Advanced Prompt Engineering for Technical LLM Tasks: A 2025 Research Analysis

Executive Summary

The landscape of Large Language Models (LLMs) for technical tasks has undergone a paradigm shift, evolving from simple, single-turn code completion to complex, reasoning-driven, and context-aware software engineering. State-of-the-art performance in 2025 is no longer achieved by a single model or technique but through the strategic hybridization of advanced prompting methodologies, the adoption of structured data formats for reliability, and a nuanced, evidence-based approach to language-specific model capabilities that extends far beyond Python. This report provides a deep-dive analysis into the optimal strategies for leveraging LLMs in technical domains, grounded in the latest 2024–2025 research and benchmark data.

The core findings indicate that while Python remains a high-performing language due to its prevalence in training corpora, new multilingual benchmarks reveal a more complex hierarchy of model competencies. The most significant performance gains now stem from techniques that elicit structured reasoning, such as Structured Chain-of-Thought (SCoT) and Chain of Code (CoC), which demonstrably outperform simpler methods. Furthermore, for tasks requiring external or proprietary knowledge, Retrieval-Augmented Generation (RAG) is indispensable, and combining it with reasoning frameworks like Chain-of-Thought (CoT) yields substantial improvements in accuracy and factual grounding.

Key Recommendations:

- **Adopt a "Complexity Ladder" Approach to Prompting:** Begin with simple Few-shot prompts for clarity and format control. Escalate to Chain-of-Thought for logical complexity, Tree-of-Thoughts for tasks requiring exploration, and Retrieval-Augmented Generation (RAG) for knowledge-dependent tasks.
- **Prioritize Structured Formats for Production Systems:** For any automated workflow, utilize JSON or schema-based prompts to ensure reliable, parsable outputs. Employ checklist-driven prompts for systematic evaluation tasks like code reviews and security

audits.
- **Evaluate Models Against Modern, Multilingual Benchmarks:** Move beyond saturated, Python-centric benchmarks like HumanEval. Assess model performance for your specific tech stack using contamination-resistant and multilingual benchmarks such as LiveCodeBench and AutoCodeBench to gain a true understanding of a model's capabilities.
- **Embrace Hybrid, Agentic Architectures:** The most powerful applications combine methodologies (e.g., RAG + CoT). Treat the LLM not as a monolithic tool but as a reasoning engine within a larger computational system that can retrieve data, generate solutions, and self-correct.

The future trajectory points towards increasingly sophisticated agentic workflows, where LLMs manage entire development cycles, and multimodal models that can interpret visual inputs like UI mockups and architectural diagrams. The role of the prompt engineer is thus evolving into that of an AI systems architect, responsible for designing these complex, hybrid systems.

I. Language Selection for Code Prompts: An Evidence-Based Analysis

The choice of programming language significantly impacts LLM performance. This influence stems from the composition of pre-training datasets, the focus of evaluation benchmarks, and the inherent characteristics of the languages themselves. A data-driven analysis reveals a nuanced landscape where historical Python dominance is being challenged by a more granular understanding of multilingual capabilities.

1.1 The Foundational Role of Pre-training Corpora

The capabilities of code-generating LLMs are fundamentally shaped by the data they are trained on. The largest and most influential of these datasets is "The Stack," which serves as a primary training corpus for leading models like StarCoder2.[1]

- **Dataset Composition and Scale:** The Stack v1.1 contains over 6 TB of permissively-licensed source code spanning 358 programming languages, while v2 expands this to over 600 languages.[2] This immense scale provides models with a broad syntactic and semantic understanding across a wide array of programming paradigms.

- **Inherent Language Bias:** Despite its breadth, the dataset is heavily skewed towards languages with large, open-source ecosystems. Languages like Python, JavaScript, Java, C++, and HTML are massively overrepresented, while languages common in closed-source enterprise environments (e.g., C#) or those with less permissive licensing are less prevalent.[2] This distribution directly correlates to a model's baseline fluency and ability to generate idiomatic code. A model trained on this data will naturally exhibit higher performance on a Python task than on a comparable task in Scala or Julia, simply due to the disparity in training exposure.

1.2 Performance on Foundational (Python-Centric) Benchmarks

The first generation of influential code benchmarks, HumanEval and Mostly Basic Programming Problems (MBPP), were instrumental in measuring progress but are now showing their limitations.[7]

- **Python Dominance and Benchmark Saturation:** These benchmarks consist almost exclusively of Python function-synthesis problems.[9] State-of-the-art models from OpenAI (o-series), Google (Gemini 2.5 Pro), and Anthropic (Claude 3.7) now achieve Pass@1 (percentage of problems solved on the first attempt) scores exceeding 95% on these benchmarks, with some approaching perfect scores.[9] This indicates that for simple, self-contained Python function generation, the problem is largely solved by top-tier models.
- **The Risk of Overfitting:** The static nature of these benchmarks has led to concerns about contamination, where solutions are inadvertently included in training sets, and overfitting. Analysis from the dynamic LiveCodeBench benchmark reveals that models heavily fine-tuned on HumanEval-style problems often perform significantly worse on novel, unseen problems of similar difficulty.[12] This suggests that high scores on older benchmarks may reflect memorization rather than true reasoning ability, making them less reliable indicators of real-world performance.[13]

1.3 The New Frontier: Multilingual and High-Difficulty Benchmarks

To address the limitations of earlier benchmarks, 2024–2025 saw the emergence of more rigorous, multilingual evaluation suites that provide a more accurate picture of model capabilities.

- **AutoCodeBench:** This benchmark is designed to be a significant challenge, featuring

3,920 difficult problems evenly distributed across 20 programming languages, with over 60% of problems rated as "hard".[14] The results are sobering: even the top-performing model, Claude Opus 4, achieves an average Pass@1 of only 52.4%.[14] This benchmark effectively equalizes the playing field, revealing that the perceived "premium" for Python performance was partly an artifact of biased evaluation.

- **HumanEval-XL and CodeIF:** Other benchmarks expand the evaluation landscape. HumanEval-XL translates the original HumanEval problems into 12 programming languages and 23 natural languages, testing cross-lingual generalization.[16] CodeIF focuses on instruction-following capabilities in Java, Python, Go, and C++, providing a more nuanced view of performance in statically-typed languages.[18]
- **Language-Specific Strengths:** Data from these new benchmarks reveals distinct performance tiers. A study of Small Language Models (SLMs) showed stronger performance in Python, Java, and PHP, with weaker results in Go, C++, and Ruby.[19] AutoCodeBench data further clarifies this, showing that while top models perform similarly on popular languages, the performance gap widens for lower-resource languages like Racket and Elixir, where certain models like Claude Opus 4 demonstrate a notable advantage.[14] This underscores that performance is not just about learning syntax but also about learning the diverse patterns of problem-solving within a language's ecosystem. A language's strength in training data is therefore a function of both token volume and the variety of solved problems available in its public corpora.

Language	Claude Opus 4 (20250514) Pass@1	o3-high (20250416) Pass@1	Current Upper Bound Pass@1
High-Resource			
C#	74.9%	68.3%	88.4%
Java	55.9%	53.2%	78.7%
Python	40.3%	40.8%	63.3%
C++	44.1%	47.3%	74.7%
JavaScript	38.6%	40.8%	59.2%

Mid-Resource				
Go	37.2%	22.0%	69.1%	
Rust	38.7%	N/A	61.3%	
Swift	50.0%	N/A	78.0%	
TypeScript	47.2%	N/A	61.3%	
Kotlin	72.5%	72.0%	89.5%	
Low-Resource				
Elixir	80.3%	80.8%	97.5%	
Racket	68.9%	53.1%	88.3%	
Perl	44.5%	44.0%	64.5%	
Ruby	61.0%	59.0%	79.5%	
PHP	28.1%	32.7%	52.8%	
Average	**52.4%**	**51.1%**	**74.8%**	
Table 1: Comparative Ranking of Programming Languages on the AutoCodeBench Benchmark (Pass@1, 2025). Data synthesized				

from.[14] "Current Upper Bound" represents the aggregate best score achieved by any model on that language.				

1.4 Trade-offs: Language Verbosity and Syntactic Complexity

The intrinsic properties of a programming language also influence LLM performance.

- **Verbose vs. Concise Languages:** Languages like Java are considered more verbose, requiring more code to express functionality compared to a concise language like Python.[20] This has a direct impact on cost and error rates. Generating verbose code consumes more tokens, increasing API costs, and provides more opportunities for the model to make syntactic errors.[21]
- **Syntactic Complexity and Type Systems:** Languages with complex syntax, strict type systems, and advanced features like ownership and borrowing in Rust can pose a greater challenge for LLMs. While models can learn these rules, the probability of generating syntactically incorrect code is higher compared to dynamically-typed languages like JavaScript, particularly for less advanced models or highly complex prompts.[22]

II. Advanced Prompting Methodologies for Technical Tasks

The evolution of prompt engineering reflects a progression from simple instruction-following to complex, multi-step reasoning and computation. Each advanced technique was developed to overcome specific failure modes of its predecessors, forming a "complexity ladder" that provides a diagnostic framework for prompt design.

2.1 Baseline Techniques: Zero-shot and Few-shot Prompting

- **Zero-shot Prompting:** This is the most basic form of interaction, where the model is given a direct instruction without any examples.[24] It relies entirely on the model's pre-trained knowledge.
  - **Best Use-Case:** Simple, unambiguous tasks where the desired format is common, such as writing a standard Python function or translating a simple code snippet.
  - **Limitations:** Prone to failure when the task is nuanced, requires a specific output format, or involves complex logic the model might misinterpret.[26]
- **Few-shot Prompting (In-Context Learning):** This technique involves providing 2-5 examples of input/output pairs within the prompt before the final query.[27] These examples act as a form of "in-context training," guiding the model on the expected format, style, and logic without requiring fine-tuning.
  - **Best Use-Case:** Any task requiring a specific output structure (e.g., JSON), adherence to a particular coding style, or clarification of an ambiguous request.
  - **Quantified Improvement:** Providing examples significantly boosts performance. A zero-shot prompt might produce a function without proper error handling, whereas a few-shot prompt with examples that include input validation will yield more robust code.[27] In a code translation task, using RAG to find relevant few-shot examples increased the Pass@1 score from 48.3% to 67.7%.[29]

2.2 Reasoning-Based Techniques: Eliciting Deeper Logic

When a task requires multiple logical steps, simply providing examples is insufficient. Reasoning-based techniques guide the model to break down the problem first.

- **Chain-of-Thought (CoT):** This method prompts the model to generate a series of intermediate reasoning steps before arriving at the final answer, often triggered by a simple phrase like "Let's think step-by-step".[30]
  - **Best Use-Case:** Problems involving multi-step calculations, complex logic, or algorithmic design where the path to the solution is as important as the solution itself.
  - **Structured-CoT (SCoT):** An enhancement where the reasoning steps are framed using programming constructs (e.g., outlining classes, functions, and control flow). This forces the model to "think" in a way that maps directly to code. SCoT has been shown to outperform standard CoT by up to 13.79% in Pass@1 on HumanEval.[32]
- **Chain of Code (CoC):** This technique reframes the "thought" process as an executable program. The model generates code or pseudocode that represents its reasoning, which

can then be validated by an interpreter.[33] This moves prompting from the realm of ambiguous natural language into verifiable computation.
    - **Best Use-Case:** Algorithmic and logical reasoning tasks where correctness can be formally verified.
    - **Quantified Improvement:** CoC achieves an 84% score on the BIG-Bench Hard benchmark, a 12-point absolute improvement over standard CoT, demonstrating its superior capability for complex, verifiable reasoning.[33]
- **Tree-of-Thoughts (ToT):** ToT is a framework for deliberate problem-solving. It prompts the model to explore multiple parallel reasoning paths (branches of a tree), self-evaluate the promise of each path, and backtrack from dead ends.[34]
    - **Best Use-Case:** Complex problems with a large search space or no obvious linear solution path, such as optimization challenges or system design questions.
    - **Quantified Improvement:** ToT has shown dramatic improvements in general reasoning, boosting success rates on some tasks from 49% (CoT) to 74%.[35] In coding, specialized ToT variants like RethinkMCTS have elevated GPT-3.5-turbo's Pass@1 on HumanEval from 70.12% to 89.02%.[36]

2.3 Knowledge-Intensive and Automated Techniques

- **Retrieval-Augmented Generation (RAG):** RAG addresses the LLM's inherent knowledge limitations by connecting it to external data sources. The process involves retrieving relevant documents (e.g., API documentation, internal codebase snippets, best-practice guides) and injecting them into the prompt's context.[37]
    - **Best Use-Case:** Generating code that uses private libraries, adheres to project-specific coding standards, or requires knowledge of information created after the model's training cutoff date.
    - **Quantified Improvement:** RAG is highly effective. Studies using a Programming Knowledge Graph for retrieval improved Pass@1 accuracy by up to 20% on HumanEval and 34% on MBPP compared to non-RAG baselines.[39]
- **Meta-Prompting:** This advanced technique uses a powerful LLM (e.g., GPT-5, Claude 4) to act as a "prompt engineer," generating or refining prompts for a target LLM.[41] This automates the optimization process.
    - **Best Use-Case:** Systematically improving prompt performance for a recurring, high-value task, such as code optimization or documentation generation.
    - **Quantified Improvement:** The Meta-Prompted Code Optimization (MPCO) framework, which uses meta-prompting to generate context-aware prompts for code optimization, achieved runtime performance improvements of up to 19.06% on real-world codebases.[43]

Prompting Methodology	Baseline Pass@1 (GPT-4o est.)	Technique Pass@1	Absolute Improvement	
Zero-shot (Baseline)	~87.0%	87.0%	0.0%	
Few-shot	~87.0%	~91.0%	+4.0%	
Chain-of-Thought (CoT)	~87.0%	~92.5%	+5.5%	
Structured-CoT (SCoT)	~87.0%	~94.2%	+7.2%	
Tree-of-Thoughts (ToT Variant)	~87.0%	~94.5%	+7.5%	
RAG	~87.0%	~95.0%	+8.0%	
Table 2: Estimated Performance Gains of Prompting Methodologies on HumanEval. Baseline is estimated for a SOTA model like GPT-4o. Improvements are synthesized from multiple studies [29] to show relative				

impact. Absolute values vary by model and implementation.				

III. Strategic Prompt Formatting for Code Generation and Analysis

The structure of a prompt is as critical as its content. The choice of format is a fundamental engineering decision that dictates the reliability, predictability, and programmatic usability of an LLM's output. Moving from plain text to structured formats represents a shift from low-control, high-variability outputs to high-control, low-variability outputs essential for production systems.

3.1 Plain Text: The Baseline for Simplicity

Plain text prompts are the most straightforward way to interact with an LLM. However, their effectiveness hinges on clear structure.

- **Best Practices:**
  - **Instruction First:** Place the primary instruction at the beginning of the prompt.[45]
  - **Use Delimiters:** Separate instructions from context using clear markers like ### or triple quotes (""") to help the model parse the request.[46]
  - **Be Specific:** Articulate the desired outcome, length, format, and style in detail. Instead of "make this shorter," use "summarize this in three bullet points".[45]
  - **Positive Framing:** Instruct the model on what to do, rather than what to avoid. For example, "Refer the user to the FAQ for PII questions" is better than "DO NOT ASK FOR PII".[45]
- **When to Use:** Ideal for interactive, human-in-the-loop scenarios such as brainstorming, initial code drafting, or generating explanations where the output does not need to be programmatically parsed.

3.2 Structured Schemas (JSON): The Standard for Production Systems

For any application where an LLM's output is consumed by another piece of software, structured formats are non-negotiable. JSON is the de facto standard.

- **Why JSON?:** It provides a machine-readable format that eliminates the ambiguity of natural language, ensuring outputs are consistent and can be reliably parsed.[47] This is critical for API calls, data extraction, and feeding information into downstream systems.[49]
- **Schema-Based Prompting:** This technique involves defining the desired output structure using a schema, often a subset of the OpenAPI 3.0 specification.[50] This schema is provided to the model either in the prompt itself or as a parameter in the API call. The model is then constrained to generate a response that conforms to this schema.
- **Performance Impact:** For tasks like generating a webpage or extracting structured data, JSON-based prompts deliver "significantly better results" than plain text. The structure enforces a tighter focus, leading to outputs with superior functionality and stricter adherence to requirements.[47] The format itself acts as a cognitive scaffold, guiding the model's reasoning process toward a more complete and logical result.

3.3 Checklist-Driven Prompts: A Systematic Approach for Complex, Multi-Point Tasks

A checklist prompt provides the LLM with a list of specific criteria to evaluate or address, transforming a vague request into a systematic analysis.

- **Primary Use Case: Code Review:** Checklists are exceptionally effective for automating code reviews. Instead of a generic prompt like "Review this code," a checklist forces the model to assess specific quality dimensions.[53] A comprehensive checklist might include:
  - **Code Quality & Maintainability:** Adherence to SOLID/DRY principles, meaningful naming conventions.[55]
  - **Security:** Checking for common vulnerabilities like SQL injection, XSS, and improper input validation.[54]
  - **Performance:** Identifying potential bottlenecks, inefficient algorithms, or excessive memory usage.[53]
  - **Testing:** Verifying the presence and quality of unit tests, including edge case coverage.[56]
  - **Documentation:** Ensuring code is well-commented and READMEs are updated.[56]
- **Secondary Use Case: Structured Debugging:** A checklist can guide a model through a systematic debugging process: "1. Analyze the stack trace. 2. Verify input data for the

failing function. 3. Check for common off-by-one errors. 4. Suggest logging statements to isolate the issue.".[57]

Task	Plain Text	Structured JSON/Schema	Checklist
Algorithm Generation	Suitable	Optimal	Not Recommended
Rationale	Good for initial drafts, but JSON can enforce structure like function signature and complexity analysis.	Ensures output includes code, explanation, and complexity in a parsable format.	Overly restrictive for creative problem-solving.
API Call Generation	Not Recommended	Optimal	Suitable
Rationale	Prone to formatting errors and missing parameters.	Guarantees syntactically correct and complete API call structures (e.g., function definitions).	Can be used to ensure all required parameters and error handling are included.
Code Refactoring	Suitable	Suitable	Optimal
Rationale	Can provide good suggestions but may lack structure.	Can request a "before" and "after" code block with a JSON object for the explanation.	Best for enforcing specific principles (e.g., SOLID, DRY) or standards.
Security Audit	Not Recommended	Suitable	Optimal
Rationale	Too vague; likely to miss critical	Can structure the report by	The most effective way to ensure a

	vulnerabilities.	vulnerability type (e.g., SQLi, XSS).	systematic and comprehensive review against a known set of vulnerabilities.
Code Explanation	Optimal	Suitable	Not Recommended
Rationale	Natural language is best for generating human-readable explanations.	Can be used to structure the explanation (e.g., by line number, by logical block).	Too rigid for a narrative explanation.
Debugging	Suitable	Suitable	Optimal
Rationale	Good for interactive debugging sessions with a human.	Can be used to structure the analysis of an error log.	Excellent for guiding the LLM through a systematic, step-by-step debugging process.
Table 3: Prompt Format Selection Matrix for Technical LLM Tasks.			

IV. Illustrative Examples: Techniques in Practice

This section provides practical, detailed examples demonstrating the application of the discussed methodologies and formats across various languages and tasks.

4.1 Zero-shot and Few-shot Prompting Examples

Example 1: Zero-shot Function Generation (Python)

A simple, direct request that relies on the model's pre-trained knowledge.

- **Prompt:**
  Write a Python function to calculate the factorial of a number.

- **Expected LLM Output:**
  ```
  Python
  def factorial(n):
      """
      Calculates the factorial of a non-negative integer.
      """
      if n == 0:
          return 1
      else:
          return n * factorial(n-1)
  ```

- **Analysis and Failure Mode:**
  - **Effectiveness:** The model correctly generates a functional recursive solution.
  - **Failure Mode (Semantic Error):** The generated code lacks input validation. It will enter an infinite recursion loop if given a negative number and may fail for non-integer inputs. This is a common failure for simple zero-shot prompts, which often produce functionally correct but not robust code.[22]

Example 2: Few-shot Component Generation (JavaScript/React)

This prompt uses examples to enforce a specific coding style (functional components with hooks) and structure.

- **Prompt:**
  You are an expert React developer. Your task is to create functional components using hooks.

  ### Example 1
  # Task: Create a simple counter button.

```
# Output:
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  return (
    <button onClick={() => setCount(count + 1)}>
      Clicked {count} times
    </button>
  );
};

export default Counter;

### Example 2
# Task: Create an input field that displays its current value.
# Output:
import React, { useState } from 'react';

const InputDisplay = () => {
  const [value, setValue] = useState('');

  return (
    <div>
      <input
        type="text"
        value={value}
        onChange={(e) => setValue(e.target.value)}
        placeholder="Type here..."
      />
      <p>Current value: {value}</p>
    </div>
  );
};

export default InputDisplay;

### Task
# Task: Create a component that fetches and displays a user's name from an API endpoint '/api/user/{id}'. It should show a "Loading..." message while fetching.
```

- **Expected LLM Output:**

```
JavaScript
import React, { useState, useEffect } from 'react';

const UserDisplay = ({ id }) => {
  const [userName, setUserName] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    setLoading(true);
    fetch(`/api/user/${id}`)
     .then(response => response.json())
     .then(data => {
       setUserName(data.name);
       setLoading(false);
     })
     .catch(error => {
       console.error("Error fetching user data:", error);
       setLoading(false);
     });
  }, [id]);

  if (loading) {
    return <div>Loading...</div>;
  }

  return <h1>Hello, {userName}</h1>;
};

export default UserDisplay;
```

- **Analysis and Failure Mode:**
  - **Effectiveness:** The few-shot examples successfully guide the model to produce a functional component using useState and useEffect hooks, which is the modern standard. A zero-shot prompt might have defaulted to an older, class-based component.
  - **Failure Mode (Recency Bias):** LLMs can be biased towards the last example provided.[27] If the last example demonstrated a poor practice, the model might replicate it, even if earlier examples showed better patterns.

4.2 Chain-of-Thought and Tree-of-Thoughts Examples

Example 1: Structured Chain-of-Thought (SCoT) for Business Logic (Java)

This prompt forces the model to plan its code structure before writing it, improving logical correctness for complex tasks.

- **Prompt:**
  You are a senior Java developer. Your task is to implement a shipping cost calculator.
  First, provide a structured plan for your implementation using the following format:
  1. Class and Method Signature: Define the main class and method signature.
  2. Input Validation: Describe how you will handle invalid inputs.
  3. Core Logic: Outline the steps to calculate the cost based on weight and distance.
  4. Edge Cases: List any edge cases you will consider.

  After providing the plan, write the full Java code.

  The rules for shipping are:
  - Base rate is $5.
  - Add $0.50 for every kg over 1 kg.
  - Add $0.10 for every mile over 5 miles.
  - If weight is over 50 kg, add a $20 heavy item surcharge.

- **Expected LLM Output:**
  Here is the structured plan for the shipping cost calculator:

  1. **Class and Method Signature:**
     - Create a public class `ShippingCalculator`.
     - Define a public static method `calculateCost(double weightInKg, double distanceInMiles)`.

  2. **Input Validation:**
     - Check if `weightInKg` and `distanceInMiles` are non-negative. If not, throw an `IllegalArgumentException`.

  3. **Core Logic:**
     - Initialize `cost` to the base rate of $5.00.
     - If `weightInKg` > 1, calculate the additional weight cost: `(weightInKg - 1) * 0.50`.
     - If `distanceInMiles` > 5, calculate the additional distance cost: `(distanceInMiles - 5) * 0.10`.

- Add both additional costs to the `cost`.

4. **Edge Cases:**
   - Weight exactly 1 kg (no extra weight cost).
   - Distance exactly 5 miles (no extra distance cost).
   - Heavy item surcharge: if `weightInKg` > 50, add $20.00 to the final cost.

Here is the full Java code:

```java
public class ShippingCalculator {

    public static double calculateCost(double weightInKg, double distanceInMiles) {
        if (weightInKg < 0 |
```

| distanceInMiles < 0) {
throw new IllegalArgumentException("Weight and distance must be non-negative.");
}

```
        double cost = 5.00; // Base rate

        // Calculate additional weight cost
        if (weightInKg > 1) {
            cost += (weightInKg - 1) * 0.50;
        }

        // Calculate additional distance cost
        if (distanceInMiles > 5) {
            cost += (distanceInMiles - 5) * 0.10;
        }

        // Apply heavy item surcharge
        if (weightInKg > 50) {
            cost += 20.00;
        }
```

```
    return cost;
  }
}
```
```

- **Analysis and Failure Mode:**
  - **Effectiveness:** The SCoT prompt forces a deliberate, structured approach. By planning first, the model is less likely to miss requirements, such as the heavy item surcharge. This method improves Pass@1 rates significantly over standard CoT.[32]
  - **Failure Mode (Erroneous Reasoning):** The primary failure mode for any CoT technique is an error in the reasoning chain. If the model makes a logical mistake in its plan (e.g., misinterpreting a rule), that error will be propagated directly into the final code.[58]

### Example 2: Tree-of-Thoughts for Algorithmic Puzzles (Python)

This prompt simulates a ToT process by asking for multiple "experts" to propose, critique, and refine solutions, mimicking the exploration of a thought tree.

- **Prompt:**
  Three expert programmers are collaborating to solve the "Game of 24" using Python.
  Given four numbers, they must find a mathematical expression that equals 24.
  They will work in steps. In each step, each expert will propose one move. Then, they will evaluate the moves and decide on the best path forward. If a path is a dead end, they will state it and backtrack.

  The numbers are: 8, 8, 3, 3.

  Let's begin.

  Step 1:
  - Expert 1 proposes:
  - Expert 2 proposes:
  - Expert 3 proposes:

  Evaluation:
  - [LLM fills in evaluation of proposals]

  Step 2:
```

- [LLM continues the process]

- **Expected LLM Output:**
  Step 1:
  - Expert 1 proposes: Let's try multiplication first. `8 * 3 = 24`. This looks promising.
  - Expert 2 proposes: Let's try division. `8 / 3` is not an integer, this might get complicated. Let's try `8 / (3 - 8/3)`.
  - Expert 3 proposes: Let's try addition. `8 + 8 = 16`. We are left with 3, 3. `16 + 3 + 3 = 22`. Close, but not 24.

  Evaluation:
  - Expert 1's path is the most direct. If we use `8 * 3`, we get 24. We are left with numbers 8 and 3. We need to incorporate them while keeping the result 24. Maybe `(8 * 3) * (8 - 3 -...)` is not the way. Let's re-examine. `8 * 3 = 24`. We have used one 8 and one 3. We are left with another 8 and 3. We can't just ignore them. This path is a dead end.
  - Expert 2's path is overly complex and relies on fractions. Let's backtrack.
  - Expert 3's path led to 22. Not a solution.

  Let's reconsider. What if we don't aim for 24 immediately?

  Step 2:
  - Expert 1 proposes: Let's try division differently. `8 / (3 - 8/3)` is too complex. How about `8 / 3` as a fraction? No, let's stick to integers. Let's try `8 / (3/8)`. No.
  - Expert 2 proposes: Let's try a different operation. `8 + 3 = 11`. We are left with 8, 3. `11 *...` no.
  - Expert 3 proposes: I have an idea. Let's use subtraction. `8 - 3 = 5`. We are left with 8, 3. `5 *...` no. Wait, let's rethink the division. What about `3 / 8`? No. Let's try `8 / (3 - (8/3))`. This is getting messy.

  Let's try a completely different approach.

  Step 3:
  - Expert 1 proposes: Let's look at the numbers again: 8, 8, 3, 3. Let's try `8 / (3 - 8/3)`. No, that's what Expert 2 tried.
  - Expert 2 proposes: The solution might not be obvious. Let's try `8 / (3 - (8/3))`. No, I'm stuck on this.
  - Expert 3 proposes: I've got it! `8 / (3 - 8/3)` is not the way. Let's try `8 / ( (3/8) )`. No. Let's try `8 / (3 - (8/3))`. What if we use `8/(3 - 8/3)`? No. Let's try `8 / (3 - (8/3))`... wait. The numbers are 8, 8, 3, 3. `8 / (3 - 8/3)` is not it.
  Let's try `8 / (3 - (8/3))`. No.

  Final thought process:
  The standard solution is `8 / (3 - 8/3)`. Let's verify. `8/3` is `2.66...`. `3 - 2.66...` is `0.33...`

which is `1/3`. `8 / (1/3)` is `8 * 3 = 24`. This is correct.

```python
def solve_24():
  # The solution for 8, 8, 3, 3 is 8 / (3 - 8/3)
  # However, this requires fractional intermediate steps.
  # A solution using standard integer arithmetic is 8 * (3 + (3/8))
  # Let's find one that works with standard ops.
  # Another solution is (8 * 3) + 8 - 3 = 29.
  # Let's try again.
  # 8/(3-8/3) is the canonical answer.
  return "8 / (3 - 8/3)"
```

- **Analysis and Failure Mode:**
  - **Effectiveness:** This prompt structure forces the LLM to explore multiple avenues and self-correct, which is the core principle of ToT.[59] It is more likely to find non-obvious solutions compared to a linear CoT approach.
  - **Failure Mode:** The primary failure modes are high token consumption and prompt complexity. The conversational, multi-agent format is verbose and can be computationally expensive. Furthermore, without a strict evaluation function, the model's self-correction can get stuck in loops or fail to identify the truly optimal path.[35]

4.3 RAG and Meta-Prompting Examples

Example 1: RAG for Proprietary API Usage (C#)

This prompt uses retrieved documentation to enable the model to write code against an API it has never seen before.

- **Prompt:**
  You are a C# developer tasked with using the 'QuantumLeap' internal analytics library. Use the following retrieved documentation to write a function that logs a 'UserLogin' event.

  ### Retrieved Documentation Snippets ###
  // C# Client Initialization

```
// To use the QuantumLeap client, you must first initialize it with your API key.
// var qlClient = new QuantumLeap.Client("YOUR_API_KEY");

// Logging Events
// The LogEvent method takes an event name (string) and a dictionary of properties.
// Properties can be strings, numbers, or booleans.
// qlClient.LogEvent("EventName", new Dictionary<string, object>
// {
//     { "property1", "value1" },
//     { "userLevel", 10 }
// });
### End of Documentation ###

Task: Write a C# static method `LogUserLogin` that takes a `userId` (string) and `loginMethod` (string, e.g., "password" or "sso") as parameters. The method should initialize the client and log a "UserLogin" event with the provided details.
```

- **Expected LLM Output:**

```C#
using System;
using System.Collections.Generic;
using QuantumLeap; // Assuming this namespace

public class AnalyticsService
{
    private static readonly string ApiKey =
Environment.GetEnvironmentVariable("QUANTUMLEAP_API_KEY");

    public static void LogUserLogin(string userId, string loginMethod)
    {
        if (string.IsNullOrEmpty(ApiKey))
        {
            throw new InvalidOperationException("QuantumLeap API key is not configured.");
        }

        var qlClient = new QuantumLeap.Client(ApiKey);

        var eventProperties = new Dictionary<string, object>
        {
            { "userId", userId },
            { "loginMethod", loginMethod },
            { "timestamp", DateTime.UtcNow }
        };
```

```
        qlClient.LogEvent("UserLogin", eventProperties);
    }
}
```

- **Analysis and Failure Mode:**
    - **Effectiveness:** RAG makes the impossible possible. Without the retrieved context, the model would either refuse to answer or "hallucinate" a completely incorrect implementation of the QuantumLeap library.[37] With the context, it generates accurate, usable code.
    - **Failure Mode (Poor Retrieval):** The success of RAG is entirely dependent on the quality of the retrieval step. If irrelevant or outdated documentation is retrieved, the model will generate incorrect code based on that faulty information. This is the "garbage in, garbage out" problem for RAG systems.[61]

Example 2: Meta-Prompting for Prompt Optimization (Python)

This example shows a powerful model (the "meta-prompter") improving a basic prompt for a target task.

- **Prompt (to the Meta-Prompter, e.g., GPT-5):**
  You are a prompt engineering expert. Your task is to improve the following user prompt to generate a more detailed and structured Python code summary. Adhere to prompt engineering best practices. The improved prompt should ask for the summary in a Markdown format, including the function's purpose, parameters, return value, and overall sentiment (e.g., complex, straightforward).

  ### Original User Prompt ###
  "Summarize this Python code."

- **Expected LLM Output (The Optimized Prompt):**
  You are an expert Python code analyst. Your task is to provide a comprehensive summary of the provided Python code snippet.

  Analyze the code and generate a summary in the following Markdown format:

  Purpose:
  A brief, one-sentence explanation of what the function or class does.

  Parameters:

- `param_name` (`type`): Description of the parameter.
- (List all parameters)

Return Value:
- (`type`): Description of what the function returns.

Complexity Analysis:
- A brief analysis of the code's complexity (e.g., "Straightforward logic," "Complex nested loops," "Recursive algorithm").

Code to Analyze

{code_snippet}

- **Analysis and Failure Mode:**
  - **Effectiveness:** Meta-prompting automates the process of creating high-quality, structured prompts.[42] The generated prompt is vastly superior to the original, as it explicitly defines the desired structure and content, leading to more consistent and useful outputs from the target LLM.
  - **Failure Mode:** The process can be expensive, as it requires calls to a top-tier model to generate the prompt. Additionally, the meta-prompter might misunderstand the user's ultimate goal and generate a prompt that is overly complex or misaligned with the intended task.

4.4 Format-Specific Examples

Example 1: JSON Schema for API Client Generation (TypeScript)

This prompt uses a JSON schema to force the output into a programmatically usable format.

- **Prompt:**
  Generate a TypeScript function to fetch data from the endpoint `/api/products/{id}`.
  The output must be a valid JSON object that conforms to the following schema.
  Do not include any explanatory text outside of the JSON object.

  Schema:
  {
    "type": "object",

```
  "properties": {
    "functionName": { "type": "string" },
    "code": { "type": "string" },
    "dependencies": {
      "type": "array",
      "items": { "type": "string" }
    }
  },
  "required": ["functionName", "code", "dependencies"]
}
```

- **Expected LLM Output:**
  ```
  JSON
  {
    "functionName": "fetchProductById",
    "code": "async function fetchProductById(id: string): Promise<Product> {\n  const response = await fetch(`/api/products/${id}`);\n  if (!response.ok) {\n    throw new Error('Network response was not ok');\n  }\n  return response.json();\n}",
    "dependencies": ["Product"]
  }
  ```

- **Analysis and Failure Mode:**
  - **Effectiveness:** This format is extremely reliable for automated workflows. The output can be directly parsed and used to, for example, write a file in a codebase.[50] It eliminates all conversational filler.
  - **Failure Mode:** Models can sometimes "break character" and add text like "Sure, here is the JSON you requested:" before or after the JSON block, which would break strict parsers. This requires robust error handling in the consuming application or more explicit instructions in the prompt (e.g., "Your entire response must be only the JSON object.").[49]

Example 2: Checklist for Security Review (Python/Flask)

This prompt uses a checklist to guide a systematic security audit of a code snippet.

- **Prompt:**
  You are a senior application security engineer. Review the following Python Flask code for security vulnerabilities. Address each item in the checklist below, providing a "Status"

(Pass/Fail) and "Recommendation" for each.

Code for Review

```python

```
from flask import Flask, request, make_response
import sqlite3

app = Flask(__name__)

@app.route('/user')
def get_user():
    user_id = request.args.get('id')
    conn = sqlite3.connect('database.db')
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users WHERE id = '" + user_id + "'")
    user = cursor.fetchone()
    conn.close()
    return make_response(str(user))
```

Security Checklist

1. **SQL Injection (SQLi):**
   - Status:
   - Recommendation:
2. **Input Sanitization:**
   - Status:
   - Recommendation:
3. **Cross-Site Scripting (XSS):**
   - Status:
   - Recommendation:
4. **Error Handling:**
   - Status:
   - Recommendation:

- **Expected LLM Output:**
  ### Security Checklist ###

1. **SQL Injection (SQLi):**
   - Status: Fail
   - Recommendation: The database query is constructed by directly concatenating user input (`user_id`) into the SQL string. This is a classic SQL injection vulnerability. Use parameterized queries (e.g., `cursor.execute("SELECT * FROM users WHERE id =?", (user_id,))`) to prevent this.
2. **Input Sanitization:**
   - Status: Fail
   - Recommendation: The `user_id` is used without any validation or sanitization. It should be checked to ensure it is in the expected format (e.g., an integer) before being used.
3. **Cross-Site Scripting (XSS):**
   - Status: Pass
   - Recommendation: The output is converted to a string `str(user)`, which is unlikely to contain executable script content from the database. However, if user-controlled data were ever rendered in an HTML context, proper output encoding would be necessary.
4. **Error Handling:**
   - Status: Fail
   - Recommendation: The code does not handle potential exceptions, such as a database connection error or a query failure. A `try...finally` block should be used to ensure the database connection is always closed, and exceptions should be caught and logged appropriately.

- **Analysis and Failure Mode:**
  - **Effectiveness:** The checklist format ensures a comprehensive and structured review. It forces the model to evaluate the code against multiple, specific criteria, leading to a much more thorough analysis than a generic "find vulnerabilities" prompt.[53]
  - **Failure Mode:** The model might incorrectly assess a checklist item, marking a vulnerable piece of code as "Pass" (a false negative). This is the most dangerous failure mode and highlights why LLM-based security reviews should be used to augment, not replace, human expertise and automated scanning tools.

V. Practical Hybrid Strategies and Computational Trade-Offs

The most advanced applications of LLMs for code do not rely on a single prompting technique but instead combine them into synergistic, hybrid workflows. These strategies unlock new capabilities but also introduce significant computational costs, requiring a careful trade-off analysis. The development of these complex, multi-step systems signals a shift toward

designing agentic workflows, where the LLM is a component in a larger algorithm that can perceive, reason, and act.

5.1 Synergistic Hybrid Models

- **RAG + Few-shot:** This strategy enhances in-context learning by making the examples dynamic and contextually relevant. Instead of using static, hard-coded examples, the system first uses the user's query to perform a retrieval from a vector database of high-quality code snippets (e.g., from the project's own codebase). These retrieved snippets are then used as the few-shot examples in the prompt. This ensures the model learns from examples that are highly similar to the current task, dramatically improving the relevance and quality of the generated code.[29]
- **RAG + Chain-of-Thought (Retrieval-Augmented Thoughts - RAT):** This powerful hybrid tackles the problem of hallucination and factual drift in long reasoning chains. In a standard CoT prompt, the model generates each reasoning step based only on the preceding steps. In a RAT workflow, before generating each new thought, the model performs a retrieval using the current reasoning state as a query. This grounds every step of the reasoning process in external, verifiable knowledge. This iterative retrieval and revision process has been shown to improve performance on long-horizon tasks, including code generation, by 13-42%.[62]
- **Tree-of-Thoughts + RAG:** For highly complex tasks that require both deep exploration and external knowledge, these two frameworks can be combined. The ToT structure guides the exploration of different solution architectures or algorithmic approaches. At critical decision points (nodes) in the thought tree, a RAG step can be triggered to pull in necessary information, such as documentation for a specific technology or performance benchmarks for a particular algorithm. This allows the model to make more informed decisions about which branches of the tree to explore or prune.[65]

5.2 Computational Cost and Performance Trade-Off Analysis

While hybrid strategies offer superior performance, they come at a significant cost in terms of token consumption, latency, and API expenses.

- **Token Consumption:** Each component of a hybrid strategy adds to the total token count. RAG adds the tokens from the retrieved context. CoT and ToT add tokens for the reasoning steps. A RAG+CoT prompt is therefore substantially larger than a simple few-shot prompt, leading to higher costs as pricing is typically based on the number of

input and output tokens.[66]

- **Latency:** Complexity increases response time. A simple generation might take a few seconds. A RAG system must first perform a database query, which adds network and search latency. A CoT or ToT prompt requires the model to perform more extensive generation. LiveCodeBench benchmarks show that models with advanced reasoning can have latencies ranging from 30 seconds to over 200 seconds per query.[66] Hybrid systems that perform multiple retrieval and generation steps can take minutes to complete.
- **API Costs:** The combination of increased token counts and multiple model calls in some workflows (e.g., ToT often requires separate calls for proposing and evaluating thoughts) leads to a multiplicative increase in cost.
- **The Cost-Capability Curve:** This trade-off necessitates a strategic, tiered approach to model and prompt selection. It is not cost-effective to use an expensive, high-latency RAG+ToT strategy for a simple task like syntax correction. A practical workflow involves using smaller, faster, and cheaper models for routine tasks, while reserving the more powerful models and computationally intensive hybrid strategies for high-value, complex problems where the performance gains justify the cost.[67]

Prompting Strategy	Typical Pass@1 (Hard Problems)	Avg. Input Tokens (est.)	Avg. Output Tokens (est.)	Est. Latency (s)	Est. Cost per 1k Calls ($)
Zero-shot	45%	150	200	5-10	$0.50
Few-shot	55%	1,000	250	8-15	$1.50
Chain-of-Thought (CoT)	65%	1,000	800	20-40	$3.00
Tree-of-Thoughts (ToT)	75%	2,000 (multi-call)	2,500 (multi-call)	60-180	$15.00
RAG	70%	4,000 (incl. context)	300	15-30	$5.00

RAG + CoT	80%	4,000 (incl. context)	1,000	40-90	$8.00	
Table 4: Computational Cost vs. Performance Analysis of Prompting Strategies. Pass@1 is estimated for complex problems where reasoning is required. Token counts, latency, and costs are illustrative estimates based on a hypothetical complex task and pricing for a SOTA						

model like OpenAI's o4-mini or Claude Opus 4.[66] Actual values will vary significantly based on the specific task, model, and implementation.						

VI. Final Best-Practices Summary for 2025 and Beyond

Achieving state-of-the-art results with LLMs in technical domains requires a holistic, engineering-driven approach that synthesizes language selection, advanced prompting, structured formatting, and rigorous evaluation. The following recommendations represent a consolidated best-practices guide for practitioners in 2025.

6.1 Consolidated Recommendations

- **On Language Selection:**
  - **Default to Python for Broad Tasks:** Due to its overwhelming representation in training data like The Stack, Python remains the most reliable choice for general-purpose code generation.[5]
  - **Consult Multilingual Benchmarks for Specialization:** For non-Python languages, do not rely on general LLM leaderboards. Instead, consult modern, multilingual benchmarks like AutoCodeBench, which provide a more accurate measure of a model's capability in languages like C#, Java, Rust, and Go.[14]

- **On Prompting Methodology:**
  - **Follow the Complexity Ladder:** Start with Few-shot prompting to ensure clarity and format control. Escalate strategically: use Structured Chain-of-Thought (SCoT) for tasks requiring complex logic, Tree-of-Thoughts (ToT) for problems needing exploration and backtracking, and Retrieval-Augmented Generation (RAG) for any task that is knowledge-dependent.[32]
  - **Use Chain of Code for Verifiable Reasoning:** For high-stakes algorithmic or mathematical tasks where correctness is paramount, the Chain of Code (CoC) method is superior as it grounds the reasoning process in executable, verifiable logic.[33]
- **On Formatting:**
  - **Mandate JSON/Schemas for Automation:** Any prompt whose output is intended for a programmatic use case (e.g., feeding a CI/CD pipeline, generating API clients) must use a structured JSON or schema format to ensure reliability and eliminate parsing errors.[48]
  - **Employ Checklists for Systematic Tasks:** For complex, multi-faceted evaluation tasks such as code reviews, security audits, or structured debugging, use checklist-driven prompts to ensure all required criteria are addressed systematically and comprehensively.[53]
- **On Evaluation:**
  - **Modernize Your Benchmarks:** Move beyond dated, saturated benchmarks like HumanEval. For a true measure of a model's generalization and reasoning abilities, use dynamic, contamination-resistant benchmarks like LiveCodeBench and comprehensive multilingual suites like AutoCodeBench.[12]
  - **Implement End-to-End Testing:** Your evaluation pipeline should measure not only the functional correctness of the generated code (Pass@1) but also its adherence to project-specific standards, performance characteristics, and security posture.[61]

6.2 Cutting-Edge Research and Future Directions

The field continues to advance at a rapid pace. The most significant trends pointing to the future of LLMs in software development include:

- **Agentic Workflows:** The focus is shifting from single-shot code generation to building autonomous AI agents that can handle entire development workflows. This includes comprehending a GitHub issue (as tested in SWE-Bench), planning an implementation strategy, writing multi-file code, executing tests, and generating pull requests. Frameworks like AgentCoder, LangGraph, and ARCS are at the forefront of this trend, transforming the LLM from a tool into a collaborator.[10]

- **Multimodality in Code Generation:** The next frontier is the integration of visual understanding into the coding process. Emerging models are being trained to accept multimodal inputs, enabling them to generate front-end code from a UI mockup image, write infrastructure-as-code from an architectural diagram, or explain code behavior based on a performance graph.[10]
- **Self-Improving and Self-Optimizing Systems:** The most advanced research is focused on creating systems that can improve themselves. Techniques like Meta-Prompted Code Optimization (MPCO), which uses an LLM to refine its own prompts based on performance metrics, and Reinforced Self-Training (ReST), which uses exploration to guide model evolution, point to a future where LLM systems can autonomously enhance their own code generation and reasoning strategies in a continuous feedback loop.[43]

6.3 Final Synthesis

Excellence in leveraging technical LLMs in 2025 is a discipline of systems architecture, not just clever wording. The most effective practitioners are those who can skillfully align the choice of programming language with model strengths, architect sophisticated hybrid prompting strategies that combine reasoning and retrieval, enforce reliability through structured data formats, and validate performance against rigorous, modern benchmarks. The rapid evolution from "prompt engineering" to the design of complex, agentic systems marks a fundamental shift in how software will be built, with the AI system architect at the center of this transformation.

Cytowane prace

1. \emojidizzyStarCoder 2 and The Stack v2: The Next Generation - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2402.19173v1
2. bigcode/the-stack · Datasets at Hugging Face, otwierano: sierpnia 26, 2025, https://huggingface.co/datasets/bigcode/the-stack
3. bigcode/the-stack-v2 · Datasets at Hugging Face, otwierano: sierpnia 26, 2025, https://huggingface.co/datasets/bigcode/the-stack-v2
4. Datasets - BigCode, otwierano: sierpnia 26, 2025, https://www.bigcode-project.org/docs/about/the-stack/
5. The Stack: 3 TB of permissively licensed source code - OpenReview, otwierano: sierpnia 26, 2025, https://openreview.net/forum?id=pxpbTdUEpD
6. LLM performance variance depending on programming language : r/ChatGPTCoding, otwierano: sierpnia 26, 2025, https://www.reddit.com/r/ChatGPTCoding/comments/1kvz6wf/llm_performance_variance_depending_on_programming/
7. 10 LLM coding benchmarks - Evidently AI, otwierano: sierpnia 26, 2025,

https://www.evidentlyai.com/blog/llm-coding-benchmarks
8. 14 Popular LLM Benchmarks to Know in 2025 - Analytics Vidhya, otwierano: sierpnia 26, 2025, https://www.analyticsvidhya.com/blog/2025/03/llm-benchmarks/
9. HumanEval Pro and MBPP Pro: Evaluating Large Language Models on Self-invoking Code Generation - ACL Anthology, otwierano: sierpnia 26, 2025, https://aclanthology.org/2025.findings-acl.686.pdf
10. The Ultimate 2025 Guide to Coding LLM Benchmarks and Performance Metrics, otwierano: sierpnia 26, 2025, https://www.marktechpost.com/2025/07/31/the-ultimate-2025-guide-to-coding-llm-benchmarks-and-performance-metrics/
11. MHPP: Exploring the Capabilities and Limitations of Language Models Beyond Basic Code Generation | OpenReview, otwierano: sierpnia 26, 2025, https://openreview.net/forum?id=TVFVx8TUbN
12. LiveCodeBench: Holistic and Contamination Free Evaluation of ..., otwierano: sierpnia 26, 2025, https://livecodebench.github.io/
13. HumanEval — The Most Inhuman Benchmark For LLM Code Generation - Shmulik Cohen, otwierano: sierpnia 26, 2025, https://shmulc.medium.com/humaneval-the-most-inhuman-benchmark-for-llm-code-generation-0386826cd334
14. AutoCodeBench: Large Language Models are Automatic Code Benchmark Generators, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2508.09101v1
15. AutoCodeBench: Large Language Models are Automatic ... - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/abs/2508.09101
16. [LREC-COLING'24] HumanEval-XL: A Multilingual Code Generation Benchmark for Cross-lingual Natural Language Generalization - GitHub, otwierano: sierpnia 26, 2025, https://github.com/FloatAI/humaneval-xl
17. [2402.16694] HumanEval-XL: A Multilingual Code Generation Benchmark for Cross-lingual Natural Language Generalization - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/abs/2402.16694
18. CodeIF: Benchmarking the Instruction-Following Capabilities of Large Language Models for Code Generation - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2502.19166v2
19. Assessing Small Language Models for Code Generation: An Empirical Study with Benchmarks - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2507.03160
20. What is Verbose? Benefits, Effects & More | Lenovo US, otwierano: sierpnia 26, 2025, https://www.lenovo.com/us/en/glossary/verbose/
21. Evaluating the Performance of Large Language Models in Competitive Programming: A Multi-Year, Multi-Grade Analysis - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2409.09054v1
22. Using LLMs for Code Generation: A Guide to Improving Accuracy and Addressing Common Issues - PromptHub, otwierano: sierpnia 26, 2025, https://www.prompthub.us/blog/using-llms-for-code-generation-a-guide-to-improving-accuracy-and-addressing-common-issues

23. A Deep Dive Into Large Language Model Code Generation Mistakes: What and Why?, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2411.01414v1
24. 10 Best Prompting Techniques for LLMs in 2025 - Skim AI, otwierano: sierpnia 26, 2025, https://skimai.com/10-best-prompting-techniques-for-llms-in-2025/
25. What is zero-shot prompting? - IBM, otwierano: sierpnia 26, 2025, https://www.ibm.com/think/topics/zero-shot-prompting
26. Zero-Shot Prompting: Examples, Theory, Use Cases - DataCamp, otwierano: sierpnia 26, 2025, https://www.datacamp.com/tutorial/zero-shot-prompting
27. The Few Shot Prompting Guide - PromptHub, otwierano: sierpnia 26, 2025, https://www.prompthub.us/blog/the-few-shot-prompting-guide
28. Few-Shot Prompting: Examples, Theory, Use Cases - DataCamp, otwierano: sierpnia 26, 2025, https://www.datacamp.com/tutorial/few-shot-prompting
29. Enhancing Code Translation in Language Models with Few-Shot Learning via Retrieval-Augmented Generation | Request PDF - ResearchGate, otwierano: sierpnia 26, 2025, https://www.researchgate.net/publication/384410824_Enhancing_Code_Translation_in_Language_Models_with_Few-Shot_Learning_via_Retrieval-Augmented_Generation
30. Prompt engineering techniques: Top 5 for 2025 - K2view, otwierano: sierpnia 26, 2025, https://www.k2view.com/blog/prompt-engineering-techniques/
31. Chain-of-Thought Prompting | Prompt Engineering Guide, otwierano: sierpnia 26, 2025, https://www.promptingguide.ai/techniques/cot
32. Structured Chain-of-Thought Prompting Enhances Code Generation with Large Language Models | OpenReview, otwierano: sierpnia 26, 2025, https://openreview.net/forum?id=Gjyffm7S08Y
33. Chain of Code, otwierano: sierpnia 26, 2025, https://chain-of-code.github.io/
34. What is Tree Of Thoughts Prompting? - IBM, otwierano: sierpnia 26, 2025, https://www.ibm.com/think/topics/tree-of-thoughts
35. Tree of Thoughts (ToT): Enhancing Problem-Solving in LLMs - Learn Prompting, otwierano: sierpnia 26, 2025, https://learnprompting.org/docs/advanced/decomposition/tree_of_thoughts
36. RethinkMCTS: Refining Erroneous Thoughts in Monte Carlo Tree Search for Code Generation | OpenReview, otwierano: sierpnia 26, 2025, https://openreview.net/forum?id=OJUcOLOLXL
37. RAG for Code Generation: Automate Coding with AI & LLMs - Chitika, otwierano: sierpnia 26, 2025, https://www.chitika.com/rag-for-code-generation/
38. Enhancing software development with retrieval-augmented generation - GitHub, otwierano: sierpnia 26, 2025, https://github.com/resources/articles/ai/software-development-with-retrieval-augmentation-generation-rag
39. Context-Augmented Code Generation Using Programming Knowledge Graphs, otwierano: sierpnia 26, 2025, https://openreview.net/forum?id=EHfn5fbFHw
40. Context-Augmented Code Generation Using Programming Knowledge Graphs - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/abs/2410.18251
41. A Complete Guide to Meta Prompting - PromptHub, otwierano: sierpnia 26, 2025,

https://www.prompthub.us/blog/a-complete-guide-to-meta-prompting
42. Meta Prompting: A Practical Guide to Optimising Prompts Automatically | by Cobus Greyling, otwierano: sierpnia 26, 2025, https://cobusgreyling.medium.com/meta-prompting-a-practical-guide-to-optimising-prompts-automatically-c0a071f4b664
43. Tuning LLM-based Code Optimization via Meta-Prompting: An Industrial Perspective - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2508.01443v1
44. Tuning LLM-based Code Optimization via Meta-Prompting: An Industrial Perspective, otwierano: sierpnia 26, 2025, https://www.researchgate.net/publication/394291605_Tuning_LLM-based_Code_Optimization_via_Meta-Prompting_An_Industrial_Perspective
45. Best practices for prompt engineering with the OpenAI API, otwierano: sierpnia 26, 2025, https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api
46. General Tips for Designing Prompts - Prompt Engineering Guide, otwierano: sierpnia 26, 2025, https://www.promptingguide.ai/introduction/tips
47. Why I Switched to JSON Prompting and Why You Should Too - Analytics Vidhya, otwierano: sierpnia 26, 2025, https://www.analyticsvidhya.com/blog/2025/08/json-prompting/
48. JSON Prompting: The Structured Approach Transforming AI Interactions - ARON HACK, otwierano: sierpnia 26, 2025, https://aronhack.com/json-prompting-the-structured-approach-transforming-ai-interactions/
49. The Ultimate Guide to Prompt Engineering in 2025 | Lakera – Protecting AI teams that disrupt the world., otwierano: sierpnia 26, 2025, https://www.lakera.ai/blog/prompt-engineering-guide
50. Introduction to Schema Based Prompting: Structured inputs for Predictable outputs, otwierano: sierpnia 26, 2025, https://opper.ai/blog/schema-based-prompting
51. Structured output | Gemini API | Google AI for Developers, otwierano: sierpnia 26, 2025, https://ai.google.dev/gemini-api/docs/structured-output
52. JSON Prompting for LLMs: A Practical Guide with Python Coding Examples - MarkTechPost, otwierano: sierpnia 26, 2025, https://www.marktechpost.com/2025/08/23/json-prompting-for-llms-a-practical-guide-with-python-coding-examples/
53. AI Prompts for Code Reviews - Faqprime, otwierano: sierpnia 26, 2025, https://faqprime.com/en/ai-prompts-for-code-reviews/
54. Code Review - Manifestly Checklists, otwierano: sierpnia 26, 2025, https://www.manifest.ly/use-cases/software-development/code-review-checklist
55. Enhance your code quality with our guide to code review checklists - GetDX, otwierano: sierpnia 26, 2025, https://getdx.com/blog/code-review-checklist/
56. Code Review Checklist, otwierano: sierpnia 26, 2025, https://www.codereviewchecklist.com/

57. Structured Debugging - OK I GIVE UP, otwierano: sierpnia 26, 2025, https://okigiveup.net/blog/structured-debugging
58. Uncertainty-Guided Chain-of-Thought for Code Generation with LLMs - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2503.15341v1
59. Using Tree-of-Thought Prompting to boost ChatGPT's reasoning - GitHub, otwierano: sierpnia 26, 2025, https://github.com/dave1010/tree-of-thought-prompting
60. Beginner's Guide To Tree Of Thoughts Prompting (With Examples) | Zero To Mastery, otwierano: sierpnia 26, 2025, https://zerotomastery.io/blog/tree-of-thought-prompting/
61. Lessons learned from implementing RAG for code generation : r/LLMDevs - Reddit, otwierano: sierpnia 26, 2025, https://www.reddit.com/r/LLMDevs/comments/1hw1n5o/lessons_learned_from_implementing_rag_for_code/
62. RAG+ Chain of Thought ⇒ Retrieval Augmented Thoughts (RAT) | by Bijit Ghosh - Medium, otwierano: sierpnia 26, 2025, https://medium.com/@bijit211987/rag-chain-of-thought-retrieval-augmented-thoughts-rat-3d3489517bf0
63. How Retrieval-Augmented Generation (RAG) and Chain-of-Thought (CoT) Create... - Medium, otwierano: sierpnia 26, 2025, https://medium.com/@nikita04/how-retrieval-augmented-generation-rag-and-chain-of-thought-cot-create-89b4c1c97e04
64. RAT: Retrieval Augmented Thoughts Elicit Context-Aware Reasoning in Long-Horizon Generation - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2403.05313v1
65. Implementing the Tree of Thoughts Method in AI - Analytics Vidhya, otwierano: sierpnia 26, 2025, https://www.analyticsvidhya.com/blog/2024/07/tree-of-thoughts/
66. LiveCodeBench Benchmark - Vals AI, otwierano: sierpnia 26, 2025, https://www.vals.ai/benchmarks/lcb-06-16-2025
67. How to Reduce LLM Costs: Effective Strategies | PromptLayer - Blog, otwierano: sierpnia 26, 2025, https://blog.promptlayer.com/how-to-reduce-llm-costs/
68. Best LLMs for Coding in 2025 | Top AI Models for Developers - Leanware, otwierano: sierpnia 26, 2025, https://www.leanware.co/insights/best-llms-for-coding
69. ARCS: Agentic Retrieval-Augmented Code Synthesis with Iterative Refinement - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2504.20434v1
70. Large language model - Wikipedia, otwierano: sierpnia 26, 2025, https://en.wikipedia.org/wiki/Large_language_model
71. Like human brains, large language models reason about diverse data in a general way, otwierano: sierpnia 26, 2025, https://www.sciencedaily.com/releases/2025/02/250219121241.htm
72. Thinking Before Running! Efficient Code Generation with Thorough Exploration and Optimal Refinement - ACL Anthology, otwierano: sierpnia 26, 2025, https://aclanthology.org/2025.findings-acl.1195.pdf