Exercise 1: Prompting, Debugging and Innovation for Code Generation with LLMs

Sreevidya Bollineni
GitHub Repository: https://github.com/Sreevidya-B/CS520_Exercise1

Part 1: Prompt Design & Code Generation

1.1Overview

This section evaluates the code generation capabilities of two LLMs using three prompting strategies on 10 HumanEval benchmark problems.

Experimental Setup:

- **Models:** GPT-4o (OpenAI), Claude 3.5 Sonnet (Anthropic)
- **Strategies:** Chain-of-Thought (CoT), Self-Planning, Self-Debugging
- **Problems:** 10 HumanEval tasks (IDs: 0, 1, 2, 10, 16, 20, 25, 50, 75, 100)
- **Total Experiments:** 60 (10 problems × 2 models × 3 strategies)
- **Metric:** pass@1 (percentage solved with 1 attempt)

1.2 Problem Selection

Problem IDs	Description
0, 1, 2	List/string operations, basic logic
10, 16, 20, 25	String manipulation, searching
50, 75, 100	Encoding, prime operations, iterations

1.3 Prompting Strategies

Chain-of-Thought (CoT):

Let's solve this step by step.
First, think through the problem:
1. What is the problem asking for?
2. What approach should we use?
3. What are the edge cases?
Then provide your solution: {problem_prompt}

Rationale: Encourages explicit reasoning before implementation.

Self-Planning:

Before implementing, let's create a plan:
Problem: {problem_prompt}
Plan:
- What data structures will you use?
- What is your algorithm?
- What edge cases need handling?
Now implement your plan:
Rationale: Systematic problem decomposition ensures complete solutions.

Self-Debugging:

{problem_prompt}
After writing your solution:
1. Identify potential bugs or edge cases
2. Explain how your code handles them
3. Provide the corrected implementation

Rationale: Self-reflection to catch errors before execution.

1.4 Results

pass@1 Performance:

Model	Strategy	pass@1	Problems Solved
GPT-4o	CoT	90.00%	9/10
GPT-4o	Self-Planning	100.00%	10/10
GPT-4o	Self-Debugging	70.00%	7/10
Claude 3.5 Sonnet	CoT	60.00%	6/10
Claude 3.5 Sonnet	Self-Planning	80.00%	8/10
Claude 3.5 Sonnet	Self-Debugging	60.00%	6/10

Key Findings:

1. **Best Strategy:** Self-Planning achieved highest scores for both models (GPT-4o: 100%, Claude 3.5 Sonnet: 80%)
2. **Model Comparison:** GPT-4o outperformed Claude 3.5 Sonnet across all strategies (+10-30%)
3. **Common Failures:**
   - **Claude 3.5 Sonnet:** Missing from typing import List (4/10 failures in CoT/Self-Debugging)
   - **GPT-4o:** Self-Debugging produced verbose outputs causing incomplete code (3/10 failures)

Failure Patterns:

- **Import Errors (Claude 3.5 Sonnet):** Systematic omission of typing module imports
- **Structural Issues (GPT-4o):** Self-Debugging mixed explanatory text with code

1.5 Analysis

Why Self-Planning Succeeded:

- Forces structured decomposition before coding
- Explicitly addresses data structures and edge cases
- Produces cleaner, implementation-focused code

Why Self-Debugging Underperformed:

- Generated verbose explanations alongside code
- Over-complicated simple problems
- Caused import/structure issues in both models

Part 2: Debugging & Iterative Improvement

2.1 Failure Case Selection

Case 1: HumanEval/1 (separate_paren_groups) - Cross-Model Import Failure

- **Affected:** GPT-4o Self-Debugging, Claude 3.5 Sonnet CoT, Claude 3.5 Sonnet Self-Debugging
- **Error:** *NameError: name 'List' is not defined*
- **Rationale:** Same error, different root causes across models/strategies

Case 2: HumanEval/10 (make_palindrome) - Missing Helper Function

- **Affected:** GPT-4o Self-Debugging only
- **Error:** *NameError: name 'is_palindrome' is not defined*
- **Rationale:** Shows GPT-4o specific weakness; Claude 3.5 Sonnet passed all strategies

2.2 Case 1 Analysis: Missing Import

Problem Definition:

```
from typing import List

def separate_paren_groups(paren_string: str) -> List[str]:
    """ Separate nested parentheses groups into individual strings """
```

GPT-4o Self-Debugging Failure:

```
# Generated code (missing import):
def separate_paren_groups(paren_string: str) -> List[str]:
    result = []
    # ... implementation
```

Root Cause: Verbose debugging output separated imports from executable code

Claude 3.5 Sonnet CoT Failure:

```
# Generated code (no import):
def separate_paren_groups(paren_string: str) -> List[str]:
    result = []
    # ... implementation
```

Root Cause: Model never auto-imports typing module (systematic behavior)

Comparative Insight:

- **GPT-4o:** Strategy-specific issue (works in CoT/Self-Planning)
- **Claude 3.5 Sonnet:** Model-specific limitation (fails consistently)

2.3 Case 2 Analysis: Missing Helper Function

Problem Definition:

```
def is_palindrome(string: str) -> bool:
    return string == string[::-1]

def make_palindrome(string: str) -> str:
    """ Find shortest palindrome beginning with given string """
```

GPT-4o Self-Debugging Failure:

```
# Generated code (missing helper):
def make_palindrome(string: str) -> str:
    for i in range(len(string)):
        if is_palindrome(string[i:]):  # ← is_palindrome not defined!
            return string + string[:i][::-1]
```

Root Cause: Debugging mindset focused only on main function, assumed helper was built-in

GPT-4o CoT Success (same model, different strategy):

```
def is_palindrome(string: str) -> bool:
    return string == string[::-1]

def make_palindrome(string: str) -> str:
    for i in range(len(string)):
        if is_palindrome(string[i:]):
            return string + string[:i][::-1]
```

Why Succeeded: Step-by-step reasoning asked "what functions do I need?"

Claude 3.5 Sonnet: Passed all strategies (includes all provided code by default)

2.4 Prompt Refinements

Applied to All Strategies:
1. Added "CRITICAL REQUIREMENTS" section
2. Explicit import instructions
3. "Code-only output" requirement
4. Helper function inclusion mandate

Refined Chain-of-Thought (CoT) Prompt:

Let's solve this step by step.
CRITICAL REQUIREMENTS:
1. Always include necessary imports at the top (from typing import List, Tuple, Dict, Optional)
2. Provide ONLY executable Python code
3. Include ALL helper functions mentioned in the problem

Think through:
- What imports do I need? (List them explicitly)
- What helper functions are provided? (Include them all)
- What is the main algorithm?
- What are the edge cases?

Now provide your complete solution starting with imports:
{problem_prompt}

Key Changes:

- Added upfront "CRITICAL REQUIREMENTS" checklist
- Explicit instruction to list imports first
- Mandate to include all helper functions

- Simplified reasoning steps to prevent over-explanation

Refined Self-Planning Prompt:

Before implementing, let's create a plan:
CRITICAL REQUIREMENTS:

1. Always include necessary imports at the top (from typing import List, Tuple, Dict, Optional)
2. Include ALL helper functions from the problem
3. Provide ONLY executable Python code

Problem: {problem_prompt}
Plan:

- What imports are needed? (List them first)
- What helper functions are provided? (Include all)
- What data structures will you use?
- What is your algorithm?
- What edge cases need handling?

Now implement your complete plan starting with imports:

Key Changes:

- Added "CRITICAL REQUIREMENTS" before planning
- Prioritized imports and helper functions in planning checklist
- Emphasized "complete plan" including all dependencies
- Explicit "code-only output" requirement

Refined Self-Debugging Prompt:

{problem_prompt}
COMPLETENESS CHECKLIST - Verify before submission:

1. All necessary imports included (from typing import ...)
2. All helper functions from problem included
3. Function signatures exactly match the prompt
4. No explanatory text outside code blocks
5. Common errors checked:
   - Missing imports
   - Missing helper functions
   - Empty input handling

Now provide ONLY the final, complete Python code starting with imports:

Key Changes:

- Replaced "debug/fix" mindset with "completeness verification"
- Listed "Missing imports" and "Missing helper functions" as top errors
- Added explicit "Provide ONLY the final code" to prevent verbose output
- Removed debugging commentary requirement that caused code fragmentation

Summary of Changes Addressing Identified Failures:

For Case 1 (Missing Imports):

- Explicit requirement: "Always include necessary imports at the top"
- Added to thinking/planning steps: "What imports do I need?"
- Positioned imports as first item in all checklists

For Case 2 (Missing Helper Functions):
- Added mandate: "Include ALL helper functions from the problem"
- Added to planning: "What helper functions are provided?"
- Listed as common error in completeness checklist

For Both Cases:
- "Code-only output" prevents mixing explanations with code
- "Complete solution starting with imports" ensures proper structure
- Removed verbose debugging instructions that fragmented code

2.5 Refined Prompts Evaluation

Results Comparison:

Model	Strategy	Original	Refined	Improvement
GPT-4o	CoT	90.0%	100.0%	+10.0%
GPT-4o	Self-Planning	100.0%	100.0%	+0.0%
GPT-4o	Self-Debugging	70.0%	100.0%	+30.0%
Claude 3.5 Sonnet	CoT	60.0%	100.0%	+40.0%
Claude 3.5 Sonnet	Self-Planning	80.0%	100.0%	+20.0%
Claude 3.5 Sonnet	Self-Debugging	60.0%	100.0%	+40.0%

Average Improvement: +23.3%

Key Insights:

- All refined strategies achieved 100% pass@1
- Largest improvements: Claude 3.5 Sonnet CoT/Self-Debugging (+40%), GPT-4o Self-Debugging (+30%)
- Explicit structural requirements (imports, helper functions) eliminated all failures
- Model-specific issues (Claude 3.5 Sonnet imports) resolved with targeted instructions

Part 3: Novel Strategy - Iterative Test-Driven Agent

3.1Strategy Description

Approach: Multi-shot code generation with real test execution feedback loop

Methodology:

1. Generate initial solution with clean, minimal prompt
2. Execute code against actual test cases
3. If tests fail, provide **concrete error messages** to LLM
4. LLM debugs based on **real runtime feedback**
5. Repeat up to 3 iterations or until all tests pass

Rationale: Real test execution provides objective, specific debugging guidance superior to abstract self-reflection.

Base Prompt (Iteration 1):

Solve this problem. Provide ONLY executable Python code with necessary imports.

{problem_prompt}

Debugging Prompt (Iterations 2-3):

Your previous solution failed with this error:
{actual_error_message}

Fix the code and provide the corrected solution:

3.2 Implementation

- **Models:** GPT-4o, Claude 3.5 Sonnet
- **Max Iterations:** 3 per problem
- **Feedback:** Actual Python exceptions and assertion failures from *exec()*
  - **Example:**

    Example Feedback Loop (if iteration was needed):

    Iteration 0: Generate code
    Iteration 1: Error: "NameError: name 'List' is not defined"
      - Provide error to LLM for debugging

    Iteration 2: Fixed code with proper imports
      - Tests pass
- **Timeout:** 5 seconds per test execution

3.3 Results

Performance:

Model	pass@1	First-Try Success	Avg Iterations
GPT-4o	100.0%	10/10	0.0
Claude 3.5 Sonnet	100.0%	10/10	0.0

Observation: Both models solved all problems on first attempt; iterative feedback mechanism was never utilized.

3.4Comparison with Baselines

Improvement Over Part 1 Worst Baselines:

Model	Part 1 Worst	Part 3 Novel	Improvement
GPT-4o	70% (Self-Debugging)	100%	+30.0%
Claude 3.5 Sonnet	60% (CoT/Self-Debugging)	100%	+40.0%

Comparison with Part 2 Refined:

- Part 2 refined strategies also achieved 100%
- Novel strategy matched but did not exceed refined single-shot approaches
- However, novel strategy used simpler base prompt without extensive checklists

3.5 Analysis

Why Iterative Feedback Wasn't Needed:

1. Selected problems are within current LLM capabilities with minimal prompting
2. Simple, clean prompts sufficient for these HumanEval tasks
3. Modern LLMs (GPT-4o, Claude 3.5 Sonnet 3.5) have strong baseline code generation

When This Approach Would Excel:

1. Complex problems with non-obvious edge cases
2. Ambiguous requirements requiring iterative clarification
3. Novel problem types outside training data
4. Subtle runtime bugs only detectable through execution
5. Integration tasks requiring compatibility testing

Value of Infrastructure:

- Framework is ready for harder problems

- Real test feedback is more reliable than self-reflection
- Enables debugging with concrete, actionable information
- Scalable to more complex multi-file projects

Overall, the Iterative Test-Driven Agent achieved 100% pass@1 on both models, matching Part 2 refined strategies. While the iterative mechanism wasn't utilized for these problems (all solved first-try), the approach demonstrates a robust framework for complex code generation where real-time debugging feedback provides value over single-shot generation. The +30-40% improvement over Part 1 baselines validates the importance of prompt engineering, though the specific contribution of iterative feedback cannot be isolated from the cleaner base prompt design.

Summary: All Strategies Across All Parts

Model	Strategy	Part-1	Part-2	Part-3
GPT-4o	CoT	90.0%	100.0%	100.0%
GPT-4o	Self-Planning	100.0%	100.0%	100.0%
GPT-4o	Self-Debugging	70.0%	100.0%	100.0%
Claude 3.5 Sonnet	CoT	60.0%	100.0%	100.0%
Claude 3.5 Sonnet	Self-Planning	80.0%	100.0%	100.0%
Claude 3.5 Sonnet	Self-Debugging	60.0%	100.0%	100.0%
Average		**76.7%**	**100.0%**	**100.0%**

Conclusions:

Key Findings

1. **Strategy Matters:** Self-Planning consistently outperformed other single-shot approaches
2. **Model Differences:**
   - GPT-4o: Better baseline, vulnerable to strategy-induced structural issues
   - Claude 3.5 Sonnet: Requires explicit imports, more literal interpretation
3. **Refinement Impact:** Explicit requirements (+23.3% average improvement) eliminated systematic failures
4. **Iterative Potential:** Test-driven feedback framework ready for harder problems

Learnings

- No universal prompt works across all models
- Explicit structural requirements (imports, helper functions) are critical
- Simple, clean prompts can match complex engineered prompts on straightforward tasks
- Real test execution provides superior debugging signal to self-reflection

GitHub Repository Contents:

- data folder – Contains selected problems and their indices
- generated_code - Model outputs for all experiments
- prompts - All prompt templates (CoT, Self-Planning, Self-Debugging, Refined CoT, Refined Self-Planning, Refined Self-Debugging)
- results folder – contains results for all 3 parts
- Terminal_Outputs folder – contains the outputs that I got on VS code
- Python scripts for all the 3 parts

Repository Link: https://github.com/Sreevidya-B/CS520_Exercise1