# Fine-Tuning Language Models & Managing Large Codebases in Limited Context Windows

Strategies for Optimizing LLM Performance with Technical Codebases

July 5, 2025

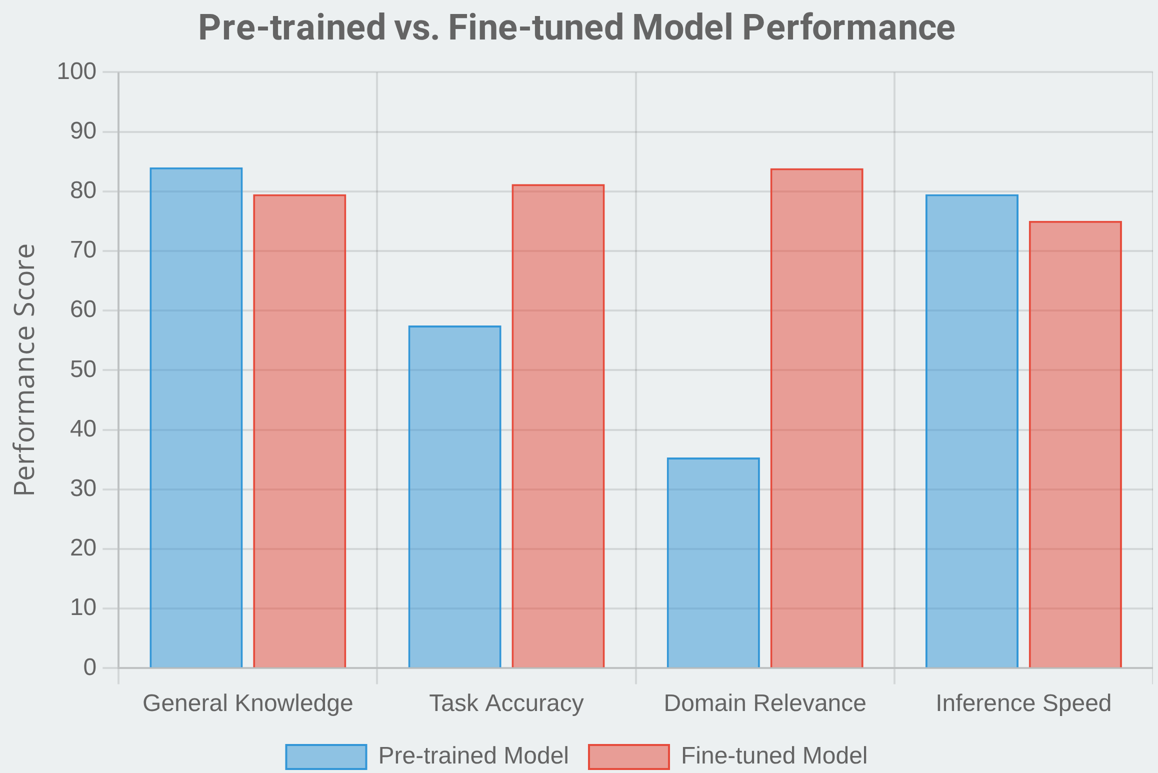
# Introduction to Fine-Tuning Language Models

## What is Fine-Tuning?

Fine-tuning is the process of **further training a pre-trained model** on a specific dataset to adapt it for particular tasks or domains.

## Why Fine-Tune?

- Improves performance on **domain-specific tasks**
- Requires less data and compute than training from scratch
- Enables customization while leveraging pre-trained knowledge
- Reduces hallucinations for specialized applications



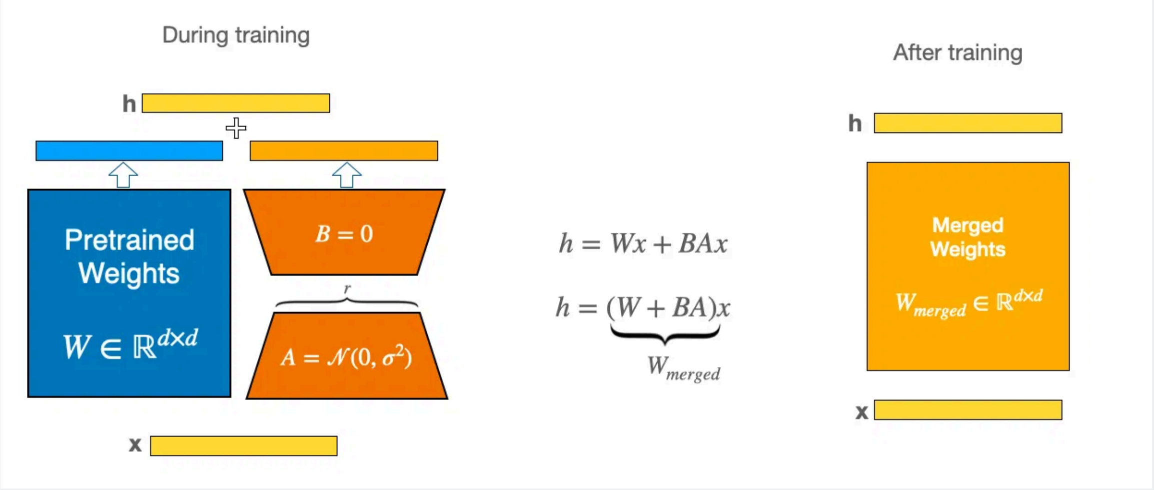

> *"Fine-tuning tailors the model to have better performance for specific tasks, making it more effective and versatile in real-world applications."*

# Full Fine-Tuning vs. Parameter-Efficient Fine-Tuning

## Comparison

| Aspect | Full Fine-Tuning | PEFT |
|---|---|---|
| **Parameters Updated** | All | Small subset |
| **Memory Requirements** | High | Low |
| **Training Speed** | Slow | Fast |
| **Performance** | Excellent | Very Good |
| **Catastrophic Forgetting** | High Risk | Lower Risk |

PEFT (Parameter-Efficient Fine-Tuning) methods update only a small fraction of model parameters, dramatically reducing computational and memory requirements while maintaining comparable performance.




## Key PEFT Approaches

- LoRA: Low-Rank Adaptation using matrix decomposition
- QLoRA: Quantized LoRA for even greater memory efficiency
- Adapter Tuning: Inserting trainable layers between frozen ones
- Prefix Tuning: Adding trainable prefix tokens to inputs

# PEFT Techniques: LoRA, QLoRA, Adapter Tuning

## LoRA (Low-Rank Adaptation)

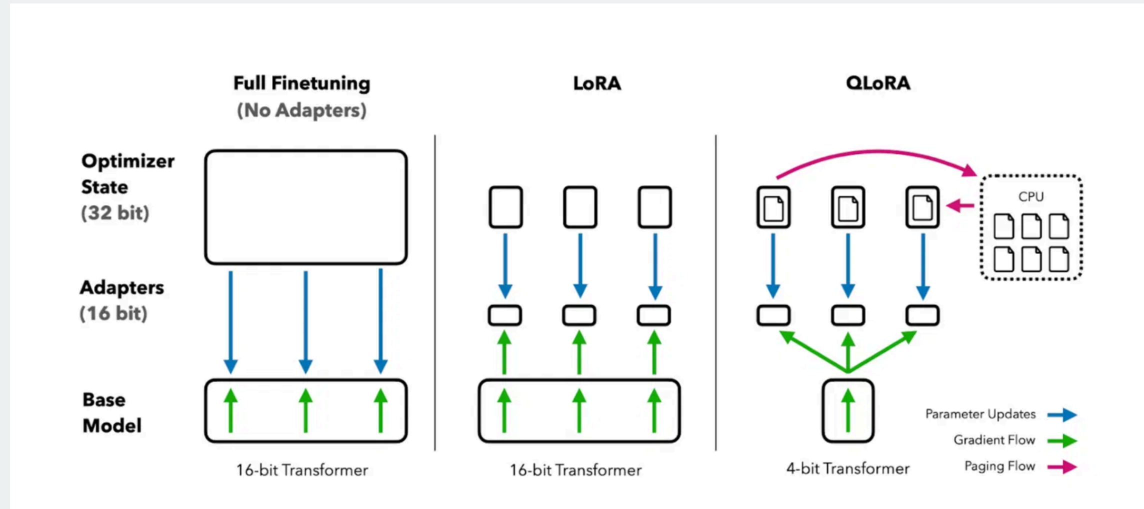
Decomposes weight updates into low-rank matrices, reducing trainable parameters by 10,000×.

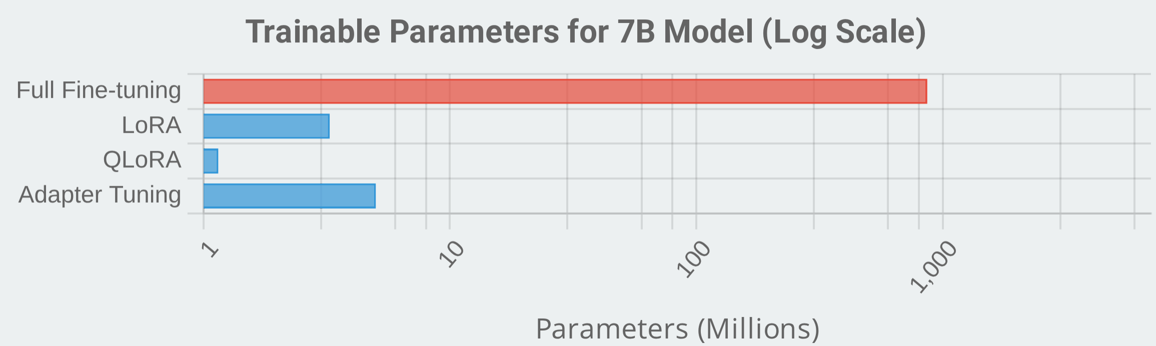
## QLoRA (Quantized LoRA)

Combines 4-bit quantization with LoRA for extreme memory efficiency, enabling fine-tuning of 65B+ models on consumer GPUs.

## Adapter Tuning

Inserts small trainable modules between frozen layers, enabling task-specific adaptation with minimal parameters.



# When to Fine-Tune vs. Prompt Engineering or RAG

## Decision Framework

**Choose Fine-Tuning When:**

- Need consistent, specialized behavior
- Have high-quality labeled data (100s-1000s examples)
- Task requires deep domain adaptation
- Inference speed is critical

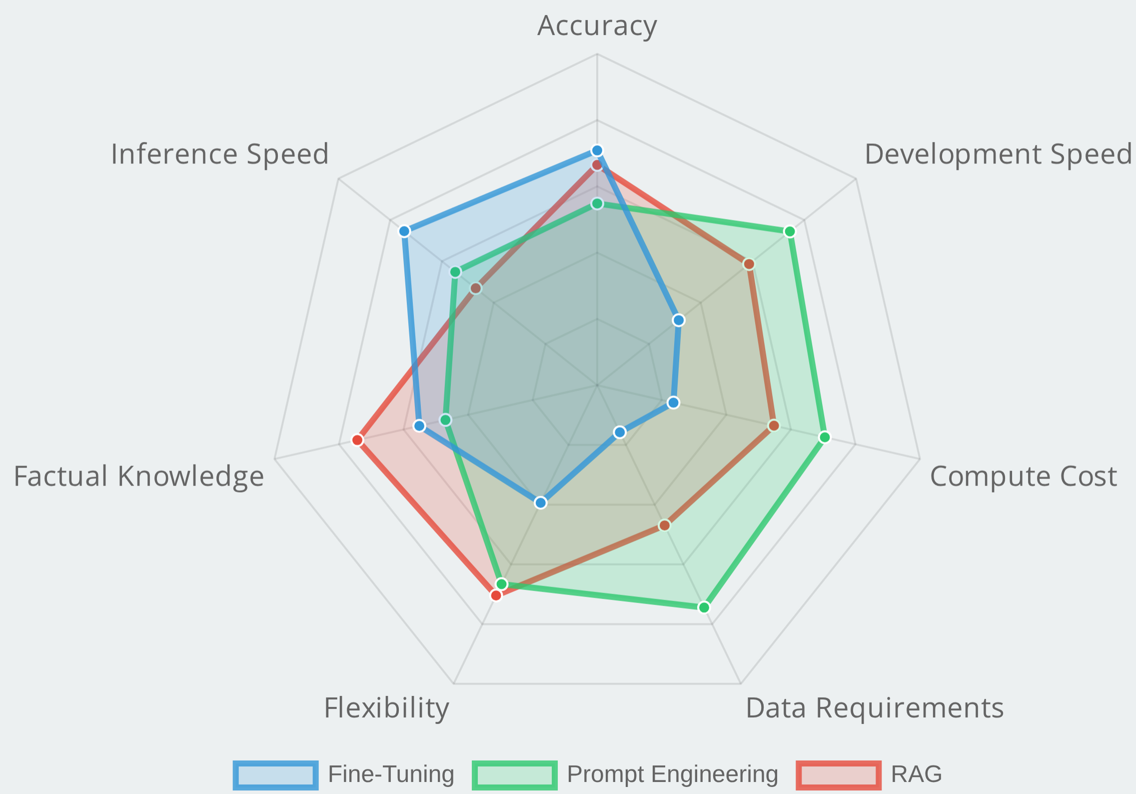
**Choose Prompt Engineering When:**

- Need quick iteration and experimentation
- Have limited examples (0-10)
- Task is within model's general capabilities
- Flexibility to change instructions is important

**Choose RAG When:**

- Need access to external/up-to-date knowledge
- Working with proprietary or changing information
- Factual accuracy is critical
- Want to reduce hallucinations

**Approach Comparison (Higher is Better)**




Hybrid approaches often yield the best results. Consider fine-tuning a model to better utilize RAG or follow instructions, then using prompt engineering to guide its behavior in specific contexts.

# Challenges in Fine-Tuning Large Models

## Compute Cost

Fine-tuning large models (>10B parameters) requires significant GPU resources, often making it prohibitively expensive for many organizations.

## Catastrophic Forgetting

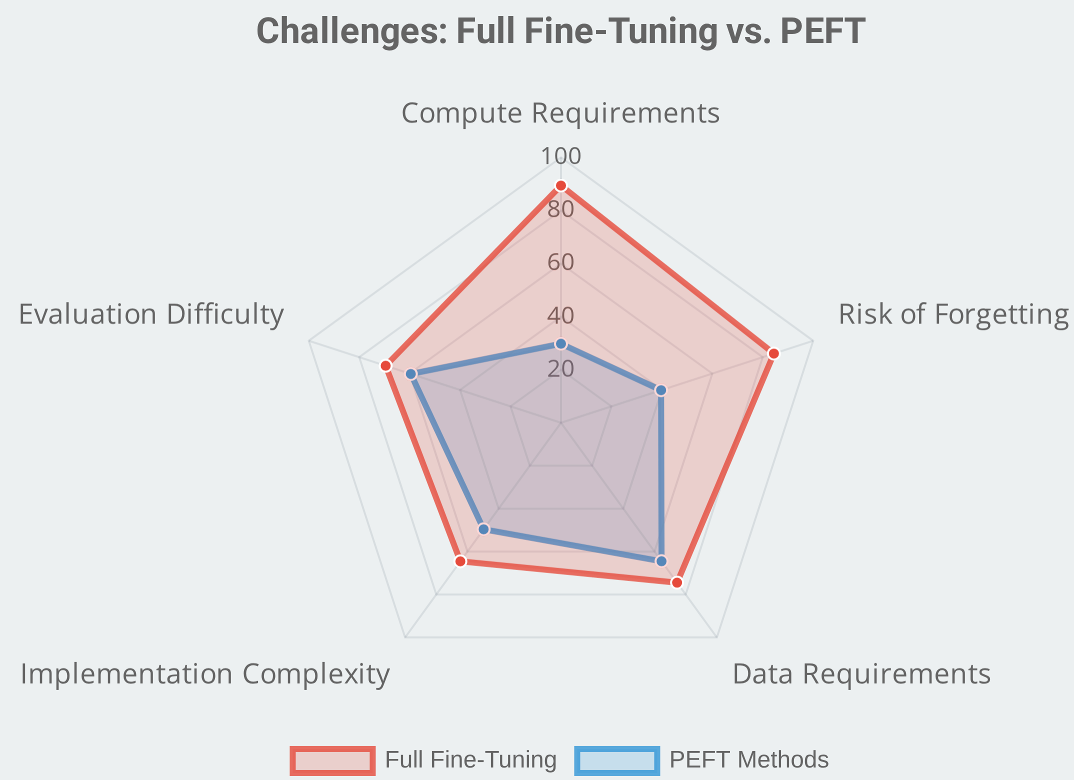
Models can lose previously learned knowledge when fine-tuned on new tasks, especially with small or domain-specific datasets.

## Data Quality Issues

Fine-tuning results are highly dependent on data quality. Biased, noisy, or insufficient data can lead to poor performance or reinforced biases.

## Evaluation Complexity

Determining if fine-tuning improved performance requires sophisticated evaluation metrics beyond simple accuracy measures.




**Mitigation Strategies:**

- Use PEFT techniques to reduce compute requirements
- Implement continual learning approaches to combat forgetting
- Invest in high-quality, diverse datasets with careful curation
- Develop comprehensive evaluation frameworks specific to your use case

# Context Window Limits in LLMs

## What is a Context Window?

The context window is the maximum amount of text an LLM can process at once, measured in tokens. It determines how much information the model can "remember" during a conversation or task.

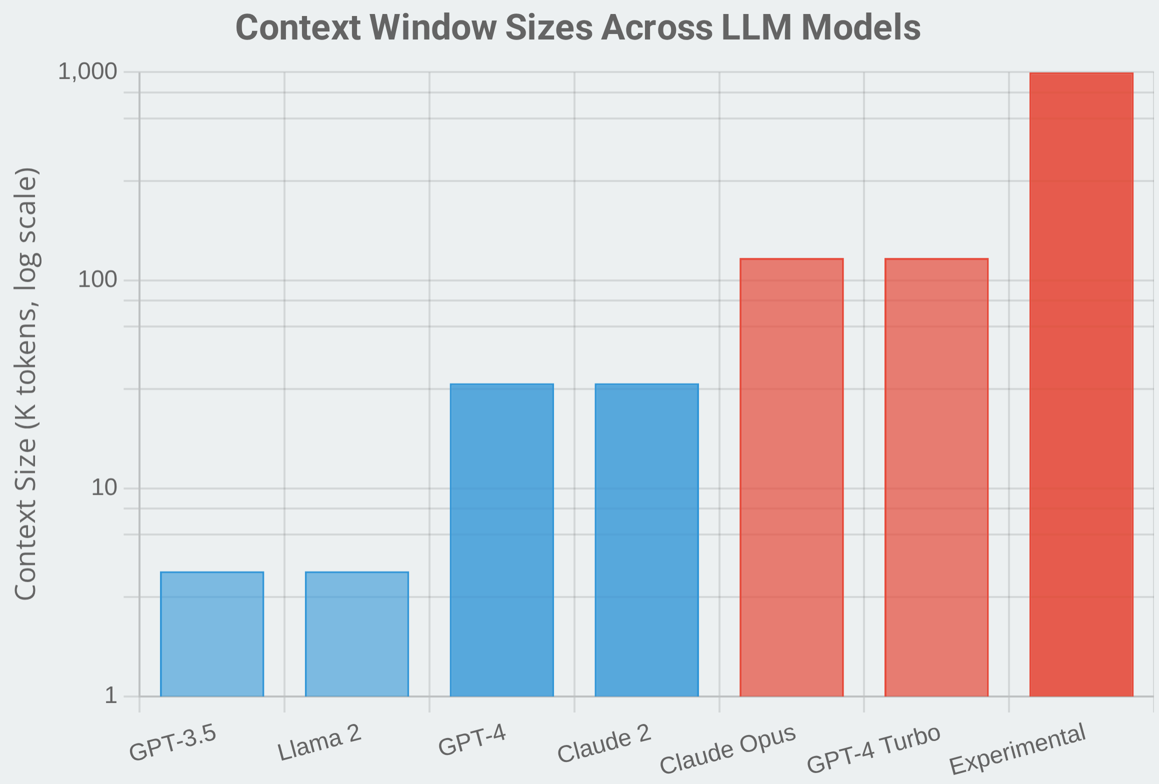
## Common Context Window Sizes

**4K tokens** (~3,000 words): GPT-3.5, Llama 2

**32K tokens** (~24,000 words): GPT-4, Claude 2

**128K tokens** (~96,000 words): Claude Opus, GPT-4 Turbo

**1M+ tokens**: Experimental models (e.g., Anthropic's 100K context)




## Implications for Code Processing

- Large codebases easily exceed context windows (e.g., Linux kernel: ~30M lines)
- Code requires **structural understanding** across files
- Context fragmentation can lead to inconsistent responses
- Longer contexts increase latency and token costs

# Chunking Strategies for Large Codebases

## Chunking Approaches

**Semantic Chunking**

Divides code based on logical units of functionality, preserving semantic relationships.

**Function-Level Chunking**

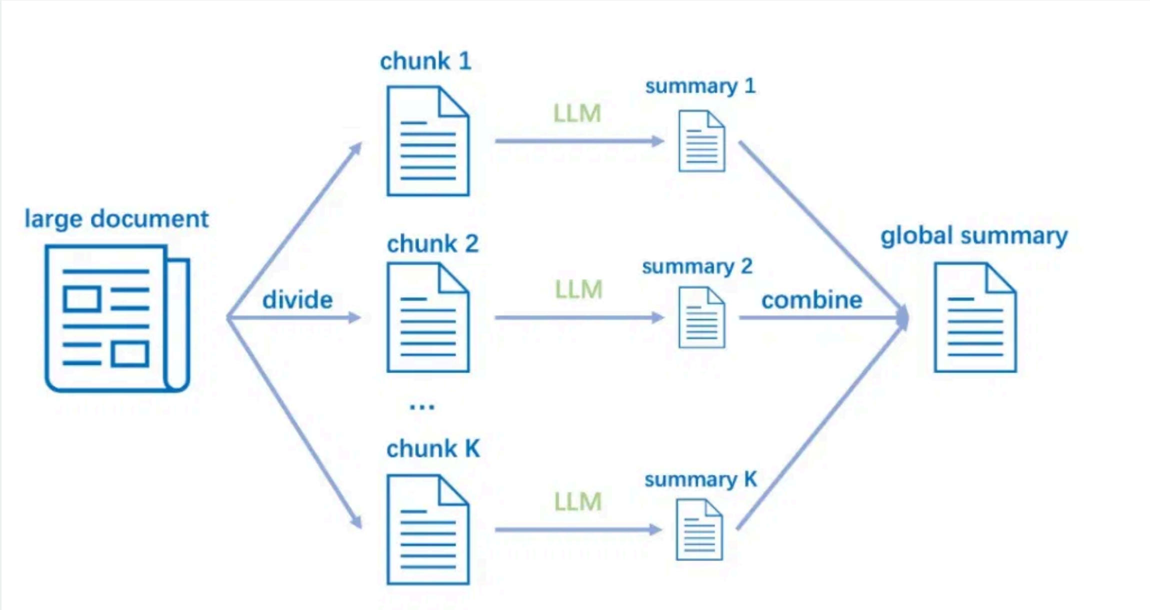
Treats individual functions or methods as discrete chunks, maintaining functional integrity.

**File-Level Chunking**

Uses entire source files as chunks, preserving module-level context but potentially exceeding token limits.

**Fixed-Size Chunking with Overlap**

Splits code into fixed-size segments with overlapping boundaries to maintain context across chunks.




## Implementation Considerations

- Chunk Size: Balance between context preservation and token limits
- Overlap Ratio: Typically 10-20% to maintain cross-chunk context
- Metadata Preservation: Include imports, class definitions, and dependencies
- Language-Specific Parsing: Use AST parsers for accurate semantic boundaries

# RAG Pipelines for Code Understanding

## RAG for Code

Retrieval-Augmented Generation (RAG) combines retrieval systems with generative models to enhance code understanding and generation by accessing relevant code snippets on demand.

**1. Code Indexing**

Parse, chunk, and embed code repository into a vector database, preserving metadata like file paths and dependencies.

**2. Query Processing**

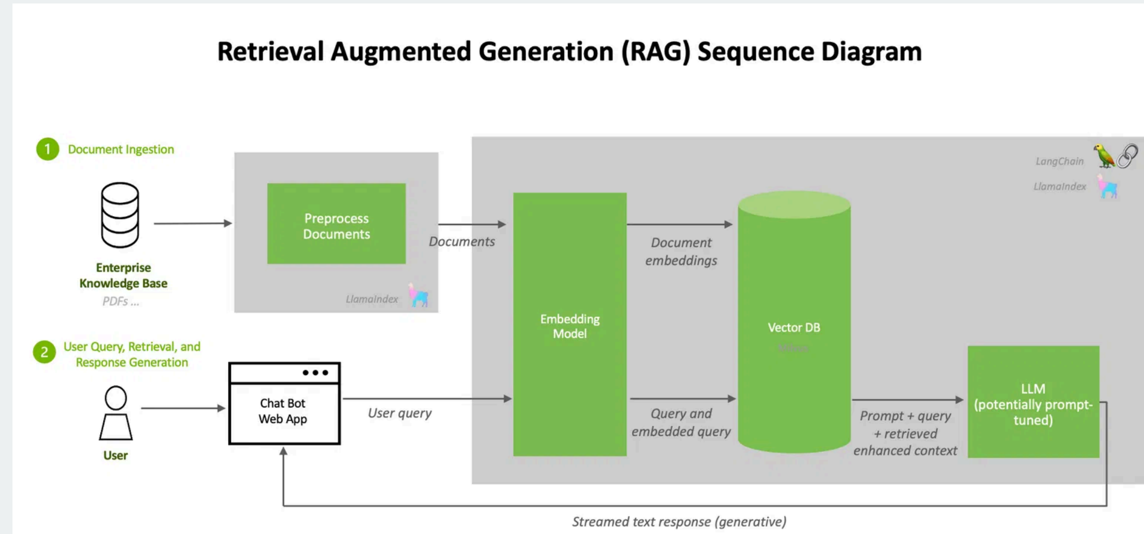
Transform user queries into code-aware embeddings that capture programming language semantics.

**3. Relevant Code Retrieval**

Retrieve the most semantically similar code chunks using vector similarity search.

**4. Context-Aware Generation**

Augment LLM prompt with retrieved code to generate contextually informed responses or code completions.



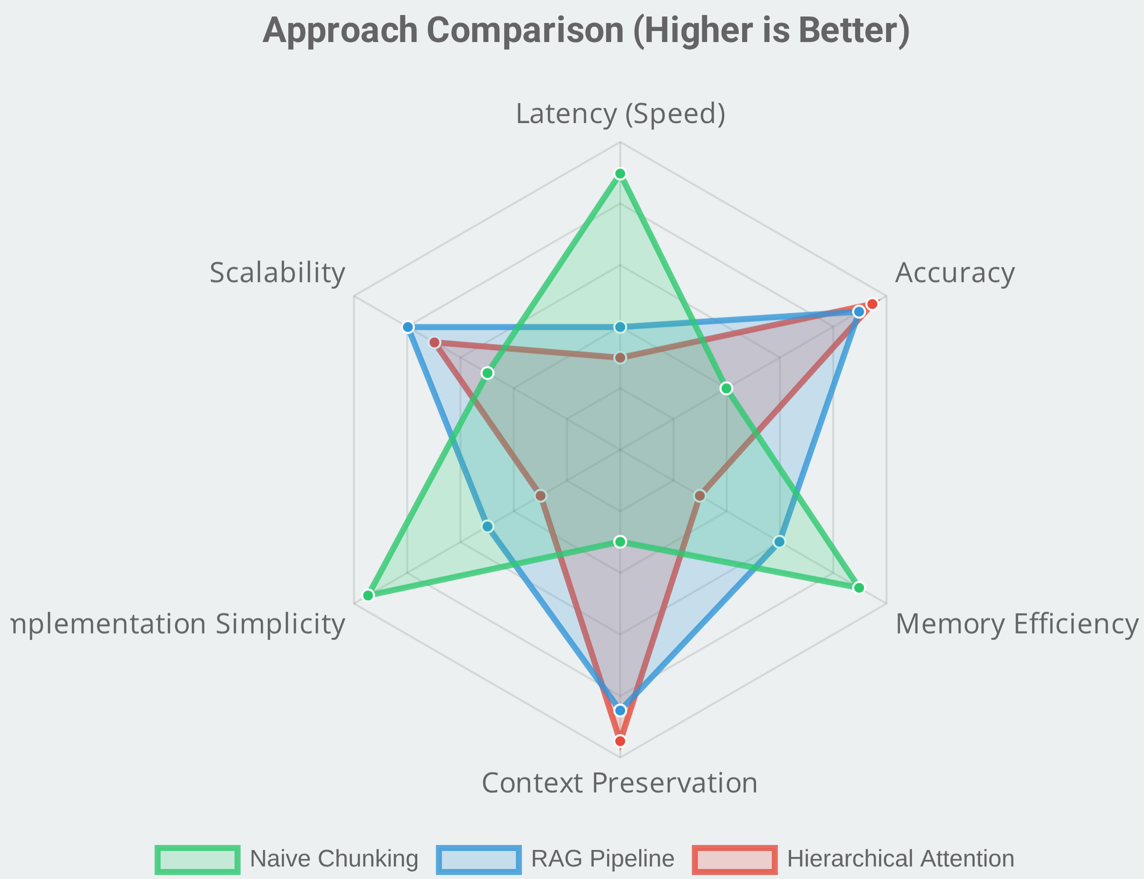

**Code-Specific RAG Enhancements:**

- AST-aware chunking to preserve syntactic structure
- Symbol tables to track variable and function references
- Dependency graphs to capture cross-file relationships
- Multi-stage retrieval for both broad and deep context

# Trade-offs Between Approaches

| Approach | Latency | Accuracy | Memory | Context Integrity |
|---|---|---|---|---|
| Naive Chunking | ★★★ | ★★ | ★★★ | ★ |
| Semantic Chunking | ★★ | ★★★ | ★★ | ★★★ |
| Sliding Window | ★★ | ★★★ | ★★ | ★★★ |
| RAG Pipeline | ★★ | ★★★ | ★★ | ★★★ |
| Hierarchical Attention | ★ | ★★★★ | ★ | ★★★★ |
| File-Tree Aware | ★★ | ★★★★ | ★★ | ★★★ |

**Key Insight:** No single approach is optimal for all scenarios. The best strategy often involves combining multiple techniques based on specific requirements and constraints.




## Decision Factors

- **Codebase Size**: Larger codebases benefit more from hierarchical approaches
- **Query Complexity**: Complex queries require better context preservation
- **Response Time**: Real-time applications may prioritize latency over accuracy
- **Hardware Constraints**: Limited resources may necessitate simpler approaches

# Limitations & Research Gaps

## Current Limitations

**Token Limits & Long-Range Dependencies**
Even with 128K context windows, models struggle with coherence across long sequences.

**RAG Limitations for Code**
Standard RAG approaches often fail to capture cross-file references and global state.

**Semantic Understanding Challenges**
Chunking strategies struggle with complex control flow and data dependencies.

**Evaluation Difficulties**
Lack of standardized benchmarks for long-context code understanding tasks.

## Promising Research Directions

**Code-Specific Attention Mechanisms**
Attention patterns that follow code structure rather than sequential proximity.

**Hybrid Symbolic-Neural Approaches**
Combining static analysis with neural methods for better program semantics.

**Persistent Memory Architectures**
External memory that persists beyond the immediate context window.

**Multi-Modal Code Representations**
Incorporating visual elements like control flow graphs alongside textual code.

# Best Practices & Emerging Methods

## Best Practices

**Choose the Right Approach**
Fine-tune for specialized tasks, RAG for knowledge-intensive tasks, prompt engineering for flexibility.

**Optimize Chunking Strategy**
Use language-specific parsers to create semantically meaningful chunks that preserve code structure.

**Implement Hybrid Approaches**
Combine techniques: fine-tune models to better utilize RAG, use hierarchical retrieval for large codebases.

**Rigorous Evaluation**
Test on diverse, real-world code samples with metrics for correctness, coherence, and execution.

## Emerging Methods

**Mixture-of-Experts (MoE)**
Models with specialized sub-networks that activate based on input type, enabling efficient scaling.

**Persistent Memory**
External memory systems that maintain information beyond the immediate context window.

**Flash Attention 2**
Optimized attention mechanisms that reduce memory usage and enable longer contexts with less compute.

**Code-Specific Architectures**
Models designed specifically for code with built-in understanding of programming language semantics.