Università di Catania

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
MASTER'S DEGREE IN COMPUTER SCIENCE (MSC)

Dott. Andrew R. Darnall

LeetLLaMA System and Behcnmark Report

COURSE - INGEGNERIA DEI SISTEMI DISTRIBUITI E LABORATORIO

Academic Year 2024 - 2025

LeetLLaMA

Leveraging AI Collaboration with Programmers

Dott. Andrew R. Darnall

July 28, 2025

Contents

Chapter 1

Project Overview

This project begins with an experimental benchmarking of four lightweight Large Language Models (LLMs), each heavily quantized and optimized for GPU-accelerated inference in resource-constrained environments.

The benchmarking serves as the foundational phase of the project. The second, implementation-focused phase involves the end-to-end development and cloud deployment of the solution using Amazon Web Services (AWS). This includes leveraging standard industry practices in:

- Infrastructure as Code (IaC),
- DevOps and GitOps workflows,
- Container orchestration (e.g., Kubernetes, ECS).

The benchmark evaluates both the baseline performance of the selected LLMs and the impact of grounding them with external knowledge sources. Specifically, it examines:

1. The performance deltas introduced by different types of Retrieval-Augmented Generation (RAG),
2. The structural characteristics and granularity of the knowledge base used during grounding.

This chapter lays the groundwork for understanding both the methodology and motivation behind the selection of models, tools, and deployment strategies in the subsequent phases.

Chapter 2

The Evolution of Software Engineering in the Era of AI

2.1 The ChatGPT Moment: When the World Took Notice

The paradigm shift brought about by Large Language Models (LLMs) became widely recognized with the release of OpenAI's **ChatGPT** in late 2022, a user-facing application built on the *InstructGPT* model [1]. For the first time, the general public—not just researchers and engineers—witnessed a machine capable of producing coherent, context-aware responses, generating code, summarizing documents, and answering questions with conversational fluency.

While LLMs had existed prior to this moment, ChatGPT's ease of access and surprisingly humanlike interaction catalyzed a wave of curiosity, investment, and practical experimentation across virtually every domain, including software development. The combination of natural language instruction and actionable output revealed the true disruptive potential of these models—not only as tools for text generation, but as collaborators in thought and execution.

2.2 A Brief History of LLMs Before the Boom

Despite their sudden visibility, LLMs are the product of a long trajectory in natural language processing. Early attempts to model language statistically were replaced by neural architectures such as word2vec, followed by the breakthrough transformer model introduced in *Attention Is All You Need* (Vaswani et al., 2017) [2]. The transformer architecture gave rise to successive generations of ever-larger pre-trained models, in-

cluding BERT (2018), GPT-2 (2019), and GPT-3 (2020), each pushing the boundaries of what was possible with scale and compute.

These models were trained on massive text corpora, capturing complex linguistic patterns and generalizing across tasks. The foundational idea of instruction-tuning, as introduced by InstructGPT, allowed these models to better align with user intent, paving the way for ChatGPT's conversational fluency.

2.3 The Software Engineering Community Reacts

The software engineering world responded with a blend of amazement, skepticism, and creativity. Many developers quickly recognized that LLMs could automate routine programming tasks—code completion, documentation generation, refactoring suggestions, and even unit test creation.

GitHub Copilot, powered by OpenAI's Codex model, became one of the earliest mainstream integrations of LLMs into development workflows. Initial concerns about hallucinated output and insecure code were tempered by real productivity gains. Many engineers began to integrate these tools as thought partners—consulting them the way one might bounce ideas off a colleague.

Communities emerged around prompt engineering, fine-tuning, and deploying open-source models for code generation. As with any major technological leap, the discourse oscillated between cautious optimism and unfounded panic.

2.4 Tools, Culture, and the Rise of the "Vibe Coders"

With the LLM boom came an explosion of new developer-centric tools and platforms:

- **Code-focused LLMs**: Open-source alternatives like CodeLLaMA, StarCoder, and DeepSeekCoder gave rise to custom fine-tuning and on-device deployment.
- **Prompt-oriented IDE integrations**: VSCode extensions, Chat-based terminals, and browser-based copilots became staples.
- **Infra-as-code AI tooling**: AutoInfra, Terraform-based code generators, Helm chart assistants, etc.

Alongside this came a cultural wave—self-described *"vibe coders"*—newcomers drawn not by formal CS backgrounds, but by curiosity, creativity, and access to powerful generative tools. This democratization blurred traditional boundaries between

developer and non-developer, encouraging new forms of collaboration and prototyping.

2.5 Debunking the Replacement Myth: The Bigger Picture

Fears of developer obsolescence quickly circulated: *"Will AI replace programmers?"* While not baseless in emotional tone, these concerns largely miss the complexity of what software engineering entails.

Programming is but one piece—often a relatively small one—of the engineering lifecycle. Real-world software requires architectural design, integration with distributed systems, performance optimization, security auditing, deployment strategy, and team communication. These are domains in which human context, trade-off analysis, and domain knowledge remain irreplaceable.

Historically, programming has evolved through layers of abstraction: from assembly to high-level languages, from imperative to declarative paradigms, from manual deployment to CI/CD pipelines. LLMs are a continuation of this trend—not the end of it. They offload boilerplate and enhance expressivity, but they do not replace the engineer's judgment or responsibility.

2.6 LLMs as Learning Companions and Productivity Boosters

Used effectively, LLMs offer unprecedented productivity gains. They accelerate exploration, debug unfamiliar APIs, generate scaffolding code, and reduce the cognitive load associated with routine tasks. More profoundly, they serve as dynamic learning companions.

For aspiring programmers, the availability of conversational AI tutors dramatically lowers the barrier to entry. Unlike static documentation or forum posts, these models offer context-aware, iterative explanations—tailored to the user's current level and goals.

As someone who entered the field during what many call the "good old days" of Stack Overflow and cryptic documentation, I can personally attest to the sheer contrast. The tools now available to newcomers are nothing short of revolutionary. That said, the fundamentals remain unchanged: critical thinking, curiosity, and a commitment to clean, maintainable code.

In the same way calculators didn't kill mathematics, LLMs won't kill programming. Instead, they will redefine what it means to build software—shifting the emphasis from syntax memorization to creative problem-solving.

Chapter 3

Vector vs. Graph RAG: Knowledge Retrieval Strategies in Modern AI Systems

3.1 The Challenge of Fine-Tuning: Data and Compute Bottlenecks

While Large Language Models (LLMs) have demonstrated exceptional generalization across diverse tasks, tailoring them to specific domains remains a formidable challenge. The most direct approach—**fine-tuning**—involves retraining the model on domain-specific data. However, this comes with two significant costs:

1. **Data scarcity**: LLMs typically require millions to billions of tokens for meaningful domain adaptation. Curating such corpora in niche fields (e.g., legal, medical, or industrial contexts) is often infeasible, both due to availability and privacy constraints.

2. **Compute intensity**: Training or even fine-tuning large models demands high-end GPU clusters, specialized software stacks, and considerable energy consumption. Fine-tuning a 13B parameter model, for instance, might require hundreds of GPU-hours, and full retraining is often reserved for resource-rich institutions.

These limitations motivated the development of alternative methods to inject knowledge into LLMs at inference time—without retraining the model. This gave rise to the paradigm known as **Retrieval-Augmented Generation** (RAG).

3.2 From Expert Systems to Grounded Language Models

The idea of augmenting reasoning systems with a structured external knowledge base is not new. In fact, it dates back to **expert systems** of the 1970s and 1980s. These systems, such as MYCIN and DENDRAL, operated by consulting human-curated rule sets and knowledge graphs to make decisions.

Retrieval-Augmented Generation revives this idea in the context of neural models. Instead of encoding domain knowledge inside the weights of the model, we *ground* the model by supplying relevant knowledge at inference time. The LLM serves as a language-based reasoning engine, while the external data store acts as the domain expert.

In this hybrid architecture, grounding allows models to produce responses that are factually anchored, up-to-date, and contextually rich—while avoiding the cost and brittleness of full fine-tuning.

3.3 Vector-Based RAG: Embeddings and Dense Similarity

The most widely used form of RAG is **Vector RAG**. At a high level, it works as follows:

1. A large collection of documents (or document chunks) is transformed into dense vector representations (embeddings) using a pre-trained model such as OpenAI's Ada, BERT, or SBERT.

2. At inference time, a user query is embedded using the same model.

3. The system performs a similarity search (typically cosine or dot-product) to retrieve the top-k semantically similar documents.

4. These retrieved documents are fed into the LLM as context (usually via prompt concatenation), guiding it to produce informed answers.

This technique is popular due to its simplicity, compatibility with vector databases (e.g., FAISS, Pinecone, Weaviate), and ability to operate on unstructured text without requiring annotation or formal schema.

3.4 Limitations of Vector RAG

Despite its utility, Vector RAG has significant limitations:

- **Fuzzy retrieval only**: Vector search is based on similarity, not meaning or logic. It lacks the ability to follow causal chains or structured relationships.
- **Loss of fine granularity**: Embedding-based retrieval often retrieves entire chunks, not specific facts, which can result in noisy or partially relevant contexts.
- **Inconsistency**: Semantically similar but factually irrelevant documents may be retrieved due to limitations of the embedding model.
- **Context window limits**: Feeding large volumes of retrieved text into the LLM risks hitting the token limit and reducing focus.

These issues become especially apparent in domains requiring precision, traceability, or multi-hop reasoning. This is where Graph-based approaches enter.

3.5 Graph RAG: From Association to Structured Reasoning

Graph-based RAG augments LLMs with structured, semantically rich knowledge graphs (KGs). Unlike vector embeddings, KGs capture entities and their explicit relationships.

There are two major variants of Graph RAG:

3.5.1 Association Graph RAG

In this approach, documents or concepts are represented as nodes, and co-occurrence or thematic similarity forms the edges. This graph can be constructed automatically using statistical methods or embedding similarity.

- Retrieval is guided by traversing neighboring nodes or calculating centrality and connectivity.
- Can uncover latent semantic structures across document sets.

3.5.2 Ontology-Based Knowledge Graph RAG

This is a more formal approach:

- Knowledge is stored as triples $(subject, predicate, object)$ in a graph, such as in RDF or OWL formats.
- Queries use structured languages (e.g., SPARQL), and reasoning can involve inference over class hierarchies and logic rules.
- These graphs are ideal for domains like biomedical research, law, or enterprise data where relationships are known and curated.

In both cases, retrieved subgraphs (not just raw text) are transformed into textual summaries or context documents, then passed to the LLM. This enables more precise and relationally-aware generation.

3.6 Comparative Advantages: When to Use What

Table 3.1: Comparison of Vector and Graph RAG

Aspect	Vector RAG	Graph RAG
Ease of Setup	Simple pipelines using embeddings and vector DBs	Requires graph construction and curation
Scalability	Easily scales to millions of docs	Less scalable; graphs grow in complexity
Interpretability	Low (dense black-box vectors)	High (human-readable relationships)
Reasoning	Shallow/fuzzy matching	Supports logical and multi-hop reasoning
Domain Suitability	General-purpose search and chatbots	High-stakes domains (e.g., finance, healthcare)

In practice, hybrid models are emerging that combine the breadth of Vector RAG with the precision of Graph RAG—often using graphs to filter or re-rank vector search results.

3.7 Multi-Source, Conditional, and Causal RAG (CAG)

The RAG landscape continues to evolve. Several advanced paradigms have emerged to push the boundaries:

- **Multi-source RAG**: Combines heterogeneous sources—e.g., unstructured text, relational databases, APIs—into a unified retrieval framework.
- **Conditional RAG**: Dynamically selects the retrieval method (vector vs. graph vs. API) based on the user query or metadata.
- **Causal RAG (CAG)**: Focuses on retrieving documents or graph segments that help explain cause-effect relationships. This involves building causal graphs from text and surfacing explanations, not just facts.

These innovations signal a future where RAG pipelines are not monolithic, but *adaptive retrieval strategies*, capable of understanding the query's intent and tailoring their knowledge access accordingly.

Chapter 4

Benchmarking Methodology and Experimental Results

4.1 Evaluated Models and Setup

In this benchmark, we evaluate four modern, small-scale Large Language Models (LLMs) optimized for GPU-accelerated, resource-constrained environments:

- **LLaMA3.2 3B**
- **Qwen3 4B**
- **Phi-3 3.8B**
- **Gemma3 4B**

All models were quantized for inference efficiency and deployed in a standardized environment to ensure consistent evaluation. These models were chosen for their favorable performance-to-size ratio and compatibility with lightweight cloud or edge deployments.

4.2 Effect of Model Size on Performance

As expected, there is a positive correlation between model size and overall performance on the tasks evaluated. Larger models such as Qwen3 4B and Gemma3 4B generally outperformed smaller ones (like LLaMA3.2 3B), particularly in scenarios requiring more complex reasoning.

However, this gain comes with increased latency and memory consumption, making the deployment trade-offs relevant in production settings. The performance increase tends to plateau slightly between 3.8B and 4B parameters, suggesting diminishing returns at that scale for the benchmarked tasks.

4.3 Grounding Strategies: RAG Configurations

We compare performance under several Retrieval-Augmented Generation (RAG) grounding strategies:

1. **Baseline (no RAG)**: The LLM relies solely on its pretrained internal knowledge.
2. **Classic RAG on LeetCode dataset**: Dense vector retrieval from curated LeetCode problems and solutions.
3. **Classic RAG on Python documentation**: Embedding-based retrieval from official Python docs.
4. **Classic RAG on both**: Combined context from LeetCode and Python docs.
5. **Graph RAG on LeetCode dataset**: Retrieval based on an association graph linking related problems and solutions.
6. **Graph RAG on Python documentation**: Ontology-driven graph modeling API relationships and common usage patterns.
7. **Graph RAG on both sources**
8. **Hybrid: Classic RAG on LeetCode, Graph RAG on Python docs**
9. **Hybrid: Graph RAG on LeetCode, Classic RAG on Python docs**

4.4 Grounding Performance Patterns

The grounding using the LeetCode dataset consistently outperformed grounding via Python documentation. This likely stems from the fact that the LeetCode corpus aligns more closely with the nature of the benchmark tasks, which involve algorithmic problem solving.

Furthermore, the differences between Classic RAG and Graph RAG were subtle in terms of raw performance metrics. However, it is important to note that retrieval quality, document chunking strategies, and graph construction fidelity play significant roles and could tilt the results more distinctly with better tuning.

4.5 Evaluation Metrics

To assess the quality of the generated code and completions, we use the following metrics:

- **CodeBLEU**: An extension of BLEU tailored for code, accounting for syntax structure, data flow, and semantic equivalence [3].
- **ROUGE-L**: Evaluates overlap of longest common subsequences, useful for measuring textual similarity and structural fluency.
- **Self-BLEU**: Measures intra-set diversity—how similar model-generated outputs are to each other—to detect mode collapse or over-repetition.

These metrics offer a balanced view of output quality (CodeBLEU), readability and fluency (ROUGE-L), and diversity (Self-BLEU), providing a more holistic performance profile than traditional metrics alone.

4.6 Benchmark Results Overview

The following figures present a summary of the average scores across all evaluated models and grounding setups:

Figure 1: Baseline Performance (No RAG)

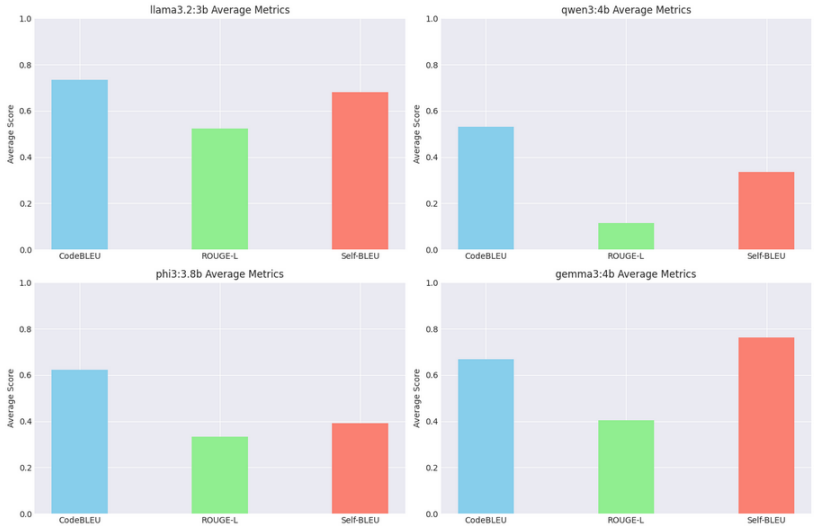

Figure 4.1: Average evaluation metrics across all models in the baseline (no-RAG) setting.

Figure 2: Hybrid Grounding Performance

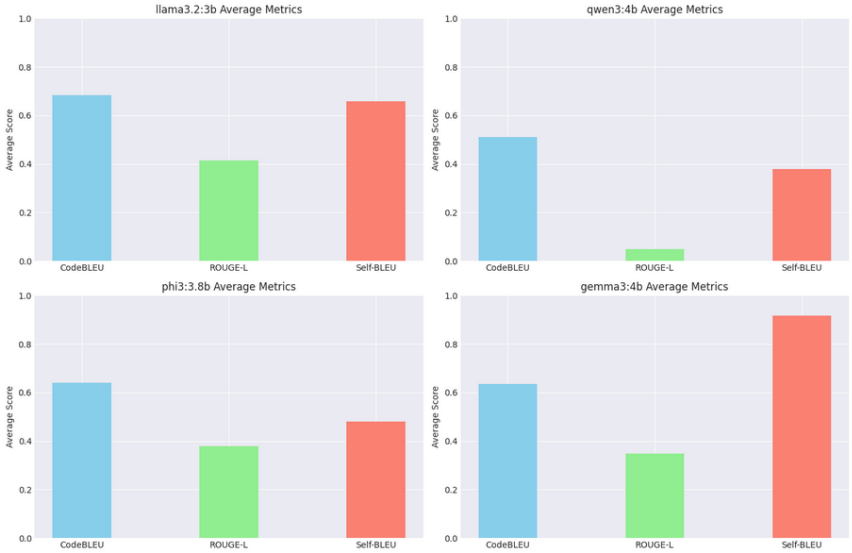

Figure 4.2: Average evaluation metrics for Graph RAG on LeetCode and Classic RAG on Python documentation.

These graphs clearly highlight the performance lift introduced by grounding, especially when LeetCode data is involved. The hybrid setup combining Graph RAG and Classic RAG yields a favorable balance between precision and contextual richness.

Full tabular results for each model and metric are included in Appendix A.

Chapter 5

Conclusions and Reflections

What Has Been Achieved

Throughout this project, we have journeyed across the intersection of software engineering and modern artificial intelligence, focusing on the growing synergy between Large Language Models (LLMs) and programming workflows.

The work began with a comprehensive survey of how the landscape of software engineering has transformed since the advent of ChatGPT and instruction-tuned models. We examined the cultural and technical ramifications—ranging from toolchain innovation to the rise of "vibe coders" and the reshaping of learning practices among developers. This contextualized the increasingly symbiotic relationship between human engineers and generative AI systems.

Subsequently, the theoretical underpinnings of Retrieval-Augmented Generation (RAG) were explored in depth. Both Vector RAG and Graph RAG paradigms were dissected, contrasted, and discussed in terms of their architectural structures, information access strategies, and domain applicability. By connecting modern RAG methods to earlier expert systems and semantic web concepts, the work bridged contemporary trends with foundational knowledge engineering.

The core of this work, however, lies in the benchmark. Four state-of-the-art yet compact LLMs—LLaMA3.2 (3B), Qwen3 (4B), Phi-3 (3.8B), and Gemma3 (4B)—were evaluated across a spectrum of grounding strategies. The results, analyzed via CodeBLEU, ROUGE-L, and Self-BLEU, provided nuanced insights into how performance scales with model size, the choice of retrieval strategy, and the nature of the knowledge base.

Looking Forward: What Comes Next

While the current work offers valuable observations, it is far from exhaustive. Several promising avenues remain for future exploration:

- **Scaling up grounding**: Leveraging larger, multi-modal knowledge bases or integrating real-time API sources to simulate dynamic expert systems.
- **Hybrid retrieval orchestration**: Experimenting with automatic switching between vector and graph RAG based on query type, using retrieval policy models.
- **Interactive agents**: Extending the setup from single-query answering to stateful, multi-turn agents with memory and planning, using the RAG system as a backend.
- **Latency and efficiency benchmarking**: Incorporating performance metrics like latency, throughput, and memory consumption to complement quality scores.
- **User studies**: Evaluating the real-world utility of the RAG-augmented LLMs with developers through usability studies or task-based evaluations.

These directions hold potential not only for improving the technical performance of LLM systems, but also for deepening their integration into the workflows of real-world software engineers.

Drawing Meaning from the Results

The experimental findings affirm a few key insights:

1. Even small-scale LLMs, when paired with appropriate grounding strategies, can approach or surpass much larger models operating without context.
2. Graph RAG offers competitive performance to Classic RAG, particularly when the retrieval graph is cleanly constructed. However, its benefits become more pronounced in domains where structured relationships and multi-hop reasoning are critical.
3. Grounding with a purpose-aligned corpus—such as the LeetCode dataset—proved more beneficial than using a general reference source like Python documentation. This reinforces the importance of dataset alignment, regardless of retrieval method.

4. Hybrid grounding configurations, combining Graph RAG and Classic RAG, yielded the best average results in certain cases—highlighting the value of compositional knowledge access.

In short, this project validates the notion that retrieval-aware systems can dramatically improve the capabilities of compact LLMs—making them not just lighter alternatives to frontier models, but powerful, tunable tools for real-world applications.

Thank you for your attention.

Dott. Andrew R. Darnall

Bibliography

[1] Long Ouyang, Jeff Wu, Xu Jiang, et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[3] Shuo Ren, Zhi Tu, Daya Chen, et al. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.