkage Layer (Pre-Calibration)**:
   Raw confidence scores were first passed through a power-shrink function ($p' = p^\gamma$). This aggressively pulled inflated scores down, creating a more manageable range for calibration. Empirically tuned shrinkage factors were applied per domain:
   - Programming: $\gamma = 2.9$ → average scores reduced to ~0.64.
   - General queries: $\gamma = 1.8$ → average scores reduced to ~0.56.
     This pre-processing step was crucial for preventing instability in later calibration stages.

2. **Calibration Layer (Temperature Scaling)**:
   After shrinkage, scores were processed through down-only temperature scaling ($T \geq 1$). Unlike methods that can inflate scores (e.g., Platt scaling), this approach strictly decreased confidences toward a neutral baseline of 0.5, never raising them. This ensured that calibration always reduced risk rather than introducing new uncertainty.

3. **Guardrails and Automation**:
   To guarantee robustness, additional safeguards were introduced:
   - A “never increase” rule capped all calibrated scores at their raw value.
   - Scores of exactly 1.0 (trivial cases like “What is 2+2?”) were preserved.
   - A manager class automated fitting per domain and stored learned parameters in a calibrators.json, enabling repeatable, hands-free calibration during labeling.

Impact on the Dataset

The effect of the calibration pipeline was significant. Average raw scores that clustered near 0.85–0.99 were systematically realigned to more realistic values:

Domain	Raw Avg. Confidence	Calibrated Avg. Confidence
Programming	~0.85	~0.732
General	~0.8	~0.712

This transformation gave the final labeled dataset two critical advantages:

- **Trustworthiness**: Confidence scores now reflect *relative model uncertainty* instead of meaningless inflation. The absolute value of a score (e.g., 0.72) is less important than its relationship to other scores. If one query is assigned 0.80 and another 0.60, we can reliably treat the first as “more certain” than the second, even if neither score maps perfectly to true probability. This relative reliability allows the system to rank borderline cases, prioritize which queries to re-check, and provide BERT with a more nuanced training signal.

- **Utility for BERT**: By training on calibrated data, BERT was exposed not just to binary labels, but also to **graded signals of certainty**. This strengthened its ability to handle ambiguous or edge-case queries where intent is less obvious. Instead of treating every decision as equally confident, BERT could learn from the subtle gradations in uncertainty baked into the labeled dataset.

Conclusion of Phase 2
With this calibration pipeline in place, lightweight LLMs were transformed from overconfident and unreliable predictors into **dependable labelers**. Their outputs became structured, trustworthy, and rich enough to support the large-scale data labeling effort. This set the stage for Phase 3: scaling to hundreds of thousands of queries, auditing for bias and imbalance, and producing a robust dataset for training the final, production-grade BERT classifier.

5.0 Phase 3: Scaling to a Production-Grade Classifier and Dataset Auditing

While the calibrated LLM classifier validated our auto-labeling approach, its latency and general-purpose architecture were unsuitable for a high-throughput production environment. To meet these demands, Phase 3 focused on building a distributed, containerized labeling cluster and implementing systematic dataset auditing to ensure the quality of training data for the BERT fine-tuning stage.

5.1 Distributed Labeling Infrastructure (FastAPI)

To break single-machine bottlenecks, we built a **containerized, peer-to-peer labeling cluster** that can run across heterogeneous hardware (Windows, Linux, Raspberry Pi). Docker standardizes the runtime; FastAPI provides the control plane. The system is designed for **fault tolerance, idempotence, and linear scaling**.

Work sharding & claims

- **Input sharding:** Large CSV sources are split into fixed-size shards (byte-range or row-range), each identified by `shard_id`, `start`, `end`, and a content checksum.
- **Atomic claims:** Workers request work from the leader via `POST /claim`. The leader assigns the **next unclaimed shard** and returns a **lease** (e.g., 5–10 min TTL). Shard state transitions: `unclaimed → leased → committed → replicated`.
- **Idempotence:** A shard can only be committed once. Duplicate commits are ignored by comparing `shard_id` + checksum. If a worker dies, the lease expires and the shard returns to `unclaimed` (no overlap, no double-processing).

Leader election (no single point of failure)

- **Membership & heartbeats:** Every node posts `POST /status` to all peers (or the leader) at a fixed interval (e.g., 5s), including `node_id`, health, `current_index` (furthest assigned global offset), and lag.
- **Election rule:** Among **healthy** nodes (no missed heartbeats beyond N intervals), the **node with the lowest `current_index`** becomes leader. Ties break on `node_id`.
- **Ephemeral leadership:** Leadership is a **lease** renewed on each heartbeat. If the leader misses K heartbeats, the cluster triggers a re-election; the next eligible node takes over and **rebuilds the assignment ledger** from replicated shard manifests.
- **Cold/warm start:** On startup, each node loads local `state.json` (last committed shard list, last seen ledger hash). The newly elected leader reconciles these manifests to guarantee a single consistent view.

Replication & eventual consistency

- **Write-path:** When a worker finishes a shard, it streams outputs to local disk as `*.jsonl` and calls `POST /commit` with `{ shard_id, checksum, rows, byte_count }`.
- **N-way replication:** The leader assigns **two replica targets** (`replica_a`, `replica_b`). Workers (or the leader) push the finished `*.jsonl` to both peers via `PUT /replicate`. Each replica verifies checksum and responds with an ack.

- **Durability rule:** A shard transitions to **`replicated`** only after **2 acks**. Until then, the shard remains in `committed` and is eligible for re-push on retry.

- **Dedup & reconciliation:** Nodes maintain a per-epoch **replica manifest** (`replicated/{shard_id}.meta`). On rejoin or restart, a background reconciler compares manifests and backfills any missing replicas (eventual consistency).
- **Crash safety:** Because completed shards are **multi-homed**, a single node failure cannot lose work. On leader failover, the new leader rebuilds the ledger from peer manifests and resumes assignment at the correct global offset.

Status, health, and progress telemetry

- **Per-node telemetry:** Each node periodically sends `POST /progress` with:

    - CPU %, RAM, disk free
    - Current shard id/lease expiry
    - Shards/hour, rows/sec, moving average latency
    - Error counters (JSON parse failures, schema violations, timeouts)
    - Model/runtime hash (guard against mixed versions)

- **Aggregated view:** The leader exposes `GET /status` summarizing cluster health:

    - Active nodes / unhealthy nodes
    - Queue depth (unclaimed + leased)
    - Throughput (rows/sec) and estimated time remaining
    - Replication backlog (committed not yet replicated)

Lightweight head-node option

- **Coordinator-only mode:** Raspberry Pi devices can run in **leader-only** mode (no heavy inference). They handle claims, heartbeats, and replication orchestration, leaving GPU/CPU-rich nodes to perform labeling.

Failure scenarios (and how they're handled)

- **Worker crash:** Lease expires → shard returns to `unclaimed` → reassigned. No duplicate commits because `shard_id + checksum` must be unique at `POST /commit`.
- **Leader crash/partition:** Heartbeat lease expires → re-election triggers → new leader rebuilds the ledger from replica manifests → processing resumes from the **lowest missing shard**.

- **Network flap:** Replication retries with exponential backoff; reconcilers heal gaps once connectivity returns.

Outcome at scale

- **Linear scaling:** Adding workers increases throughput nearly linearly until the replication layer becomes the bottleneck; at that point, we parallelize replication and/or increase shard size.
- **Durability:** Finished work survives single-node failures by construction (2-way replication).
- **Observability:** The status endpoints and error counters make bottlenecks visible (CPU vs. memory bandwidth, schema errors, replication backlog), which is critical for tuning on CPU-bound hardware.

Why this matters: This isn't just a "script that labels data." It's a **mini data factory**: deterministic work partitioning, resilient coordination, durable replication, and live telemetry—designed so weak and strong machines can collaborate safely without babysitting.

This system scaled linearly with nodes added, achieving 200k+ labeled examples, while automated monitoring captured CPU usage, memory consumption, and system health. Failover tests confirmed robustness, with labeling resuming seamlessly after node failures.

5.2 Dataset Auditing at Scale (Docker)

While FastAPI endpoints enabled coordination between labeling nodes, **Docker was the backbone** of the dataset auditing workflow. Containerization standardized execution across heterogeneous hardware, providing a consistent runtime environment regardless of whether the host was Windows, Linux Mint, or Raspberry Pi.

Linux-first deployment
Most containers were deployed on lightweight Linux distributions, with Linux Mint XFCE serving as the primary base and raspberry pi OS 64bit on the single raspberry pi head Node. Linux provided a leaner runtime footprint, faster startup times, and significantly faster Ollama performance compared to Windows hosts (upwards of 400 percent on CPU hardware). Docker's reliance on Linux namespaces and cgroups also meant resource usage could be more precisely constrained, which mattered when scaling across low-power devices.

Image management and portability

- A **single base image** was built locally, not including the Ollama REST API client, but including preprocessing scripts, and auditing tools.

- That image was exported into a `.tar` archive and distributed across machines, ensuring that every node—whether Windows or Linux—was running **exactly the same environment**.
- This eliminated “dependency hell,” where slight package mismatches could otherwise cause drift between nodes.

Dataset mounting and I/O separation
To minimize container runtime and avoid unnecessary rebuilds:

- The **raw dataset was mounted as a Docker volume** from the host file system, rather than being copied into the image.
- Outputs were written to dedicated host directories outside of the container’s writable layer. This ensured that processed shards weren’t bloating the container image itself and allowed results to persist even if containers were recycled or rebuilt.
- This separation reduced both container startup time and replication overhead when auditing at scale.

Limitations and API dependencies
Each container required the Ollama REST API to be installed **individually on the host machine**. While this introduced extra setup steps and limited pure portability, it was a trade-off chosen to keep the Docker image lightweight and avoid GPU driver incompatibilities across environments. The downside was tighter coupling to local installs, but the upside was clean isolation of core logic and easy distribution of the rest of the pipeline.

Compatibility across ecosystems
Docker provided the **compatibility layer** that made this entire system feasible:

- Windows nodes could run the same images as Linux nodes, despite their underlying OS differences. This was mainly used for testing
- Raspberry Pis could participate as lightweight coordination-only leaders without diverging from the containerized architecture.
- By locking every node to the same image hash, auditing results remained consistent regardless of the machine used.

Outcome
The Docker-driven approach allowed the auditing pipeline to scale without fragmentation. Containers could be launched, replaced, or migrated between machines with zero change in configuration. This not only ensured consistency across hundreds of thousands of labeled examples but also made failure recovery simpler—restart the container, remount the dataset, and rejoin the cluster.

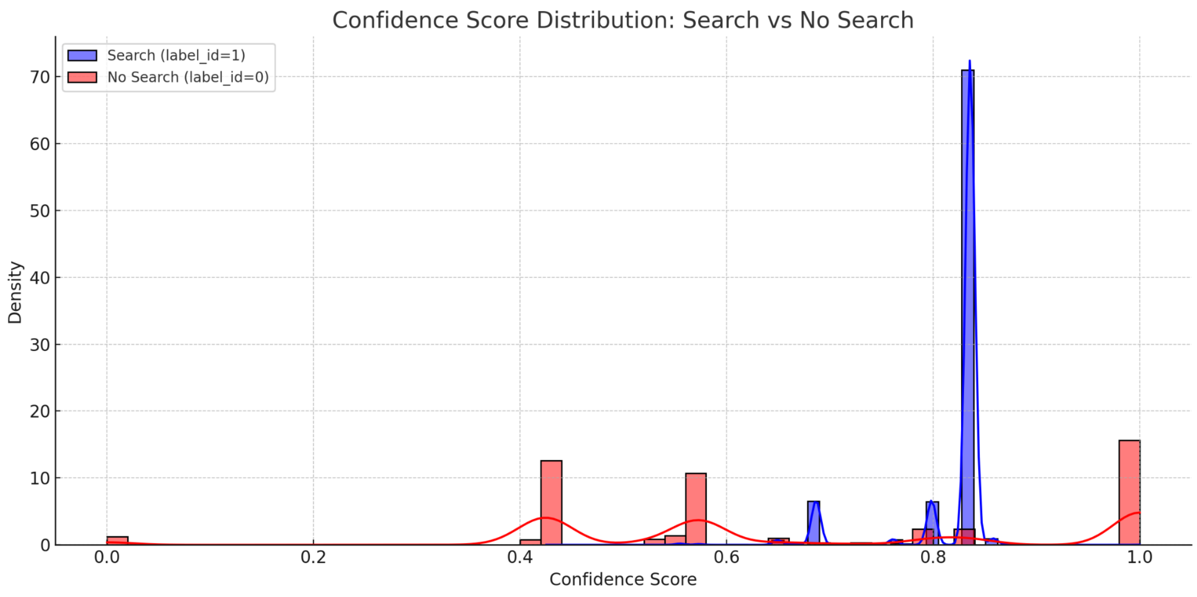
5.3 Results

The distributed labeling and auditing process produced both the scale and the insight necessary to advance toward a production-grade classifier. More than 200k examples were labeled successfully, and systematic auditing revealed critical patterns that guided both preprocessing and model design.

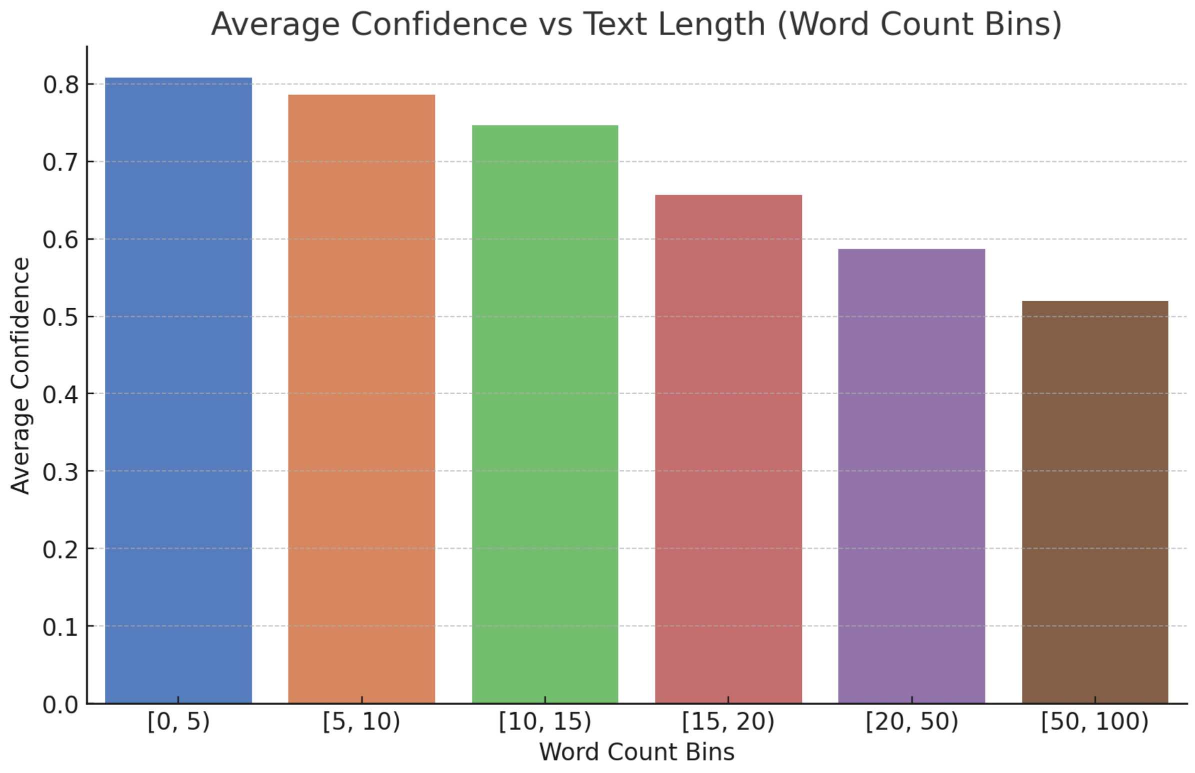
Throughput and Resilience
The cluster scaled linearly with added nodes, surpassing 200k labeled examples (down from ~300k raw due to the 650-token cap). Failover tests demonstrated that leader election and shard replication mechanisms worked as intended: when nodes failed mid-run, others seamlessly resumed processing without manual intervention. These results validated the cluster as a fault-tolerant backbone for high-volume labeling.

Dataset Quality and Insights
Auditing workflows uncovered key structural patterns in the dataset:

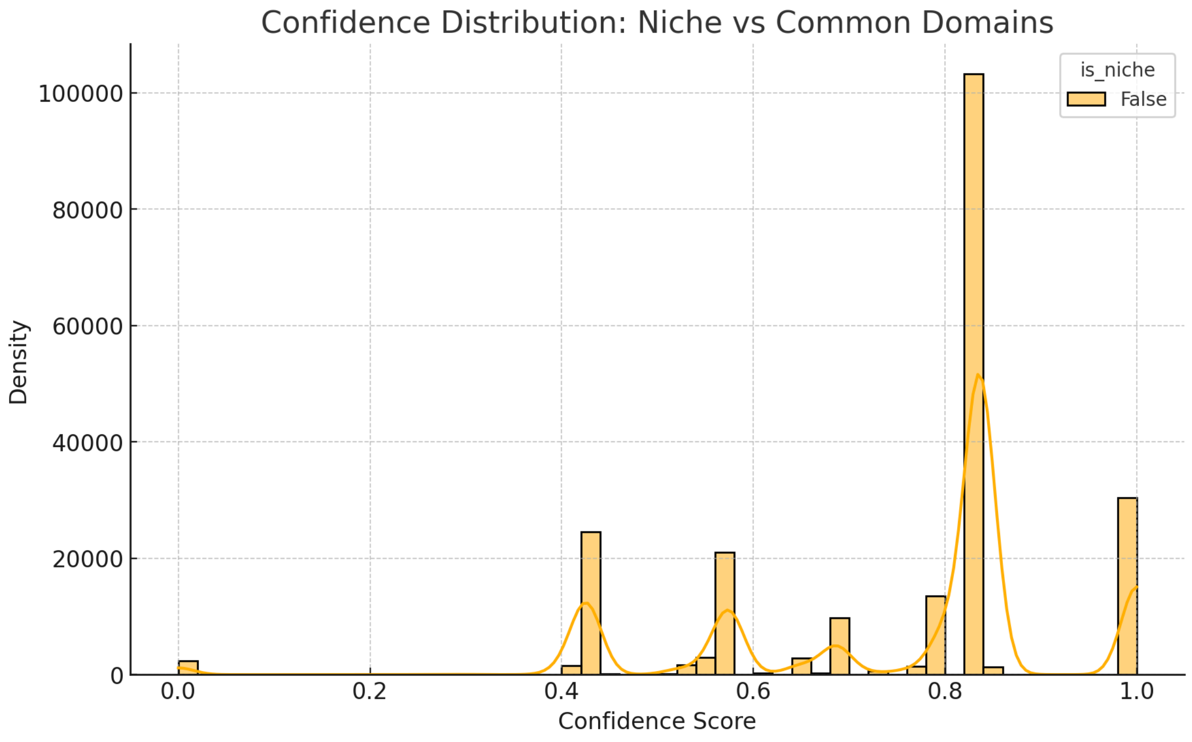
- **Confidence Distribution (Figure 1):** A density plot of search vs. no-search confidence showed raw outputs clustering around 0.8–1.0, a clear signal of systematic overconfidence. This reaffirmed the need for calibration layers to make the scores meaningful.



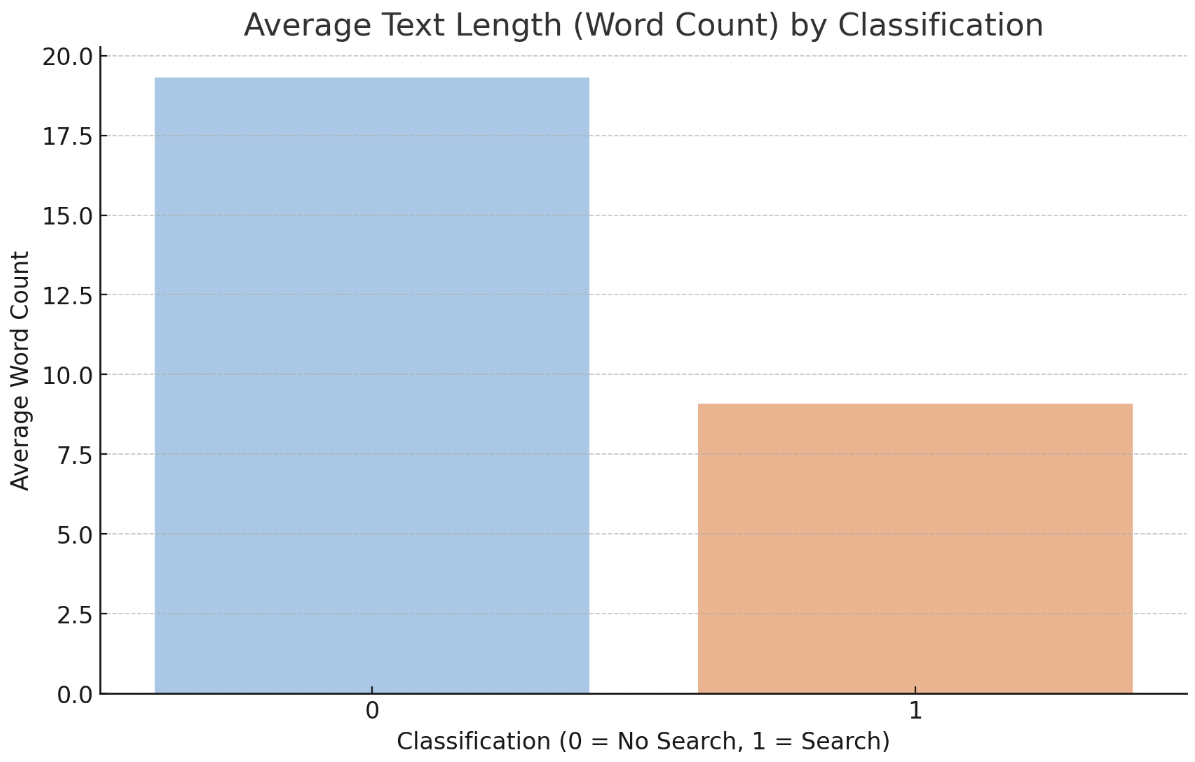
- **Length vs. Confidence (Figure 2):** Scatterplots revealed that short queries (<10 words) averaged ~0.8 confidence, while long queries (>50 words) dropped below ~0.55. This confirmed the plan to introduce fallback handling for long queries in future iterations.


Average Confidence vs Text Length (Word Count Bins)
Average Confidence
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
[0, 5)
[5, 10)
[10, 15)
[15, 20)
[20, 50)
[50, 100)
Word Count Bins

- **Domain-Aware Stratification (Figure 3):** Niche domains like medical and math displayed broader variance in both confidence and label consistency. As a result, medical queries were hardcoded to “search” (due to factual complexity), while math queries were hardcoded to “no search” (since they are purely logic-based).



- **Text Length by Label (Figure 5):** On average, “no search” queries were ~19 words long compared to ~9 words for “search” queries, quantifying the intuition that fact-seeking queries are short and definitional ones more verbose.



Fine-Tuning Baseline
With this audited dataset, preliminary runs of **bert-uncased** achieved ~90% accuracy after a single epoch. This early success validated not just the dataset’s quality, but also the entire distributed labeling and auditing framework as a pipeline capable of producing production-ready training data.

Synthesis
Together, these results demonstrate that the system does more than scale — it generates *actionable insights* into query structure and classifier behavior. By combining throughput, resilience, and deep auditing, the dataset was transformed from raw noise into a robust, high-quality foundation for training BERT.

6.0 Phase 4: Fine-Tuning the BERT Classifier (Preliminary and Future Recommendations)

Having created and audited a large-scale dataset, we entered the final phase of the search classification part of the project: using this data to solve the production challenge by fine-tuning a BERT model. This model was designed to serve as the final, optimized classifier, offering superior speed and consistency compared to the LLM-based prototype.

There wasn't enough time to go in depth fine tuning the BERT model nor is it the main focus of this report but preliminary results were very promising as detailed below.

6.1 Model Setup and Hyperparameter Rationale

The initial fine-tuning was performed using a standard `BERT-base` model, which contains approximately 110 million parameters. Following established best practices, we added a single linear classification layer on top of the `[CLS]` token representation, adapting the pre-trained transformer for our specific binary task. The hyperparameters were chosen conservatively to ensure stable and effective learning. BERT was trained to output binary classifications for this run (search, no search), future development aims to change that to a confidence aware training method.

Core Fine-Tuning Hyperparameters (For preliminary run)

Hyperparameter	Value	Rationale
`Optimizer`	`AdamW`	Decouples weight decay from the learning rate to prevent overfitting.
`Learning Rate`	`2e-5`	A low value is critical to avoid "catastrophic forgetting" of pre-trained knowledge.
`LR Schedule`	`Warmup (10%) with Linear Decay`	Stabilizes training in early iterations when gradients can be volatile.

`Batch Size`	`32`	Balances stable batch statistics with practical GPU memory constraints.
`Epochs`	`1`	For compute limitation purposes and to fast track data for this report.

6.2 Preliminary Results and Future Scaling Strategy

Initial Results
The first fine-tuning run of the BERT classifier achieved ~90% training accuracy on the 200k-sample dataset after only a single epoch. This rapid convergence validated two things:

1. The **data pipeline** worked — large-scale labeling and calibration produced high-quality training inputs.
2. The **fine-tuning methodology** was sound — even a baseline run yielded strong performance, laying the foundation for scaling experiments.

Why Confidence-Aware Classification Matters
A key innovation of this project was designing the dataset to contain not only binary labels (`search` vs. `no search`) but also **calibrated confidence values**. Teaching BERT to learn from both provides several strategic benefits:

- **Trustworthy Decisions:** Labels alone tell the model "yes/no." Confidence values give a **degree of certainty**, allowing the system to differentiate between "obvious" and "borderline" cases. This is crucial for downstream automation (e.g., flagging low-confidence classifications for fallback).
- **Better Generalization:** By training on *soft labels* (probability distributions, e.g., `0.82/0.18` instead of `1/0`), BERT absorbs the *uncertainty patterns* of its teacher model. This distillation process helps the student model avoid overfitting and perform more robustly on ambiguous, real-world queries.
- **Confidence as a Usable Signal:** Adding a **confidence head** allows BERT to output both a classification and a scalar reliability score. Even if the absolute probabilities aren't perfect, their **relative values** (Query A more certain than Query B) give the system a practical way to rank results, trigger fallbacks, or guide user interfaces.
- **Foundation for Auditing & Scaling:** Confidence-aware training opens the door to advanced evaluation metrics (e.g., Expected Calibration Error, reliability diagrams, Brier score). These metrics go beyond accuracy, enabling us to measure how much

the system “knows what it doesn’t know.”

Strategic Roadmap
With the foundations validated, the scaling plan focuses on four pillars:

- **Compute:** Explore distributed, multi-GPU training to handle larger models and hyperparameter sweeps.
- **Advanced Models:** Benchmark stronger architectures (BERT-Large, RoBERTa, DeBERTa) for both accuracy and calibration performance.
- **Systematic Tuning:** Move beyond defaults with structured searches over learning rate, batch size, and regularization.
- **Confidence-Aware Training:** Fully formalize the use of soft labels, confidence heads, and temperature scaling. Curriculum training and ensembles will further improve robustness, while embedding-based OOD detection can downgrade confidence for queries outside the training distribution.

Why This Matters
Accuracy alone makes a model *useful*. Accuracy plus **trustworthy confidence** makes it *deployable*. By engineering the system to produce calibrated, confidence-aware outputs, we’ve created a classifier that is not only correct most of the time, but also *aware of when it might be wrong*. This property is what turns a research prototype into a reliable production system.

7.0 Conclusion: Engineering Insights and Final Reflection

Taming Model Overconfidence

A novel, multi-stage calibration pipeline was devised to tame systematically inflated confidence scores. By shrinking and rescaling outputs, the system produced relative confidence values that reflected true uncertainty rather than blind overconfidence. This transformed lightweight LLMs from fragile guessers into dependable auto-labelers, and created reusable training signals for downstream models — a capability often missing in small-model pipelines.

Scaling Beyond Single-Machine Limits

Containerization with Docker and orchestration via FastAPI enabled a distributed, fault-tolerant labeling cluster. With work sharding, leader election, and replication, the system scaled linearly across heterogeneous hardware while surviving node failures without manual intervention. This proved that, even under constrained resources, production-grade throughput is achievable with the right architecture. It was less a script, more a factory — resilient by design, and built for real-world use.

Data-Driven Classifier Optimization

The final BERT classifier validated the entire pipeline. Achieving ~90% accuracy in its first epoch, it confirmed the dataset's quality, the calibration pipeline's utility, and the auditing framework's rigor. Beyond raw accuracy, the confidence-aware design ensured the model could provide not only an answer but also a usable measure of trust in that answer. This foundation sets the stage for scaling into advanced models like RoBERTa or DeBERTa without changing the core architecture.

Final Reflection

What began as a reaction to the failure of brittle web-scraping grew into a blueprint for privacy-first, data-centric AI engineering. Every failed experiment — from dependency parsing breakdowns to toy models hallucinating with misplaced certainty — forced a redesign. Each redesign pulled the system closer to resilience.

The sleepless nights spent debugging JSON outputs, reworking Docker mounts, and watching shards flow across machines built something more than a classifier: they built a framework. A framework for engineering systems that don't just work on paper, but survive failure, scale under pressure, and respect the constraints of real hardware.

And still, this is not the end. This classifier is just one gatekeeper inside a larger offline chatbot architecture. Retrieval, summarization, and response generation still wait to be engineered. But the hardest lesson is already learned: accuracy alone doesn't make a system usable. Reliability, privacy, and resilience are what elevate a student project into something that can stand on its own.

This report captures only a slice of the journey. The project continues — each stage another iteration, another layer of engineering, another night trading sleep for code.

References (A non exhaustive list)

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. NAACL.
Vaswani, A. et al. (2017). *Attention is All You Need*. NeurIPS.

Guo, C. et al. (2017). *On Calibration of Modern Neural Networks*. ICML.
HuggingFace Transformers. Retrieved from: https://huggingface.co/docs/transformers
Docker Documentation. Retrieved from: https://docs.docker.com/
FastAPI Documentation. Retrieved from: https://fastapi.tiangolo.com/
Ollama (Llama.cpp runtime wrapper). Retrieved from: https://ollama.ai/

Project-Specific Readings

- *Data Generation for NLP Classification Dataset*.
- *Leveraging LLMs for Synthesizing Training Data Across Many Languages*.
- Amazon Science (2023). *Using Large Language Models (LLMs) to Synthesize Training Data*. Retrieved from: https://www.amazon.science/
- *Design Plan for Offline LLM Query Handling System*. (PDF, 2025).
- *Improving NLP-based Question Reconstruction for BERT Classification Pipelines*.
- *Optimizing an On-Device NLP Pipeline for Question Reconstruction & Classification*.