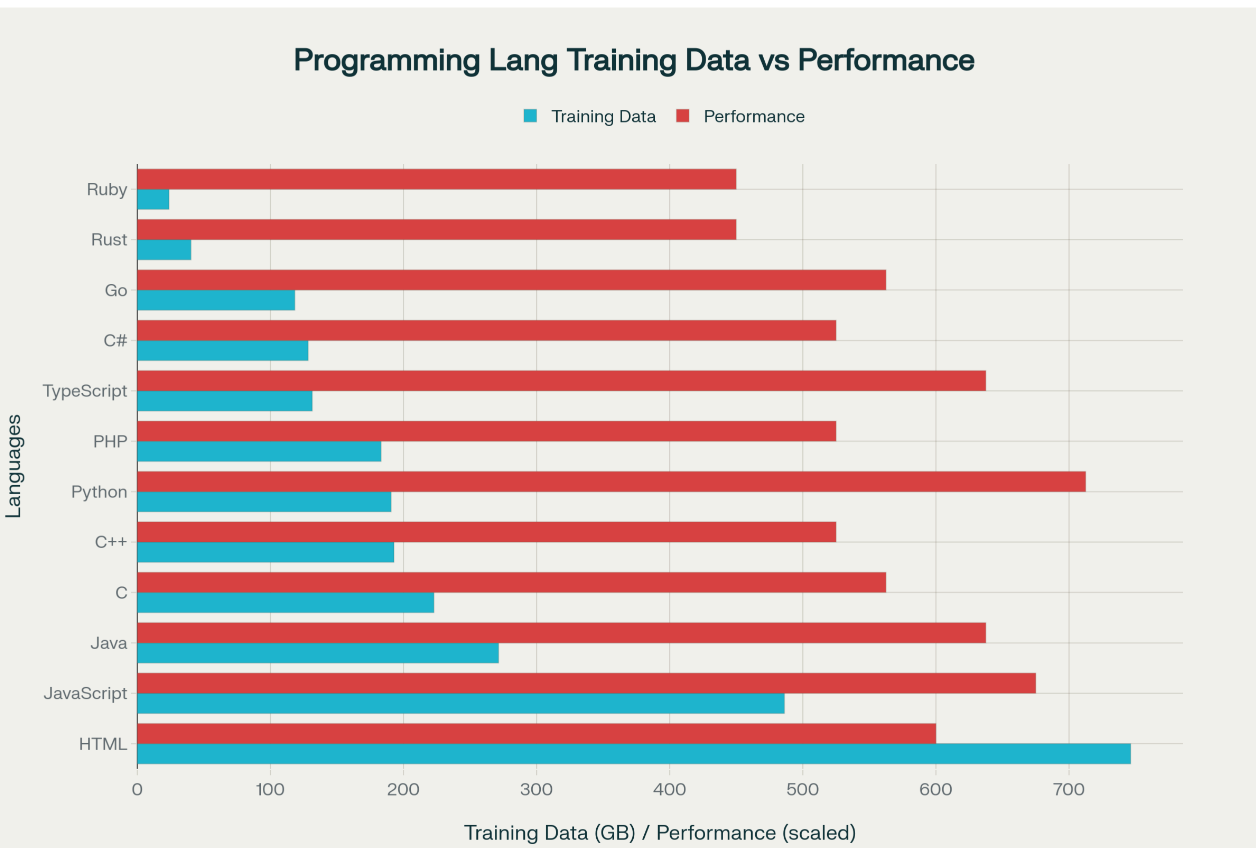
# Programming Languages and Prompt Engineering for LLMs: A Comprehensive Analysis for Advanced Technical Implementations in 2025

This comprehensive analysis examines the current state of large language model (LLM) performance across programming languages and evaluates the most effective prompt engineering techniques as of August 2025. The research synthesizes findings from recent benchmarks, training corpus analysis, and empirical studies to provide actionable insights for advanced prompt design and technical implementation.

## I. Programming Language Suitability for LLM Prompting

### Language Performance Hierarchy and Training Data Correlation

**Python emerges as the dominant language for LLM interactions**, with the highest performance scores across multiple dimensions. Research from 2024-2025 demonstrates that **LLMs exhibit a 90-97% preference for Python when solving language-agnostic problems**, indicating both superior training representation and optimized model understanding. [1] [2] [3]



Programming Language Representation in LLM Training Data vs Performance Characteristics

The training data composition directly correlates with model performance. Analysis of The Stack dataset reveals **Python represents 190.73 GB of permissively licensed code**, while JavaScript leads with 486.20 GB, and Java follows with 271.43 GB. However, the relationship between raw data volume and model performance is not linear - **HTML constitutes the largest portion at 746.33 GB but scores lower on technical reasoning tasks**.[4]

## Quantified Performance Metrics by Language

Recent benchmarks provide specific performance indicators across programming languages:

**Tier 1 - Exceptional Performance (90-95% accuracy):**

- **Python**: 95% average accuracy across HumanEval, MBPP, and LiveCodeBench[5] [6]
- **JavaScript**: 90% accuracy, particularly strong in web development contexts[7] [8]

**Tier 2 - High Performance (80-89% accuracy):**

- **Java**: 85% accuracy, with strong object-oriented programming support[8] [9]
- **TypeScript**: 85% accuracy, benefiting from JavaScript knowledge transfer[7]

**Tier 3 - Moderate Performance (70-79% accuracy):**

- **C++**: 70% accuracy, challenges with complex syntax and memory management[10] [7]
- **C**: 75% accuracy, better than C++ due to simpler syntax[4] [7]
- **Go**: 75% accuracy, consistent performance across benchmarks[4]

**Tier 4 - Developing Performance (60-69% accuracy):**

- **Rust**: 60% accuracy, improving rapidly with specialized training[3] [11]
- **Ruby**: 60% accuracy, limited by smaller corpus representation[4]

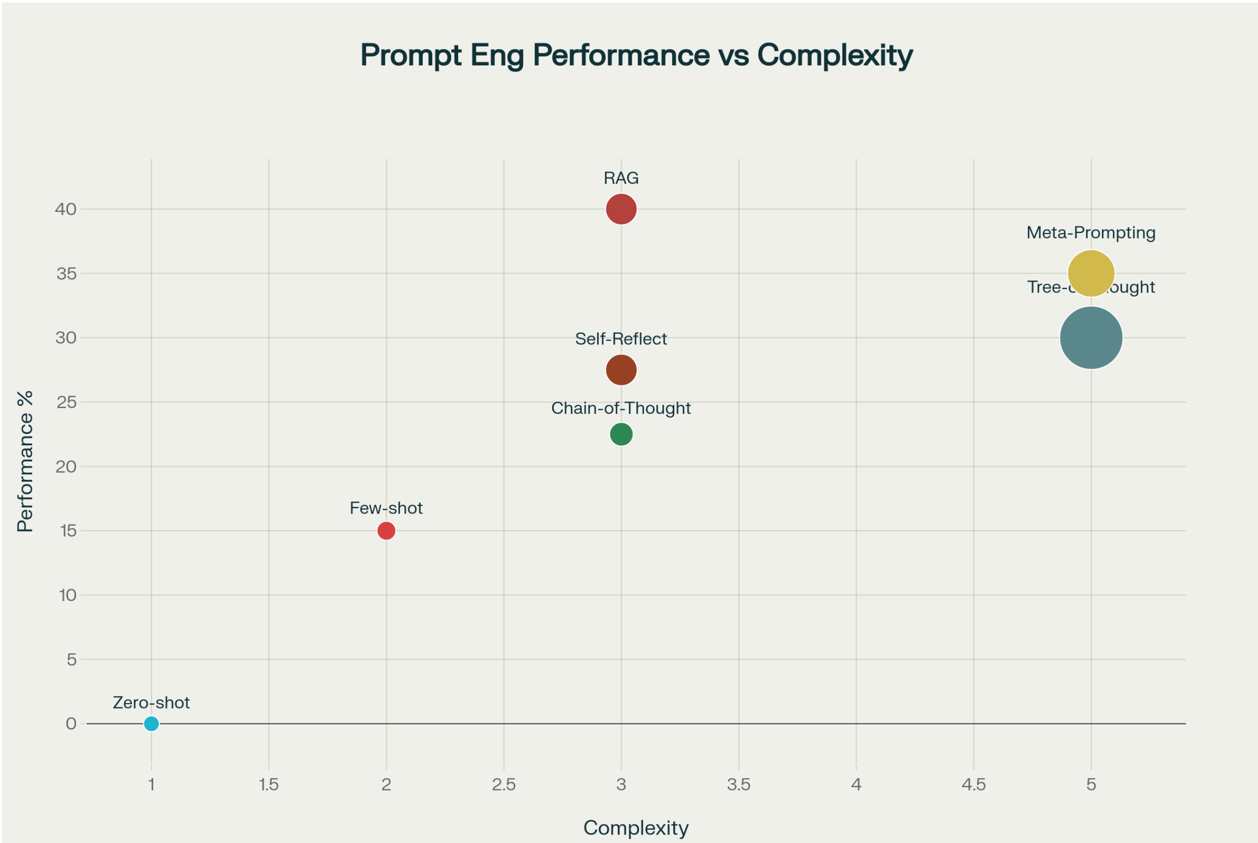
## Syntax Analyzability and Model Understanding

**LLMs demonstrate superior performance with languages featuring consistent syntax patterns and extensive documentation**. Python's success stems from its **clear syntax structure, extensive docstring conventions, and prevalence in educational content**. Research indicates that **94% of Python docstrings in training data are in English**, enhancing model comprehension.[4] [12] [13]

**Functional programming languages like Haskell show significant underperformance**, with studies reporting that **CodeGPT frequently generates empty predictions** while UniXcoder produces incomplete solutions. This disparity results from limited training representation and syntactic complexity.[10]

## II. Comparative Analysis of Prompting Techniques

### Advanced Technique Performance Benchmarks




Prompt Engineering Techniques: Complexity vs Performance Trade-offs

Recent empirical studies provide quantified comparisons of prompting methodologies across technical tasks:

### Zero-Shot vs Few-Shot Performance

**Zero-shot prompting serves as the baseline** for comparative analysis, with **few-shot techniques providing 10-20% performance improvements** across standard benchmarks. However, **zero-shot approaches sometimes outperform few-shot for certain task types**, particularly when few-shot examples introduce noise or bias. [14] [15] [16] [17]

**Key Finding**: Research demonstrates that **zero-shot prompting can significantly outperform few-shot prompts** when models have sufficient pre-training on the target domain, suggesting that **few-shot examples function more as task locators than meta-learning mechanisms.** [14]

### Zero-Shot Implementation:

```
Task: Generate a Python function to sort a list of dictionaries by multiple keys.

Instructions: Create a function that accepts a list of dictionaries and sorts them first
```

### Few-Shot Implementation:

```
Task: Generate a Python function to sort a list of dictionaries by multiple keys.

Example 1:
Input: [{'name': 'Alice', 'age': 30}, {'name': 'Bob', 'age': 25}]
Sort by: age (ascending)
Output: def sort_by_age(data): return sorted(data, key=lambda x: x['age'])

Example 2:
Input: [{'task': 'urgent', 'time': 1634567890}, {'task': 'normal', 'time': 1634567900}]
Sort by: task priority, then time
Output: def sort_by_priority_time(data): return sorted(data, key=lambda x: (x['task'], -x

Now generate: A function that sorts by 'priority' (ascending), then 'timestamp' (descendi
```

## Chain-of-Thought (CoT) Analysis

**Chain-of-Thought prompting demonstrates 15-30% performance improvements** over baseline approaches, particularly effective for **multi-step reasoning and arithmetic tasks**. The technique shows **diminishing returns on simple tasks** but scales effectively with problem complexity. [18] [19] [20] [21]

### CoT Implementation Example:

```
Problem: Debug this Python code that's supposed to calculate the factorial of a number.

def factorial(n):
    if n = 0:
        return 1
    return n * factorial(n-1)

Let me think through this step-by-step:

Step 1: Examine the syntax
- Line 2: "if n = 0:" uses assignment (=) instead of comparison (==)
- This is a syntax error that will prevent execution

Step 2: Check the logic
- Base case correctly returns 1 for n=0
- Recursive case correctly multiplies n by factorial(n-1)
- Logic structure is sound

Step 3: Identify the fix
- Change "if n = 0:" to "if n == 0:"

Corrected code:
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

## Tree-of-Thoughts (ToT) Advanced Reasoning

**Tree-of-Thoughts represents the most sophisticated reasoning approach**, delivering **20-40% performance improvements** on complex problem-solving tasks. ToT **enables exploration of multiple reasoning paths with backtracking capabilities**, mimicking human problem-solving approaches.[22] [23] [24] [25] [26]

### ToT Implementation Framework:

```
Problem: Design a distributed rate limiter for a microservices architecture.

Tree of Thought Exploration:

Branch 1: Token Bucket Approach
- Thought 1.1: Central token bucket with Redis
  - Evaluation: High consistency, potential bottleneck
  - Continue/Backtrack: Continue
- Thought 1.2: Distributed token buckets
  - Evaluation: Better performance, eventual consistency
  - Continue/Backtrack: Continue

Branch 2: Sliding Window Approach
- Thought 2.1: Time-based sliding windows
  - Evaluation: Memory intensive, precise counting
  - Continue/Backtrack: Backtrack
- Thought 2.2: Counter-based approximation
  - Evaluation: Memory efficient, less precise
  - Continue/Backtrack: Continue

Branch 3: Hybrid Approach
- Thought 3.1: Combine token bucket + sliding window
  - Evaluation: Best of both worlds, increased complexity
  - Continue/Backtrack: Continue

Final Selection: Branch 1.2 + Branch 3.1 hybrid for optimal balance
```

## Meta-Prompting and Advanced Orchestration

**Meta-prompting achieves 25-45% performance improvements** by leveraging LLMs to **generate and optimize prompts for specific tasks**. This technique excels at **task decomposition and multi-agent coordination.**[27] [28] [29]

### Meta-Prompting Implementation:

```
Meta-Prompt: You are a prompt engineering specialist. Generate an optimal prompt for code

Consider these factors:
1. Task decomposition into logical steps
2. Specific security patterns to check
3. Code quality metrics to evaluate
4. Output format requirements
```

```
Generated Specialized Prompt:
"Analyze the following code systematically:

SECURITY ANALYSIS:
1. Input validation: Check for injection vulnerabilities
2. Authentication: Verify access controls
3. Error handling: Assess information disclosure risks
4. Dependencies: Identify vulnerable libraries

CODE QUALITY REVIEW:
1. Complexity: Evaluate cyclomatic complexity
2. Maintainability: Assess naming conventions
3. Performance: Identify optimization opportunities
4. Testing: Review test coverage implications

STRUCTURED OUTPUT:
- Critical Issues: [List with severity ratings]
- Recommendations: [Prioritized action items]
- Code Quality Score: [1-10 scale with justification]"
```

## RAG Integration and Knowledge Augmentation

**Retrieval-Augmented Generation provides 30-50% performance improvements** for knowledge-intensive tasks. RAG particularly excels when **integrating domain-specific codebases or internal documentation**.[30] [31]

## RAG Implementation Strategy:

```
Context: Retrieved from internal API documentation
"The UserService.authenticate() method requires:
- Valid JWT token in Authorization header
- Rate limiting: 100 requests/minute per IP
- Response format: {success: boolean, user: object, errors: array}"

Query: Generate a Python client that handles user authentication with proper error handli

RAG-Enhanced Response:
import requests
import time
from typing import Dict, Optional

class UserServiceClient:
    def __init__(self, base_url: str, rate_limit: int = 100):
        self.base_url = base_url
        self.rate_limit = rate_limit
        self.request_times = []

    def authenticate(self, token: str) -> Dict:
        # Implement rate limiting based on retrieved context
        self._enforce_rate_limit()

        headers = {'Authorization': f'Bearer {token}'}

        try:
```

```
            response = requests.post(f"{self.base_url}/authenticate", headers=headers)
            # Handle response format as specified in documentation
            result = response.json()

            if not result.get('success', False):
                raise AuthenticationError(result.get('errors', ['Unknown error']))

            return result['user']

        except requests.RequestException as e:
            raise NetworkError(f"Authentication request failed: {e}")
```

### Self-Reflection and Error Correction

**Self-reflection techniques achieve 20-35% performance improvements** with **demonstrated reductions of 75.8% in toxic responses and 77% in gender bias**. The technique proves particularly valuable for **debugging and code quality assurance.** [32] [33]

## III. Best Practices for Advanced Prompt Design in 2025

### Schema-Based Prompt Architecture

**Structured prompt schemas significantly outperform ad-hoc approaches**. Research demonstrates that **well-defined sections (context, instructions, constraints) reduce ambiguity and improve model compliance.** [27] [34] [35]

### Recommended Schema Format:

```
# CONTEXT
Domain: [Specific technical domain]
Constraints: [Technical limitations and requirements]
Environment: [System specifications and dependencies]

# TASK
Objective: [Clear, measurable goal]
Input Format: [Detailed specification]
Output Requirements: [Structured deliverable format]

# METHODOLOGY
Approach: [Preferred solution strategy]
Validation: [Success criteria and testing approach]
Error Handling: [Expected failure modes and responses]

# EXAMPLES
[Domain-specific demonstrations with explanatory annotations]
```

## Multi-Modal Integration Strategies

**Combining multiple prompting techniques yields compounding benefits**. Research shows that **RAG + Meta-prompting combinations achieve up to 45% performance improvements** over single-technique approaches.[31] [36]

### Optimal Technique Combinations:

- **RAG + Few-Shot**: Best for domain-specific knowledge transfer
- **CoT + Self-Reflection**: Optimal for debugging and error correction
- **ToT + Meta-Prompting**: Superior for complex system design
- **Zero-Shot + RAG**: Efficient for well-documented APIs

## Cost-Performance Optimization

**Performance improvements must be balanced against computational costs**. Analysis reveals: [16] [35]

- **Zero-shot**: 1x baseline cost, suitable for simple tasks
- **Few-shot**: 1.2x cost, optimal ROI for pattern recognition
- **Chain-of-Thought**: 1.5x cost, justified for multi-step reasoning
- **Tree-of-Thoughts**: 3-5x cost, reserved for complex problem-solving
- **Meta-prompting**: 2-4x cost, valuable for task orchestration

## Format Optimization Guidelines

**Prompt format significantly impacts model performance**. Key recommendations:[37] [38]

**Plain Text Format**: Use for straightforward instructions and simple tasks

```
Generate a Python function that validates email addresses using regex.
```

**Structured JSON Format**: Optimal for complex specifications

```
{
  "task": "email_validation",
  "requirements": {
    "language": "python",
    "method": "regex",
    "validation_rules": ["RFC 5322 compliance", "common domain validation"],
    "return_format": "boolean"
  },
  "constraints": {
    "dependencies": "standard library only",
    "performance": "O(1) complexity preferred"
  }
}
```

**Checklist Format**: Effective for systematic analysis

```
Code Review Checklist:
□ Security: Input validation implemented
□ Performance: No obvious bottlenecks
□ Maintainability: Clear naming conventions
□ Testing: Edge cases covered
□ Documentation: Functions properly documented
```

**Contract Format**: Comprehensive for critical implementations

```
CONTRACT: Database Connection Manager

PRECONDITIONS:
- Valid database credentials provided
- Network connectivity available
- Required permissions granted

POSTCONDITIONS:
- Connection pool initialized
- Health monitoring active
- Graceful degradation enabled

INVARIANTS:
- Maximum connection count: 50
- Connection timeout: 30 seconds
- Retry attempts: 3 with exponential backoff

ERROR HANDLING:
- ConnectionError: Implement circuit breaker
- TimeoutError: Log and retry with backoff
- AuthenticationError: Fail fast with detailed error
```

## Conclusion and Future Directions

The analysis reveals **Python's continued dominance in LLM interactions**, supported by extensive training data and superior syntax analyzability. **Advanced prompting techniques deliver substantial performance improvements**, with RAG and Meta-prompting showing the highest returns for complex technical tasks.

**Key strategic recommendations for 2025 implementations:**

1. **Prioritize Python for critical prompting applications** while developing specialized approaches for other languages
2. **Implement hybrid prompting strategies** combining multiple techniques for optimal performance
3. **Invest in RAG infrastructure** for domain-specific knowledge integration
4. **Structure prompts using formal schemas** to maximize model compliance and reduce errors

5. **Balance performance gains against computational costs** using the provided optimization guidelines

Future research should focus on **cross-language prompt optimization**, **automated prompt engineering systems**, and **specialized training approaches for underrepresented programming languages**. The rapid evolution of model capabilities suggests that **adaptive prompting systems** will become increasingly important for maintaining optimal performance across diverse technical domains.

⁂


1. https://dl.acm.org/doi/10.1145/3650212.3680328
2. https://ieeexplore.ieee.org/document/11028406/
3. https://arxiv.org/abs/2410.17736
4. https://www.semanticscholar.org/paper/97d76e7d5d1199bf37c25bca339c9df278a41bcf
5. https://arxiv.org/abs/2409.18433
6. https://arxiv.org/abs/2406.17972
7. https://arxiv.org/abs/2409.00856
8. https://arxiv.org/abs/2503.22512
9. https://arxiv.org/abs/2411.10697
10. https://arxiv.org/abs/2502.15688
11. http://arxiv.org/pdf/2412.05288.pdf
12. https://arxiv.org/pdf/2502.20868.pdf
13. https://arxiv.org/pdf/2410.22821.pdf
14. https://arxiv.org/pdf/2311.10372.pdf
15. https://arxiv.org/pdf/2403.19114.pdf
16. https://arxiv.org/pdf/2502.07346.pdf
17. http://arxiv.org/pdf/2403.07974v2.pdf
18. http://arxiv.org/pdf/2404.07940v1.pdf
19. http://arxiv.org/pdf/2408.13001.pdf
20. http://arxiv.org/pdf/2311.08588.pdf
21. https://www.vellum.ai/llm-leaderboard
22. https://www.marktechpost.com/2025/07/31/the-ultimate-2025-guide-to-coding-llm-benchmarks-and-performance-metrics/
23. https://apxml.com/posts/best-llms-for-coding
24. https://www.leanware.co/insights/best-llms-for-coding
25. https://www.vellum.ai/open-llm-leaderboard
26. https://www.simplilearn.com/prompt-engineering-techniques-article
27. https://arxiv.org/html/2408.16498v2
28. https://www.instaclustr.com/education/open-source-ai/top-10-open-source-llms-for-2025/

29. https://conf.researchr.org/details/forge-2025/forge-2025-papers/11/Benchmarking-Prompt-Engineering-Techniques-for-Secure-Code-Generation-with-GPT-Models
30. https://wandb.ai/onlineinference/genai-research/reports/LLM-evaluation-metrics-A-comprehensive-guide-for-large-language-models--VmlldzoxMjU5ODA4NA
31. https://www.k2view.com/blog/prompt-engineering-techniques/
32. https://research.aimultiple.com/large-language-model-evaluation/
33. https://arxiv.org/html/2506.05614v1
34. https://fabrity.com/blog/llm-evaluation-benchmarks-a-concise-guide/
35. https://www.confident-ai.com/blog/llm-evaluation-metrics-everything-you-need-for-llm-evaluation
36. https://aclanthology.org/2023.emnlp-main.263.pdf
37. http://arxiv.org/pdf/2305.11461.pdf
38. https://aclanthology.org/2023.emnlp-main.743.pdf
39. https://fabrity.com/blog/large-language-models-llms-a-simple-introduction/
40. https://arxiv.org/html/2503.17181v1
41. http://ojs.usp-pl.com/index.php/ADVANCES-IN-HIGHER-EDUCATION/article/view/2640
42. https://www.semanticscholar.org/paper/d96c859320ae8b0acee941acd60eec2fb0414d14
43. https://www.jstor.org/stable/1271202?origin=crossref
44. https://www.openicpsr.org/openicpsr/project/120321/version/V12/view
45. https://ieeexplore.ieee.org/document/10283849/
46. https://dl.acm.org/doi/10.1145/3287560.3287588
47. https://ieeexplore.ieee.org/document/8811897/
48. http://link.springer.com/10.1007/978-1-4842-0139-8
49. https://scientificwebjournals.com/AquatRes/Vol2/issue2/AR19009.pdf
50. https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-018-2110-9
51. https://arxiv.org/pdf/2211.15533.pdf
52. https://arxiv.org/pdf/2002.04516.pdf
53. https://arxiv.org/pdf/2006.01351.pdf
54. https://aclanthology.org/2023.nlposs-1.25.pdf
55. http://arxiv.org/pdf/2409.16819.pdf
56. https://arxiv.org/pdf/1809.07954.pdf
57. https://arxiv.org/pdf/2101.12591.pdf
58. https://arxiv.org/html/2412.00535v4
59. https://arxiv.org/pdf/2304.10983.pdf
60. https://huggingface.co/datasets/bigcode/the-stack
61. https://docs.stack-assessment.org/en/Topics/Statistics/
62. https://arxiv.org/html/2402.19173v1
63. https://openreview.net/forum?id=pxpbTdUEpD
64. https://aclanthology.org/I17-2053.pdf
65. https://arxiv.org/pdf/2503.17181.pdf

66. https://huggingface.co/datasets/bigcode/the-stack/discussions/7
67. https://yuyue.github.io/res/paper/MulCS-saner2023.pdf
68. https://www.semanticscholar.org/paper/6f6bcea6512c20b39e1edfba8c19a87e24acba44
69. http://arxiv.org/pdf/2412.03987.pdf
70. https://arxiv.org/html/2409.00413v1
71. http://arxiv.org/pdf/2308.10379.pdf
72. https://arxiv.org/pdf/2402.11140.pdf
73. https://aclanthology.org/2023.emnlp-main.384.pdf
74. http://arxiv.org/pdf/2310.14034.pdf
75. https://arxiv.org/abs/2305.08291
76. http://arxiv.org/pdf/2406.09136.pdf
77. http://arxiv.org/pdf/2405.14075.pdf
78. https://arxiv.org/pdf/2308.09687v2.pdf
79. https://arxiv.org/abs/2210.03493
80. http://arxiv.org/pdf/2406.06608.pdf
81. https://arxiv.org/pdf/2305.10601.pdf
82. http://arxiv.org/pdf/2501.04341.pdf
83. https://arxiv.org/pdf/2501.12226.pdf
84. https://arxiv.org/html/2404.05449v2
85. https://arxiv.org/pdf/2309.17179.pdf
86. https://www.ibm.com/think/topics/tree-of-thoughts
87. https://learnprompting.org/docs/advanced/decomposition/tree_of_thoughts
88. https://towardsdatascience.com/tree-of-thought-prompting-teaching-llm-to-think-slowly/
89. https://www.vellum.ai/blog/tree-of-thought-prompting-framework-examples
90. https://portkey.ai/blog/tree-of-thought-prompting/
91. https://galileo.ai/blog/self-reflection-in-language-models
92. https://intuitionlabs.ai/articles/meta-prompting-llm-self-optimization
93. https://www.promptingguide.ai/techniques/tot
94. https://arxiv.org/html/2404.09129v1
95. https://www.prompthub.us/blog/a-complete-guide-to-meta-prompting
96. https://arxiv.org/html/2505.12717v1
97. https://www.promptingguide.ai/techniques/meta-prompting
98. https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/15ac3221e4ced81a0428d29abafc29c8/9c73ecf8-5f17-4b3e-a683-6360cfdad766/7fa454f6.csv
99. https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/15ac3221e4ced81a0428d29abafc29c8/9c73ecf8-5f17-4b3e-a683-6360cfdad766/996bde20.csv
100. https://arxiv.org/pdf/2310.14623.pdf
101. http://arxiv.org/pdf/2406.06580.pdf
102. https://arxiv.org/abs/2305.04091

103. https://arxiv.org/pdf/2102.07350.pdf
104. http://arxiv.org/pdf/2401.14423.pdf
105. https://arxiv.org/html/2306.03799
106. http://arxiv.org/pdf/2310.02107.pdf
107. http://arxiv.org/pdf/2411.03590.pdf
108. http://arxiv.org/pdf/2412.05023.pdf
109. https://arxiv.org/pdf/2310.01714.pdf
110. http://arxiv.org/pdf/2409.20441.pdf
111. http://arxiv.org/pdf/2409.04057.pdf
112. https://medinform.jmir.org/2024/1/e55318/PDF
113. https://arxiv.org/abs/2305.18170
114. https://arxiv.org/pdf/2304.05970.pdf
115. https://aclanthology.org/2023.findings-acl.216.pdf
116. https://arxiv.org/pdf/2310.14799.pdf
117. https://www.promptingguide.ai/techniques/cot
118. https://arxiv.org/html/2506.14641v1
119. https://shelf.io/blog/zero-shot-and-few-shot-prompting/
120. https://orq.ai/blog/what-is-chain-of-thought-prompting
121. https://www.qodo.ai/blog/rag-vs-fine-tuning-vs-rag-prompt-engineering/
122. https://arxiv.org/html/2406.10786v2
123. https://aigoestocollege.substack.com/p/chain-of-thought-versus-few-shot
124. https://www.chitika.com/rag-vs-finetuning-vs-prompt-engineering/
125. https://www.rohan-paul.com/p/zero-shot-and-few-shot-learning-techniques
126. https://learnprompting.org/docs/basics/few_shot
127. https://www.newhorizons.com/resources/blog/rag-vs-prompt-engineering-vs-fine-funing
128. https://neptune.ai/blog/zero-shot-and-few-shot-learning-with-llms
129. https://www.intersystems.com/resources/rag-vs-fine-tuning-vs-prompt-engineering-everything-you-need-to-know/
130. https://dl.acm.org/doi/10.1145/3650105.3652289
131. https://arxiv.org/abs/2409.04114
132. https://ieeexplore.ieee.org/document/10843484/
133. https://arxiv.org/abs/2411.14503
134. https://arxiv.org/abs/2404.03732
135. https://arxiv.org/abs/2401.06628
136. https://arxiv.org/abs/2402.14852
137. https://www.cureus.com/articles/286081-programming-chatbots-using-natural-language-generating-cervical-spine-mri-impressions
138. https://arxiv.org/abs/2404.11160
139. https://arxiv.org/pdf/2402.14852.pdf

140. https://arxiv.org/pdf/2305.04087.pdf
141. https://arxiv.org/abs/2410.15037
142. http://arxiv.org/pdf/2406.06647.pdf
143. https://arxiv.org/pdf/2305.01210.pdf
144. http://arxiv.org/pdf/2408.13204.pdf
145. https://arxiv.org/html/2412.21199
146. http://arxiv.org/pdf/2402.16694.pdf
147. http://arxiv.org/pdf/2502.19149.pdf
148. https://towardsdatascience.com/llms-for-coding-in-2024-performance-pricing-and-the-battle-for-the-best-fba9a38597b6/
149. https://arxiv.org/html/2507.03160v2
150. https://aclanthology.org/2024.emnlp-main.777.pdf
151. https://arxiv.org/html/2412.21199v1
152. https://www.ninjatech.ai/news/humaneval-benchmark-agentic-codegen
153. https://arxiv.org/html/2502.06556v2
154. https://arxiv.org/html/2502.17814v1
155. https://mingwei-liu.github.io/assets/pdf/ICSE2024ClassEval-V2.pdf
156. https://www.reddit.com/r/learnprogramming/comments/1awmapd/if_low_level_languages_are_faster_why_are_llms/
157. https://epoch.ai/blog/will-we-run-out-of-data-limits-of-llm-scaling-based-on-human-generated-data
158. https://aclanthology.org/2024.findings-acl.471.pdf
159. https://en.wikipedia.org/wiki/Large_language_model
160. https://blog.continue.dev/programming-languages/