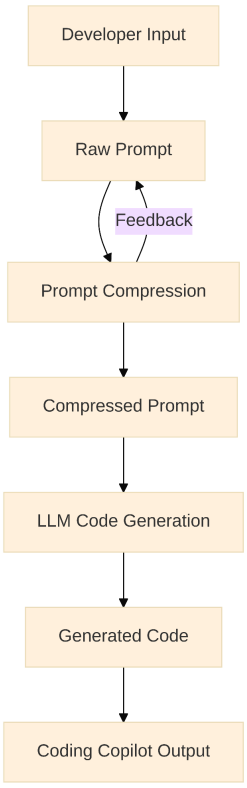
Optimizing Coding Copilots: Text-to-Prompt Compression Techniques and Their Mitigation Strategies

By: Khushi Malik

1.Summary

This report examines the critical role of prompt compression in enhancing the efficiency and effectiveness of Large Language Models (LLMs) when applied to coding copilots. It elucidates how the strategic shortening and optimization of input prompts can effectively address inherent LLM limitations, including token limits, computational costs, and processing latency, while simultaneously improving the quality of generated code.

The analysis identifies a spectrum of compression methodologies, ranging from fundamental, human-driven techniques such as information distillation and structured prompt design to sophisticated, model-based frameworks like LLMLingua and the code-specific CodePromptZip. A core finding is that code presents unique challenges for compression, primarily due to its strict semantic and syntactic requirements, where even minor information loss can lead to functional errors. The report details strategies to overcome these limitations, emphasizing the importance of iterative refinement, human-in-the-loop validation, and the integration of advanced compression techniques within Retrieval-Augmented Generation (RAG) workflows. Practical recommendations are provided for developers and engineering teams to implement robust prompt compression strategies, ensuring both cost-efficiency and the generation of high-quality, functionally correct code.



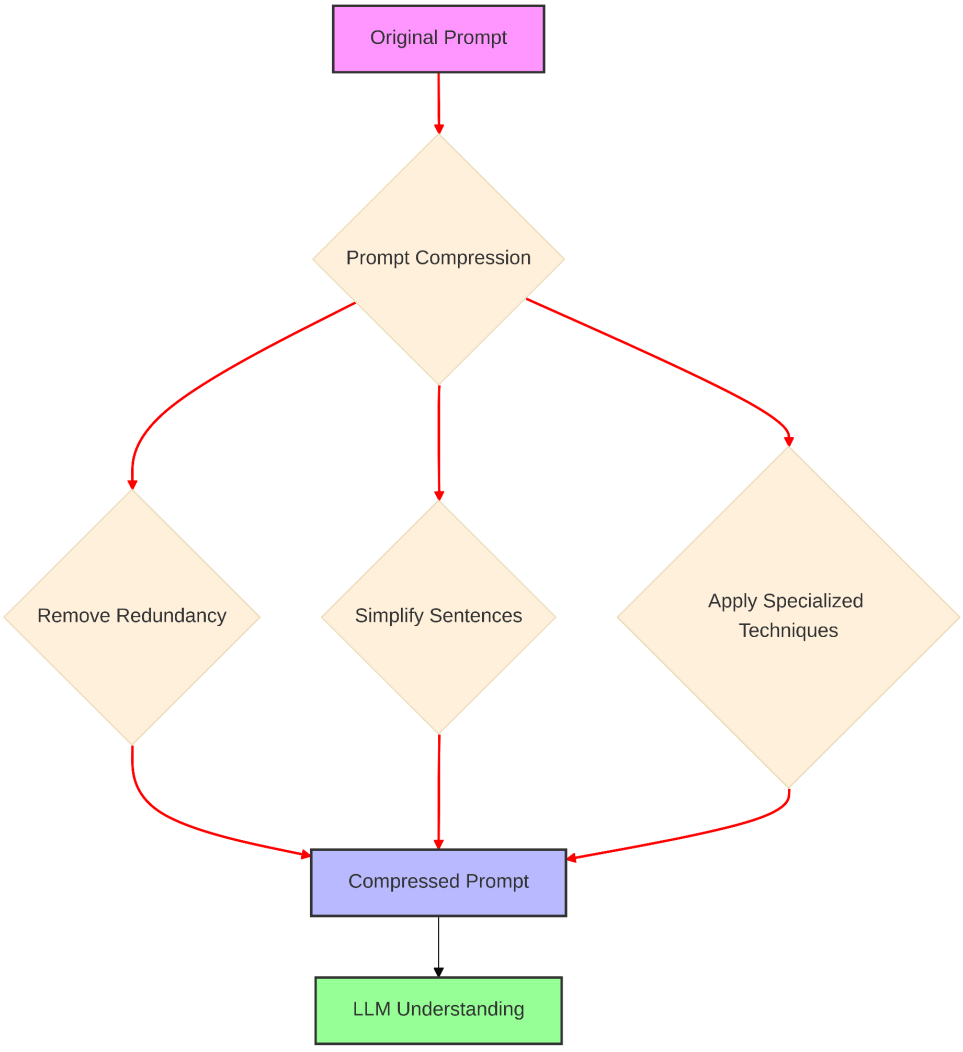
2. Introduction: The Imperative of Prompt Compression for LLM

What is Prompt Compression?

Prompt compression is the process of making the text sent to an LLM shorter and more efficient by:

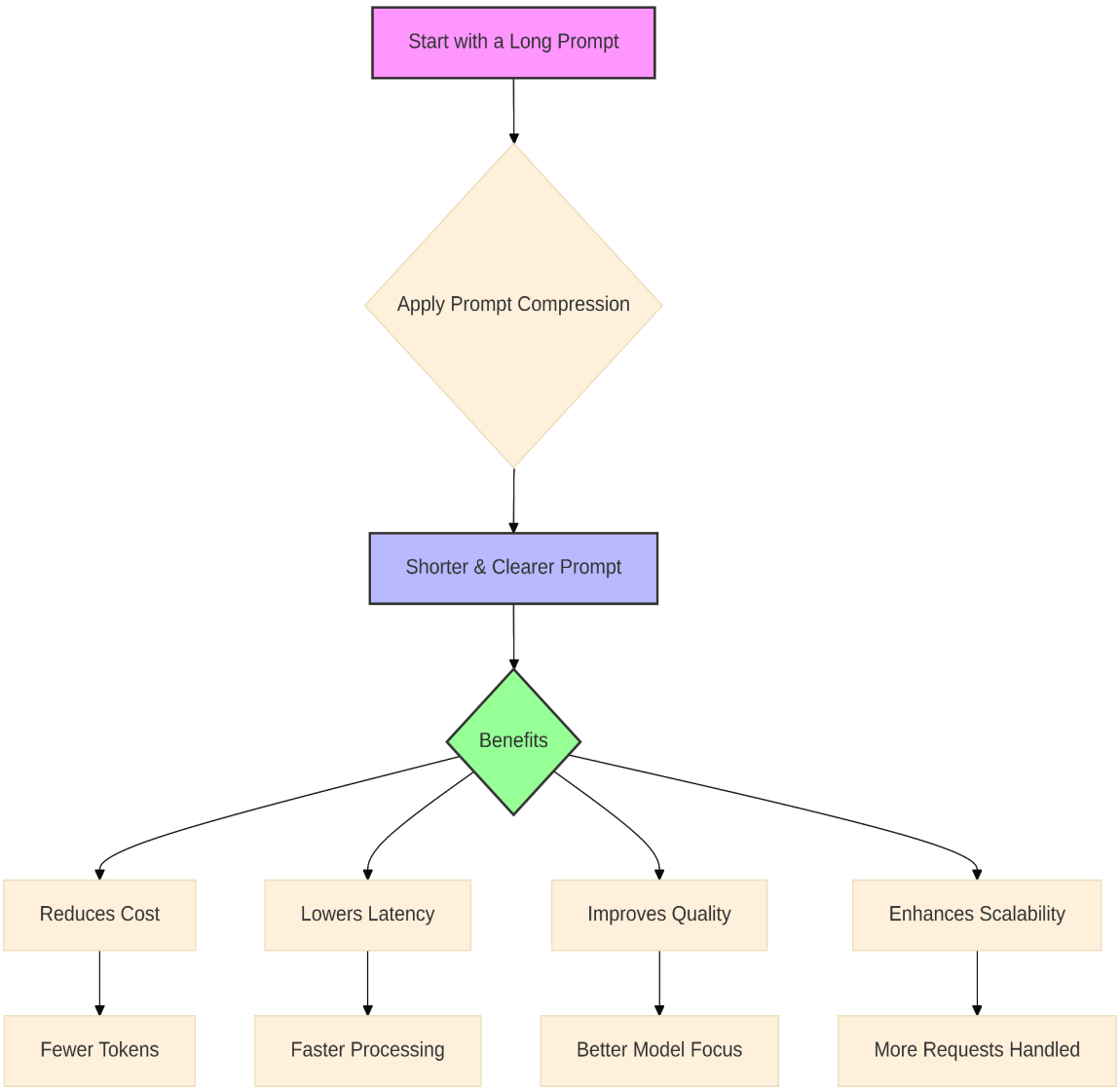
- **Removing extra words:** Eliminating unnecessary content.
- **Simplifying sentences:** Breaking down complex ideas.
- **Applying special techniques:** Using methods to pack more meaning into fewer words.

The guiding principle is **smart brevity**—every word should contribute to clarity and intent



Why is Prompt Compression Important?

- **Token limits:** LLMs can only process a finite amount of text at once; exceeding this can lead to incomplete responses.
- **Cost efficiency:** LLMs often charge per token, so shorter prompts reduce expenses.
- **Speed:** Shorter prompts are processed faster, resulting in quicker responses.
- Improved accuracy: Concise, clear prompts help LLMs generate more accurate answers.
- **Scalability:** Efficient prompts enable handling more requests, improving application performance

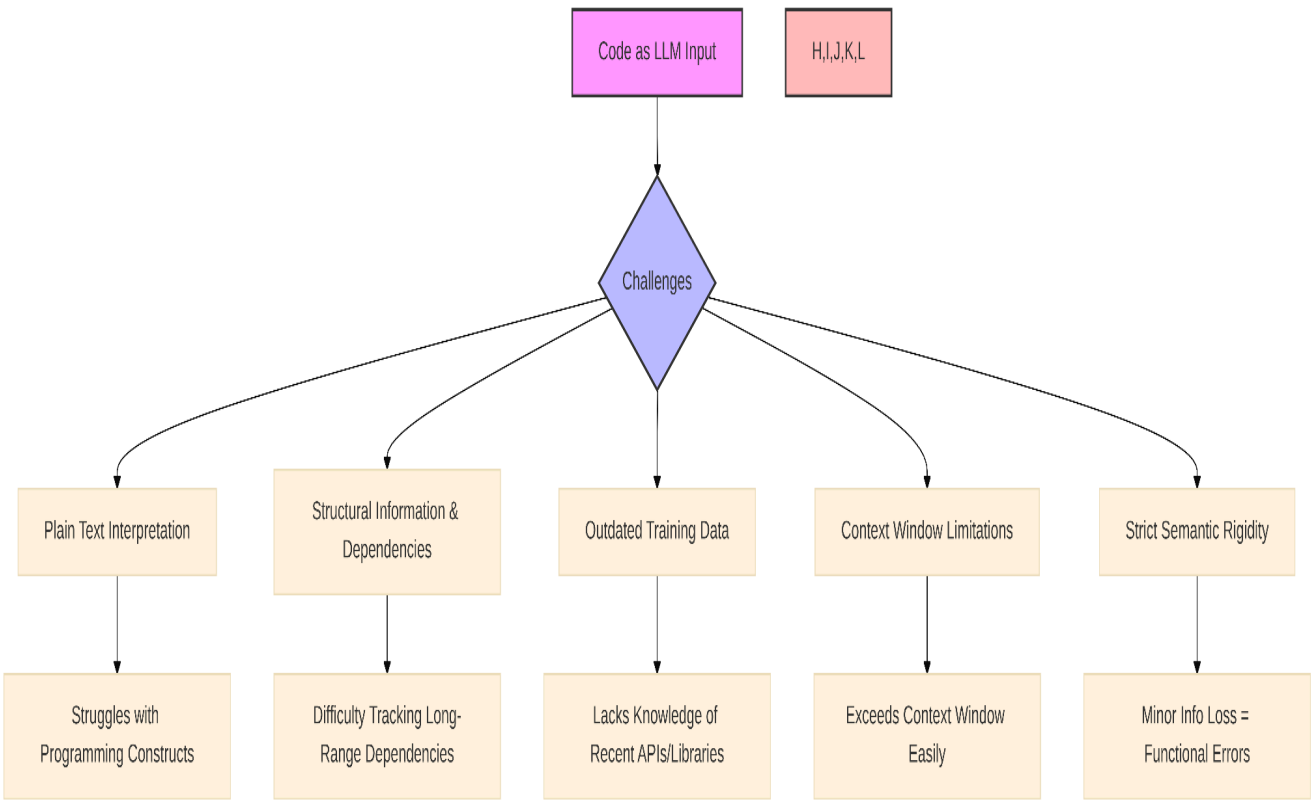


3. Prompt Compression in Coding Copilot Environments

3.1 Unique Characteristics and Challenges of Code as LLM Input

Why Code is Hard for LLMs

- **Structured data:** Code has logical structures (functions, loops) that LLMs see as plain text.
- **Hidden dependencies:** Variables and logic may span long distances in code, challenging LLMs' memory.
- **Outdated knowledge:** LLMs may lack awareness of recent programming features or library updates.
- **Large codebases:** Complex functions or projects can exceed LLM context windows.
- **Zero tolerance for error:** Unlike natural language, small code errors can break functionality



Note : *Code compression must preserve exact logic and structure to avoid functional errors.*

3.2 General Prompt Engineering Best Practices for Code Generation

How to Talk to LLMs About Code (Best Practices)

- **Start general, then specify:** State the main goal, then add details.
- **Provide examples:** Show input/output examples to clarify intent.
- **Break down tasks:** Divide complex tasks into smaller steps.
- **Be precise:** Use explicit references (e.g., function names).
- **Explain new concepts:** Briefly describe uncommon libraries or features.
- **Manage context:** Only include relevant files or code sections.
- **Keep chat history clean:** Start fresh for new tasks.
- **Iterate:** Refine prompts based on LLM feedback.
- **Maintain clean code:** Well-organized code aids LLM understanding

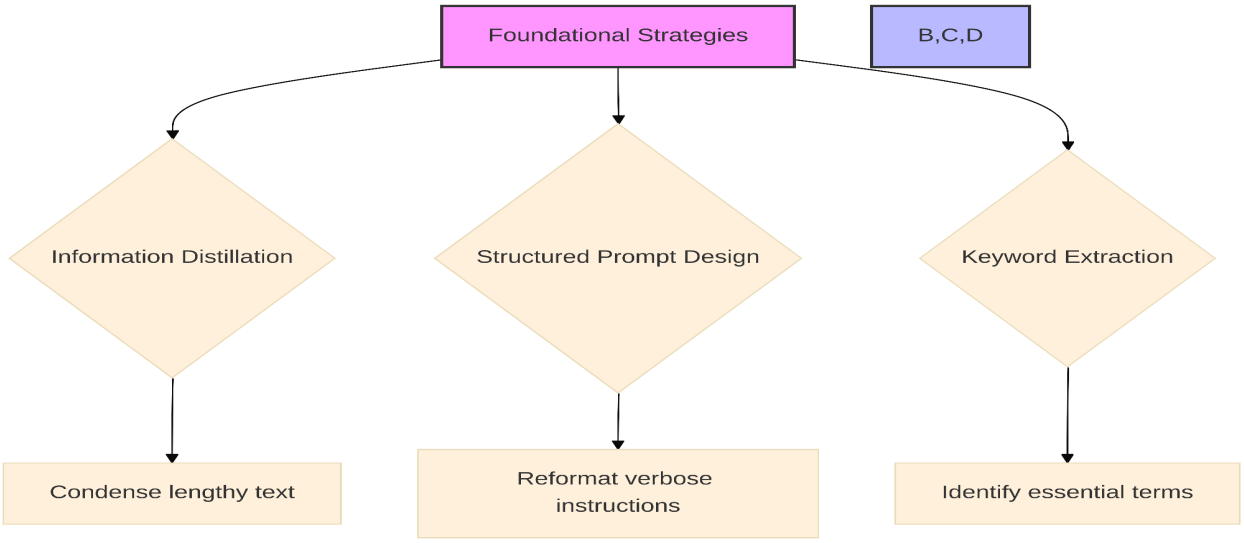
In short: *Getting LLMs to generate good code isn't just about making prompts shorter. It's about being smart and strategic with what you tell them, ensuring they have the right information, not just all the information.*

4. Techniques for Text-to-Prompt Compression in Code

4.1 Foundational Strategies: Information Distillation, Structured Prompt Design, Keyword Extraction

- **Information Distillation:** Condense descriptions to their essential meaning.
    - **Example:** Instead of "Please provide a detailed explanation of how photosynthesis works in plants," a distilled prompt would be "Explain photosynthesis in plants."
- **Structured Prompt Design:** Use bullet points or keywords for clarity.
    - **Example:** Rather than "Can you give me a comprehensive summary of the book along with its key themes and main characters?", a structured prompt would be "Book summary: key themes & main characters."
- **Keyword Extraction:** Retain only the most critical terms or concepts
    - **Example:** For "Describe the economic impacts of climate change on developing countries," the extracted keywords might be "Climate change, economic impact,

developing countries."



4.2 Advanced Frameworks and Tools: LLMLingua, CodePromptZip, and Others

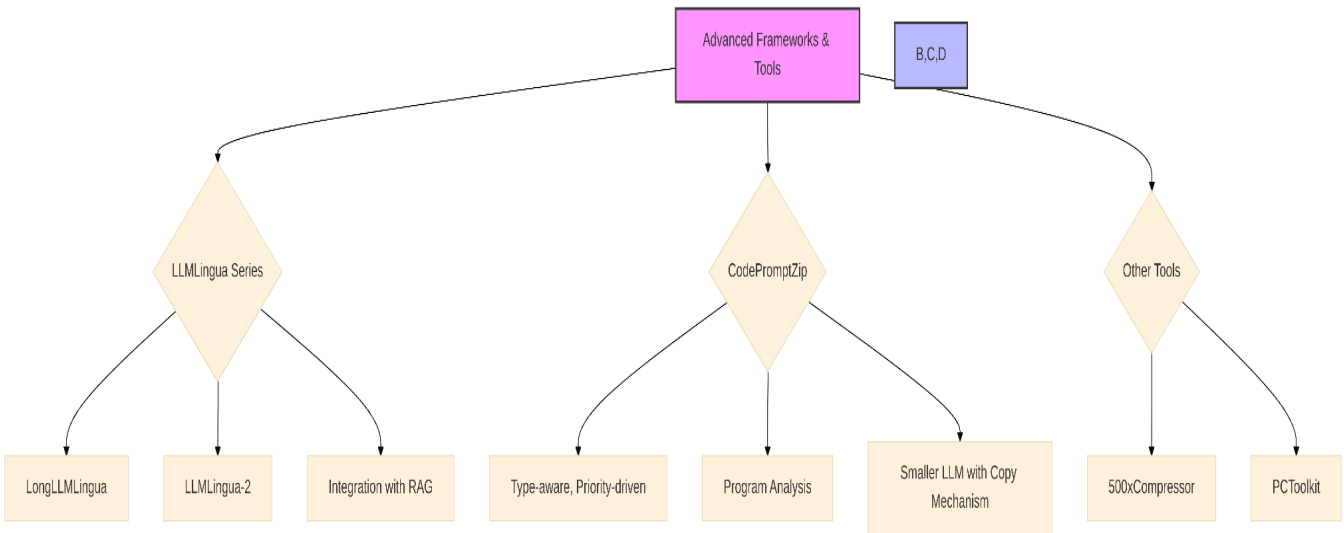
Framework/Tool	**Description & Use Case**
LongLLMLingua	Preserves vital information in very long prompts.
LLMLingua-2	Generalized, task-agnostic compression via data distillation.
CodePromptZip	Specialized for code, uses program analysis to prioritize token removal while preserving critical code elements.
500xCompressor	Designed for significant reduction in natural language contexts.
PCToolkit	Plug-and-play prompt compression for LLM efficiency.

Integration: These tools seamlessly integrate into Retrieval-Augmented Generation (RAG) pipelines, optimizing token usage and compressing documents within retrieval systems. This trend highlights a shift towards adaptive and context-aware compression, which is essential for diverse code-related tasks.

Methodology: CodePromptZip employs a type-aware, priority-driven strategy. It uses program analysis to rank the removal priorities of different code token types (e.g., Identifiers, Keywords, Operators), ensuring that critical elements are preserved.

Compressor: It trains a smaller LLM (such as CodeT5) with a copy mechanism, allowing it to directly copy essential tokens. This approach minimizes performance degradation while achieving significant compression.

Impact: This framework has demonstrated substantial improvements over entropy-based (like LLMLingua) and distillation-based baselines, signaling a move towards semantically-informed code compression that understands the underlying logic of code.



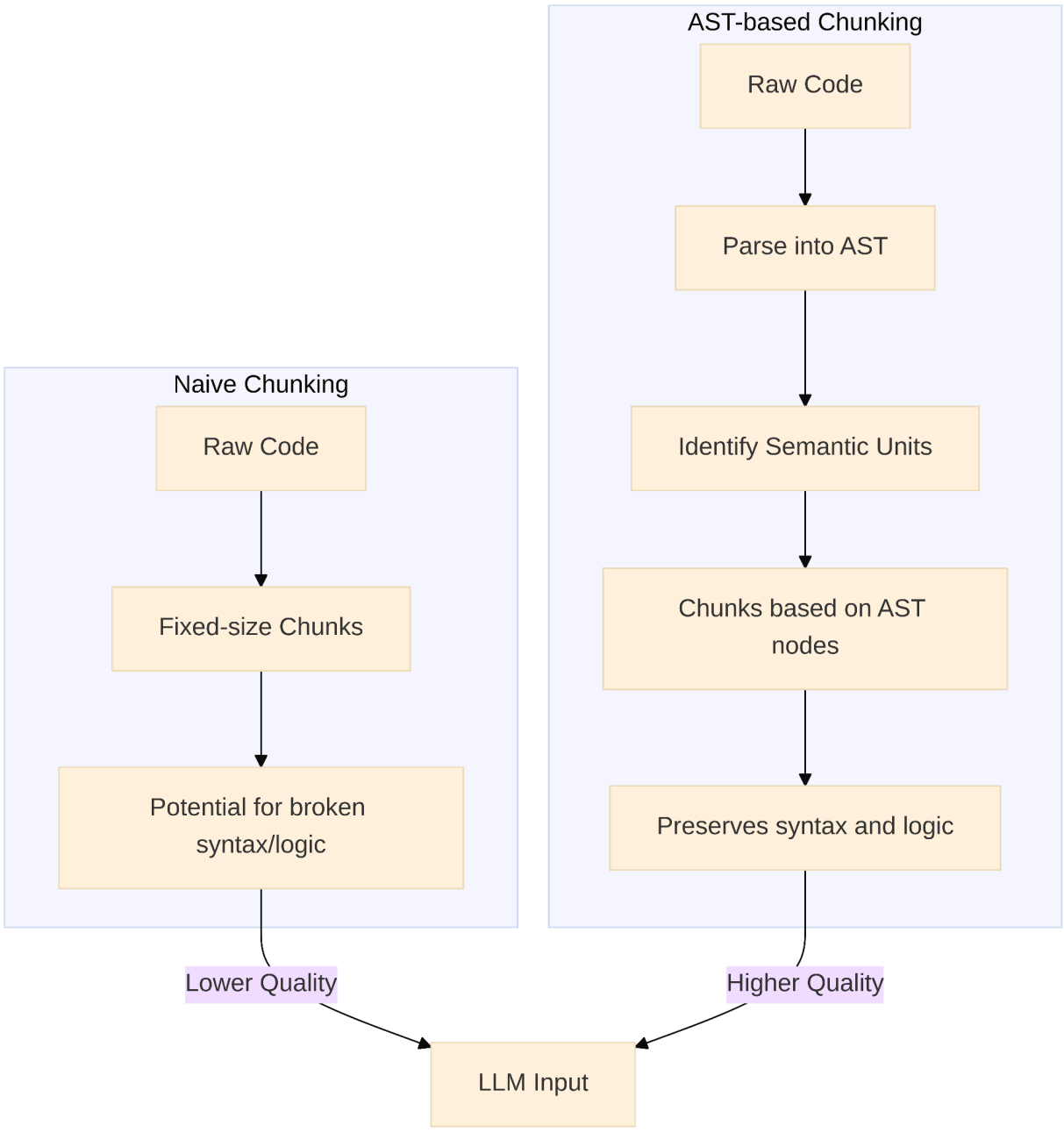
4.3 Leveraging Abstract Syntax Trees (ASTs) for Code-Specific Compression

Abstract Syntax Trees (ASTs) provide a structured, hierarchical representation of code, offering a powerful mechanism to overcome inherent LLM limitations in understanding code's complex logic and structure.

Why Use ASTs?

- They divide code **at meaningful points** (like functions, loops).
- They help avoid syntax errors.
- They allow AI to **insert or modify code accurately**.
- They preserve code structure when compressing or editing.
    - **Tools: Tree-Sitter** → Parses your code and builds ASTs automatically.

```
import ast

code = """

def greet(name):

    print("Hello", name)

"""

tree = ast.parse(code)

print(ast.dump(tree, indent=4))
```


AST-based Chunking
Raw Code
Parse into AST
Identify Semantic Units
Chunks based on AST nodes
Preserves syntax and logic
Naive Chunking
Raw Code
Fixed-size Chunks
Potential for broken syntax/logic
Lower Quality
Higher Quality
LLM Input

4.4 Code Embeddings and Contextual Compression in RAG Pipelines

RAG enables LLMs to reference external knowledge bases, such as code examples or documentation.When combined with code embeddings and contextual compression, RAG becomes exceptionally effective for managing large codebases.

- Contextual compression ensures only the most relevant, concise information is returned, preventing context window overflow.
- **LangChain's ContextualCompressionRetriever** is an example implementation, using a pipeline of chunking, filtering, and reranking steps.
- Code embeddings (often AST-based) allow efficient retrieval of semantically similar code snippets

1. **Input:** You ask:
   `"How to create a login form in Flask?"`
2. **RAG Process:**
   - **Searches docs** for Flask login code.
   - **Embeds** each code chunk using AST-based embeddings.
   - **Compresses** the search results with:
   a. `CharacterTextSplitter` – splits large files.
   b. `EmbeddingsRedundantFilter` – removes duplicates.
   c. `EmbeddingsFilter` – picks most relevant chunks.
   d. `LongContextReorder` – moves the best stuff to top/bottom.
   e. `LLMListwiseRerank` – (optional) reorders based on AI scoring.
3. **LLM generates a clean, accurate Flask login example**.

DocumentCompressorPipeline Components:

1. **CharacterTextSplitter:** Breaks large text/code into manageable chunks.

```
from langchain.text_splitter import CharacterTextSplitter

splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=100)

chunks = splitter.split_text(long_code_or_doc)
```

2. **EmbeddingsRedundantFilter:** Filters out near-duplicates using cosine similarity.

```
from langchain.retrievers.document_compressors import EmbeddingsFilter

filter = EmbeddingsFilter(embeddings=OpenAIEmbeddings(), similarity_threshold=0.9)

unique_chunks = filter.compress_documents(chunks, query="How to write async code in Python?")
```

3. **EmbeddingsFilter:** Keeps only chunks semantically close to the query. Lightweight and fast.

   Example use-case: ***Retrieving relevant PyTorch examples*** without wasting tokens.

4. **LongContextReorder:** Tackles the "lost in the middle" LLM issue by sandwiching high-value chunks front and end.

```
from langchain.retrievers.document_compressors import LongContextReorder

reorder = LongContextReorder()

ordered_docs = reorder.compress_documents(unique_chunks, query="fine-tune transformers")
```

5. **LLMChainExtractor/LLMListwiseRerank:** Heavy hitters. Use LLMs to **extract or rerank** content based on relevance. Best when accuracy > cost.

```
from langchain.retrievers.document_compressors import LLMChainExtractor

extractor = LLMChainExtractor.from_llm(ChatOpenAI())

compressed = extractor.compress_documents(docs=chunks, query="explain dropout in neural networks")
```

6. **Code Embeddings:** Capture structure and meaning of code for smarter retrieval.

   Use: `starcoder2`, `codeBERT`, or `OpenAI's code-search embeddings`.

   ```
   # Example with HuggingFace code embedding
   from sentence_transformers import SentenceTransformer
   model = SentenceTransformer('microsoft/codebert-base')
   embeddings = model.encode(code_snippets)
   ```

7. **Powerful Architectural Pattern:** The combination of RAG with code-aware compression and embeddings creates a dynamic, context-aware retrieval and compression system. This system is invaluable for effectively managing large and evolving codebases within the inherent context window limitations of LLMs.

Comparative Table of Prompt Compression Techniques for Code

Technique Name	Description	Applicability to Code	Typical Compression Ratio/Effectiveness	Key Features/Mechanism	Primary Benefits	Key Limitations/Considerations
Information Distillation	Condenses lengthy text into concise summaries, retaining core message.	General text, conceptual code descriptions.	Modest reduction.	Human-driven summarization, trimming non-essential details.	Improved focus, reduced verbosity.	Risk of over-simplification, requires human judgment.
Structured Prompt Design	Reformats verbose instructions into bullet points/commands, uses keywords.	General text, high-level code instructions.	Modest reduction.	Human-driven reformatting, keyword usage.	Enhanced clarity, better LLM focus.	May not be sufficient for complex code.

Keyword Extraction	Identifies and retains only essential terms.	Information retrieval, search-related applications.	Modest reduction.	Human-driven identification of core concepts.	Efficient for search, reduces noise.	High risk of context loss for generation tasks.
LLMLingua Series	Framework for compressing prompts while maintaining output quality.	General text, can be adapted for code.	Significant (e.g., 20-25% reduction for code contexts).[11]	Model-based token pruning, data distillation (LLMLingua-2), identifying important info (LongLLMLingua).	Cost savings, reduced latency, improved efficiency.	Potential for information loss, requires tuning.
CodePromptZip	Code-specific framework for compressing code examples in RAG.	Code-specific (e.g., AssertionGen, Bugs2Fix, CodeSuggestion).	Significant (25-40% reduction for DocStrings)[8], 23.4-28.7% improvement over baselines in tasks.[11]	Type-aware, priority-driven program analysis, small LM compressor with copy mechanism.	Preserves code semantics, high accuracy, flexible compression ratio.	Requires code-specific training data, computational overhead for compressor.
AST-based Chunking	Splits code at semantically meaningful boundaries (e.g., functions).	Structural code analysis, RAG pre-processing.	Improves context relevance, not direct token reduction.	Program analysis, tree-sitter for parsing.	Preserves syntactic validity, better code understanding for LLMs.	Requires parsing overhead, not a compression method itself but enables better chunking.
RAG with Contextual Compres	Dynamically retrieves and compresses relevant	Large codebases, API documentation,	Varies based on compressor, e.g., 2.37x token	Base retriever + DocumentCompressorPipeline (splitters, filters, reorderers).	Addresses knowledge cutoff,	Complexity of pipeline setup, potential for irrelevant retrieval.

sion	external knowledge.	dependencies.	reduction for evaluation.[19]		reduces hallucinations, dynamic context.	

5. Limitations and Challenges in Code Prompt Compression

5.1 The Risk of Semantic Information Loss and Context Dilution

- **Core challenge:** Aggressive compression can result in loss of essential context, degrading LLM response quality.
- **For code:** Even minor semantic loss (e.g., variable type, function signature) can cause severe errors, compilation failures, or runtime bugs.
- **Implication:** Code compression methods must be highly precise and context-aware due to a low tolerance for error

5.2 Impact on Code Generation Quality and Accuracy (e.g., Logical and Syntactic Errors)

- **Logical Errors:** LLMs frequently misinterpret precise logical requirements, leading to incorrect or nonsensical code behavior (e.g., incorrect conditions, erroneous constant values, mistakes in mathematical operations).
- **Syntactic Errors:** Common issues include:
    - Omission of important code sections, resulting in incomplete or fragmented outputs.
    - Failure to grasp full contextual nuances, leading to code that doesn't align with intended use.
    - Specific technical errors like incorrect function arguments, missing code blocks, or improper syntax.
    - Incorrect references to variables or functions (undefined names).
- **Empirical Observations on DocString Compression:**
    - State-of-the-art methods achieve only about 10% reduction in code comments (DocStrings).
    - Further reductions beyond 10% lead to significant degradation in code generation performance.
- **Prompt Length "Sweet Spot":**
    - Shorter prompts (under 50 words) tend to yield better performance from LLMs.
    - Longer prompts (exceeding 150 words) significantly increase error rates, often

resulting in "garbage code" or meaningless snippets.

- **Context Dilution:** Overly long prompts can cause context dilution, where simply providing more information without intelligent compression or structuring becomes counterproductive. This reduces model focus and increases error rates.
- **Conclusion:** Intelligent compression is crucial not just for cost savings but fundamentally for maintaining and improving code generation quality, as excessive uncompressed context can actively harm LLM performance.

Common Errors in LLM-Generated Code from Prompt Compression

Error Category	Specific Error Type	Description	Impact on Code	Example (Conceptual/Illustrative)
Semantic Errors	Logical Errors	Misinterpretation of task requirements leading to incorrect logic.	Incorrect runtime behavior, unexpected program flow.	Code generates sum = a - b instead of sum = a + b due to misinterpretation of "calculate total".
	Incomplete Code	Important sections of code are omitted.	Non-functional code, compilation errors, runtime crashes.	A function body is missing a critical loop or conditional block.
	Condition Errors	Missing or incorrect conditions in control flow statements.	Incorrect branching, infinite loops, logic flaws.	if (x > 0) becomes if (x) or if (x < 0) due to compressed prompt.
	Constant Value Errors	Incorrect constant values set in arguments, assignments.	Hard-to-debug logical errors, incorrect calculations.	A prompt to set a MAX_RETRIES to 5 results in MAX_RETRIES = 10.

	Reference Errors	Incorrect references to variables or functions (e.g., undefined names).	Compilation errors, runtime exceptions (e.g., NameError).	A compressed prompt removes the definition of a variable user_id, leading to its undefined use.
	Operation/Calculation Errors	Mistakes in mathematical or logical operations.	Incorrect results, data corruption.	A prompt asking for average calculation results in total / 2 instead of total / count.
Context Misunderstanding	Misunderstanding Context	Failure to grasp the full context of the prompt or surrounding code.	Code not aligning with intended use, integration issues.	Code generated for a specific framework function but misses crucial context about its asynchronous nature.
General Quality	Bad Code/Meaningless Snippets	Generation of irrelevant, nonsensical, or "garbage" code.	Wasted tokens, increased debugging time, non-usable output.	LLM generates random lines of code or non-sequiturs instead of a coherent function.

5.3 Complexity of Preserving Code Logic and Dependencies

- **Complexity:** Some coding tasks are very complicated. If we try to make them too simple, we might miss important details and the code won't work.
- **Need for Experts:** Summarizing code well needs someone who really understands programming and the project—like a skilled mechanic knowing which parts of an engine are essential.
- **Long Dependency Problem:** In code, one part often depends on another part that might be far away. For example, a variable at the top might be used much later. Removing something small can break the code.
- **Why Simple Methods Fail:** Methods that just look for common words or phrases don't work for code. They miss the deeper logic and connections.
- **Deep Understanding Needed:** We need to look at how the code is built (using tools like ASTs) and how it works, not just the words. This helps keep the code working after compression.

- **Code vs. Regular Text:** Unlike normal text, code needs to keep distant but important definitions and logic.
- **Conclusion:** Good code compression must understand both what the code means and how it's built.

6. Strategies to Overcome Limitations and Optimize for Efficiency

1. Iterative Prompt Refinement and Human-in-the-Loop Optimization

- **Continuous Improvement:**
  Getting the best results from AI is a step-by-step process. You rarely get a perfect prompt (instruction) on the first try.
    - **Start with a Clear Prompt:** Write exactly what you want the AI to do.
    - **Check the Output:** Look at the AI's answer for mistakes or missing details.
    - **Refine the Prompt:** Add more details or examples if needed.
    - **Keep Records:** Save different prompt versions and AI responses to track what works best.
    - **Get Feedback:** Ask others for suggestions and improvements.
- **Human-in-the-Loop (HITL):**
  This means humans are involved throughout the process, not just at the end.
    - **Teamwork:** Use tools that let teams work together on prompts and keep track of changes.
    - **Expert Review:** For important code, have experts check and improve the AI's work.
    - **Why It Matters:** AI can be unpredictable, especially with code. Human review helps catch mistakes and improve results

2. Domain-Specific and Type-Aware Compression Methodologies

Domain-specific and type-aware compression is about using smart methods that truly understand how code works, rather than treating it like plain text. These methods focus on keeping the important parts of code while removing unnecessary details, all without breaking the code.

- ### Why Is It Needed?
    - **Deep Understanding Required:**
      To compress code effectively, you need to know which parts are essential for the code to work. This requires real programming knowledge, not just language skills.
    - **Not Just Words:**
      Simple word-based compression doesn't work well for code because code has structure and rules that must be followed.

- **How Does It Work?**
    - **Example:** CodePromptZip
      This tool is designed to understand the different parts of code and decide what can be safely removed.
    - **Program Analysis:**
      The method looks at elements like:
        - Identifiers (variable and function names)
        - Keywords (like if, for, while)
        - Operators (like +, -, =)
        - Comments (explanations in the code)
- Prioritization:
  The system ranks code parts by how important they are. It removes less important parts first, aiming to reduce size without harming how the code works.
- Bigger Chunks, Not Just Words:
  Instead of deleting single words, these methods often remove or keep whole lines or blocks of code to preserve meaning and structure.

How Is This Different from Regular Compression?

- **Type-Aware:**
  The method knows the difference between a variable name and a comment, so it can make smarter choices.
- **Domain-Specific:**
  It's built for code, not for regular text, so it understands programming rules and logic.

Where Is This Heading?

- **Smarter AI Tools:**
  Future code compression will use AI models that understand code almost like a compiler does. They will analyze code structure, syntax, and meaning to make sure compressed code still works properly.
- **Knowledge-Based Compression:**
  This is a move away from treating code as just a string of characters, and towards using real knowledge about programming to make better decisions.

3. Advanced Context Management and Project Structure for LLM Inputs

- Holistic Information Management: Organize the entire software project to provide only the most relevant and minimal context to the AI.
- Leveraging ASTs: Use ASTs to break code into logical, meaningful chunks, helping the

AI understand methods, properties, and logic.

- Strategies for Large Codebases: Modularize projects and allow AI to retrieve relevant code intelligently.
- Integration with Development Workflows:
    - Readme-driven development: Use README files as high-level guides for context.
    - Task Decomposition: Break coding tasks into smaller, manageable steps.
    - IDE Context Management: Open/close relevant files to control context.
    - Explicit Context Provision: Use workspace/project tags in chat to provide project-wide context.

4. Integrating Compression within Retrieval-Augmented Generation (RAG) Workflows

- RAG Enhancement: Combine AI generation with dynamic retrieval of relevant code and documentation.
- Context Window Optimization: Compress retrieved information before it enters the LLM, maximizing information density and minimizing computational load.
- LangChain's ContextualCompressionRetriever:
    - Dynamic Compression: Compresses retrieved documents based on query relevance.
    - Pipeline Components:
        - CharacterTextSplitter: Breaks large code into manageable chunks.
        - EmbeddingsRedundantFilter: Removes duplicates using embeddings.
        - EmbeddingsFilter: Selects relevant documents based on semantic similarity.
        - LongContextReorder: Prioritizes the most relevant information at the beginning and end of the input.
- Significance: This approach enables efficient handling of large, evolving codebases within LLM context limits, ensuring only the most relevant information is processed

7. Conclusions and Recommendations

- Prompt Compression Is Essential:
    - It's not just about saving money. Compressing prompts makes AI coding tools work faster, more accurately, and at a larger scale by avoiding long, expensive inputs.
- Code Needs Special Treatment:
    - Code is different from normal text. It has strict rules and connections. Simple summarizing can break the code or make it wrong.
- Don't Use Overly Long Prompts:

    - Very long prompts can actually make code suggestions worse. Keep instructions focused and short.

Most Effective Strategies

- Use Human-Driven Prompt Engineering:
    - Always use basic techniques like:
        - Cutting out extra information
        - Organizing prompts clearly (bullets, steps)
        - Highlighting important keywords
        - These help the AI understand what you want.
- Leverage Advanced Code Tools:
    - Use modern tools like LLMLingua and especially CodePromptZip.
    - CodePromptZip is great because it knows which parts of code are most important and safely removes the rest.
- Integrate Code Structure Understanding:
    - Use Abstract Syntax Trees (ASTs) to break code into logical pieces before sending it to the AI.
    - This keeps the code's meaning and connections intact.
- Adopt Dynamic Context Management (RAG):
    - Use systems that fetch only the most relevant code or info for the AI right when it's needed.
    - These systems also compress that info, so the AI only sees what's important.
- Iterate and Use Human Oversight:
    - Test prompts, check the AI's output, and keep improving.
    - Always have experts review important code results to catch mistakes.

Works cited

1. medium.com, accessed on July 1, 2025, https://medium.com/@sahin.samia/prompt-compression-in-large-language-models-llms-making-every-token-count-078a2d1c7e03#:~:text=Prompt%20compression%20is%20the%20process,meaning%20and%20context%20remain%20intact.
2. Prompt Compression in Large Language Models (LLMs): Making Every Token Count | by Sahin Ahmed, Data Scientist | Medium, accessed on July 1, 2025, https://medium.com/@sahin.samia/prompt-compression-in-large-language-models-llms-making-every-token-count-078a2d1c7e03
3. GenAI: How to Reduce Cost with Prompt Compression Techniques - SitePoint, accessed on July 1, 2025, https://www.sitepoint.com/prompt-compression-reduce-genai-apps/
4. Code summarization with Granite | IBM, accessed on July 1, 2025,

https://www.ibm.com/think/tutorials/code-summarization-with-granite
5. Semantic Compression With Large Language Models - Computer Science, accessed on July 1, 2025, https://www.cs.wm.edu/~dcschmidt/PDF/Compression_with_LLMs_FLLM.pdf
6. Enhancing LLM Code Generation with RAG and AST-Based Chunking | by VXRL - Medium, accessed on July 1, 2025, https://vxrl.medium.com/enhancing-llm-code-generation-with-rag-and-ast-based-chunking-5b81902ae9fc
7. How Abstract Syntax Trees Unlock LLM's Code Understanding | by ..., accessed on July 1, 2025, https://medium.com/@nishandanilka/how-abstract-syntax-trees-unlock-llms-code-understanding-5fa88877123a
8. Less is More: DocString Compression in Code Generation | Request PDF - ResearchGate, accessed on July 1, 2025, https://www.researchgate.net/publication/391760784_Less_is_More_DocString_Compression_in_Code_Generation
9. Prompt engineering for Copilot Chat - GitHub Docs, accessed on July 1, 2025, https://docs.github.com/en/copilot/concepts/prompt-engineering-for-copilot-chat
10. How to Optimize LLM Applications With Prompt Compression Using ..., accessed on July 1, 2025, https://www.mongodb.com/developer/products/atlas/prompt_compression/
11. CodePromptZip: Code-specific Prompt Compression for Retrieval-Augmented Generation in Coding Tasks with LMs - arXiv, accessed on July 1, 2025, https://arxiv.org/html/2502.14925v1
12. CODEPROMPTZIP: Code-specific Prompt Compression for ..., accessed on July 1, 2025, https://arxiv.org/abs/2502.14925
13. Getting LLMs to more reliably modify code- let's parse Abstract Syntax Trees and have the LLM operate on that rather than the raw code- will it work? I wrote a blog post, "Prompting LLMs to Modify Existing Code using ASTs" : r/programming - Reddit, accessed on July 1, 2025, https://www.reddit.com/r/programming/comments/1iqzcf6/getting_llms_to_more_reliably_modify_code_lets/
14. LLM generated code snippet merging into existing using ASTs : r/theprimeagen - Reddit, accessed on July 1, 2025, https://www.reddit.com/r/theprimeagen/comments/1idtjp2/llm_generated_code_snippet_merging_into_existing/
15. aws.amazon.com, accessed on July 1, 2025, https://aws.amazon.com/what-is/retrieval-augmented-generation/#:~:text=Retrieval%2DAugmented%20Generation%20(RAG),sources%20before%20generating%20a%20response.
16. How to do retrieval with contextual compression | LangChain, accessed on July 1, 2025, https://python.langchain.com/docs/how_to/contextual_compression/
17. Build a ContextualCompressionRetriever in Langchain · GitHub, accessed on July 1, 2025, https://gist.github.com/AlaGrine/930ad7da30bae36355f2a85bf3912699
18. Contextual Compression: LangChain | LlamaIndex | by Sourav Verma - Medium, accessed on July 1, 2025, https://medium.com/@SrGrace_/contextual-compression-langchain-llamaindex-7675c8d1f

9eb
19. PromptOptMe: Error-Aware Prompt Compression for LLM-based MT Evaluation Metrics - ACL Anthology, accessed on July 1, 2025, https://aclanthology.org/2025.naacl-long.592.pdf
20. Show HN: Min.js style compression of tech docs for LLM context | Hacker News, accessed on July 1, 2025, https://news.ycombinator.com/item?id=43994987
21. Semantic compression - Wikipedia, accessed on July 1, 2025, https://en.wikipedia.org/wiki/Semantic_compression
22. Using LLMs for Code Generation: A Guide to Improving Accuracy and Addressing Common Issues - PromptHub, accessed on July 1, 2025, https://www.prompthub.us/blog/using-llms-for-code-generation-a-guide-to-improving-accuracy-and-addressing-common-issues
23. Shorter prompts lead to 40% better code generation by LLMs : r/PromptEngineering - Reddit, accessed on July 1, 2025, https://www.reddit.com/r/PromptEngineering/comments/1do6wx4/shorter_prompts_lead_to_40_better_code_generation/
24. Iterative Prompt Refinement: Step-by-Step Guide - Ghost, accessed on July 1, 2025, https://latitude-blog.ghost.io/blog/iterative-prompt-refinement-step-by-step-guide/
25. Prompt Alchemy: Automatic Prompt Refinement for Enhancing Code Generation - arXiv, accessed on July 1, 2025, https://arxiv.org/html/2503.11085v1
26. Prompt Management Tool for Building LLM Apps - Humanloop, accessed on July 1, 2025, https://humanloop.com/platform/prompt-management
27. human-in-the-loop · GitHub Topics, accessed on July 1, 2025, https://github.com/topics/human-in-the-loop
28. From Reading to Compressing: Exploring the Multi-document Reader for Prompt Compression - arXiv, accessed on July 1, 2025, https://arxiv.org/html/2410.04139v2
29. Iterative Refinement of Project-Level Code Context for Precise Code Generation with Compiler Feedback - arXiv, accessed on July 1, 2025, https://arxiv.org/html/2403.16792v3
30. My LLM Code Generation Workflow (for now) - DEV Community, accessed on July 1, 2025, https://dev.to/simbo1905/my-llm-code-generation-workflow-for-now-1ahj