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

Name: Lavanika Srinivasaraghavan

Course: CS 520 - Fall 2025

In this report, I have evaluated 3 prompting strategies for LLM code generation across 2 LLMs (Claude Sonnet 4.5 and GPT-5) on 10 HumanEval programming problems. The three strategies evaluated are:

1. **Chain-of-Thought (CoT)**
2. **Self-Debug**
3. **Test-Driven Specification (TDS)** - Novel strategy (Part 3 innovation)

Part 1: Prompt Design & Code Generation

1.1 Objective

Calculate Pass@k metrics (k=1, 2, 3) for Chain-of-Thought and Self-Debugging strategies and compare their performance.

Pass@k Definition: The percentage of problems for which at least one correct solution exists among k generated samples.

Formula:

```
Pass@k = 1 - (C(n-c, k) / C(n, k))
where n = total samples, c = correct samples, k = samples considered
```

1.2 Methodology

Dataset: 10 problems selected from HumanEval benchmark

- problem1: HumanEval/0 - `has_close_elements`
- problem2: HumanEval/1 - `separate_paren_groups`
- problem3: HumanEval/31 - `is_prime`
- problem4: HumanEval/10 - `make_palindrome`
- problem5: HumanEval/54 - `same_chars`
- problem6: HumanEval/61 - `correct_bracketing`

- problem7: HumanEval/108 - `count_nums`
- problem8: HumanEval/32 - `find_zero` (polynomial root finding)
- problem9: HumanEval/105 - `by_length`
- problem10: HumanEval/163 - `generate_integers`

Evaluation Setup:

- LLMs: Claude Sonnet 4.5, GPT-5
- Strategies: Chain-of-Thought (CoT), Self-Debug
- Samples: 3 per problem per strategy per model
- Total solutions: 120 (60 per model)

Prompts are present in prompts directory in the Github link.

1.3 Results

Pass@k Scores

Model	Strategy	Pass@1	Pass@2	Pass@3	Failures
Claude Sonnet 4.5	Chain-of-Thought	100%	100%	100%	0/30
Claude Sonnet 4.5	Self-Debug	100%	100%	100%	1/30
GPT-5	Chain-of-Thought	100%	100%	100%	1/30
GPT-5	Self-Debug	90%	90%	100%	3/30

Key Observations

1. Claude Sonnet 4.5: Excellent performance with 100% Pass@1 for CoT and 100% Pass@1 for Self-Debug
2. GPT-5 CoT: Perfect 100% Pass@1, matched Claude's CoT performance
3. GPT-5 Self-Debug: 90% Pass@1/2, improved to 100% Pass@3 (demonstrates value of multiple samples)
4. Initial failure count: 5 failures in 120 solutions (4.2% failure rate)

Failure Distribution

- **problem1** (has_close_elements): 2 GPT-5 Self-Debug failures
- **problem8** (find_zero): 2 failures (1 Claude Self-Debug, 1 GPT-5 CoT)
- **problem9** (by_length): 1 GPT-5 Self-Debug failure

1.4 Implementation Details

I developed `evaluate_pass_at_k.py` which does the following:

- Automated test execution
- Dynamically load code from file paths
- Error logging
- JSON, CSV, and text report generation
- Pass@k calculation using the formula

Part 2: Debugging & Iterative Improvement

2.1 Objective

Analyze failure patterns, identify root causes, implement fixes, and re-evaluate for improvement.

2.2 Failure Analysis

2.2.1 Problem1: has_close_elements (2 failures)

Failed Solutions:

- `gpt5_problem1_selfdebug_sample1.py`
- `gpt5_problem1_selfdebug_sample2.py`

Failed Prompt:

- `problem1_selfdebug.txt` for GPT5.

Error Type: Function Naming Error

Root Cause:

GPT-5 Self-Debug generated incorrect function name `check_close_elements` instead of required `has_close_elements`.

Error Message:

```
NameError: name 'has_close_elements' is not defined
```

Analysis:

- Self-Debug strategy prompted the model to debug and improve, but it didn't enforce exact naming for the function.

Fix Applied:

```
# BEFORE (incorrect)
def check_close_elements(numbers: List[float], threshold: float) -> bool:
    ...

# AFTER (corrected)
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    ...
```

Result: Both solutions now pass all tests.

2.2.2 Problem8: find_zero (2 failures)

Failed Solutions:

- `claude_problem8_selfdebug_sample1.py`
- `gpt5_problem8_cot_sample1.py`

Failed Prompt:

- `problem8_cot.txt` for both Claude Sonnet4.5 and GPT5.

Error Type: Numerical Precision Error

Root Cause:

Bisection algorithm used insufficient tolerance (1e-6). This caused accumulated floating-point errors to exceed HumanEval's required precision (<1e-4).

Error Message:

```
AssertionError: assert math.fabs(poly(coeffs, solution)) < 1e-4
```

Analysis:

- HumanEval/32 tests with 100 random polynomial coefficients.
- Requires solution precision of |poly(x)| < 1e-4.
- Original implementations used 1e-6 bisection tolerance which where insufficient.

Fix Applied:

```
# BEFORE (insufficient precision)
def find_zero(xs: list):
    begin, end = -1., 1.
    while poly(xs, begin) * poly(xs, end) > 0:
        begin *= 2.0
        end *= 2.0
    while end - begin > 1e-6:
        center = (begin + end) / 2.0
        if poly(xs, center) * poly(xs, begin) > 0:
            begin = center
        else:
            end = center
    return begin

# AFTER (improved precision)
```

```
def find_zero(xs: list):
    begin, end = -1., 1.
    while poly(xs, begin) * poly(xs, end) > 0:
        begin *= 2.0
        end *= 2.0
    while end - begin > 1e-10:
        center = (begin + end) / 2.0
        if poly(xs, center) * poly(xs, begin) > 0:
            begin = center
        else:
            end = center
    return begin
```

Result: Both solutions now pass all 100 random test cases.

2.2.3 Problem9: by_length (1 failure)

Failed Solution: `gpt5_problem9_selfdebug_sample1.py`

Failed Prompt:

- `problem9_selfdebug.txt` for GPT5.

Error Type: Logic Error

Root Cause: Incorrect filtering logic for valid digit names.

Fix: Corrected the filtering condition to properly handle digit name mappings.

Result: Solution now passes all tests.

2.3 Failures Summary

Error Type	Count	Percentage	Examples
Function Naming	3	60%	`check_close_elements` vs `has_close_elements`
Numerical Precision	2	40%	1e-6 tolerance is insufficient for 1e-4 requirement
Logic Errors	1	20%	Incorrect filtering conditions

2.4 Post-Fix Validation

All fixed solutions were re-evaluated, and all got Pass@1 = 100%.

Fixed solutions are saved in `evaluation_results/fixed_solutions/`.

Part 3: Innovation - Test-Driven Specification Strategy

3.1 Design of the Strategy

3.1.1 Motivation

Based on Part 2 analysis, I identified 2 primary failure modes.

1. Function naming errors by models

2. Numerical precision issues.

Hypothesis: A Test-Driven Specification (TDS) strategy that requires models to write tests first (using exact function names) and then only implement code to pass those tests. By doing this, we can eliminate naming errors, improve numerical precision and achieve 100% Pass@1 for both models.

3.1.2 TDS Strategy Structure

TDS follows a three-phase approach in each prompt's structure:

Phase 1: Write Comprehensive Tests

• Write 6-8 test cases using assert statements

• Use exact function name in all assertions

• It should cover: docstring examples, edge cases, boundary conditions, normal cases.

Phase 2: Implement to Pass Tests

• Implement function to pass all Phase 1 tests

• Exact function name from tests should be maintained

• For numerical problems: use stable algorithms (eg. bisection with 1e-10 tolerance)

Phase 3: Verification and Refinement

• Mental trace-through of each test case

• Include self-check questions in the prompt:

- Does function name exactly match specification?

- Can I handle all edge cases?

- Is the numerical precision sufficient (≥1e-10 for bisection)?

• Provide corrected implementation if issues are found.

3.2 Implementation

3.2.1 Prompt Generation

I created a `generate_tds_prompts.py` to automatically generate 10 TDS prompts from HumanEval specifications.

Generated Prompts: `prompts/problem*_tds.txt` (10 files)

3.2.2 Solution Generation

- Finally generated 60 TDS solutions (3 samples × 10 problems × 2 models)
- Stored in: `generated code/{model}/problem/{ model}_problem_tds_sample*.py`

3.2.3 Evaluation Setup

I modified `evaluate_pass_at_k.py` to recognize the TDS strategy.

```
# Added TDS detection (line ~140)
elif 'tds' in filename:
    strategy = 'tds'
```

3.3 Results

3.3.1 Pass@k Performance

Model	Strategy	Pass@1	Pass@2	Pass@3	Failures
Claude Sonnet 4.5	COT	100%	100%	100%	0/30
Claude Sonnet 4.5	SELFDEBUG	100%	100%	100%	1/30
Claude Sonnet 4.5	TDS	100%	100%	100%	2/30
GPT-5	COT	100%	100%	100%	1/30
GPT-5	SELFDEBUG	90%	90%	100%	3/30
GPT-5	TDS	90%	90%	90%	3/30

3.3.2 Aggregate Comparison

Strategy	Avg Pass@1	Total Failures	Success Rate
COT	100.0%	1/60	98.3%

SELFDEBUG	95.0%	4/60	93.3%
TDS	95.0%	5/60	91.7%

Overall: 94.4% success rate (170/180 solutions passed)

TDS underperformed expectations.

3.4 Failure Analysis

3.4.1 TDS Failures

All 5 TDS failures occurred in problem8 (HumanEval/32 - polynomial root finding):

- Claude: 2 failures (`sample1`, `sample3`)
- GPT-5: 3 failures (`sample1`, `sample2`, `sample3`)

3.4.2 Root Cause Analysis

Example Failing TDS Implementation (Claude sample1):

```
def find_zero(xs: list):
    left, right = -1000, 1000

    # Find initial bracket
    while poly(xs, left) * poly(xs, right) > 0:
        left *= 2
        right *= 2

    tolerance = 1e-6
    while right - left > tolerance:
        mid = (left + right) / 2
        mid_value = poly(xs, mid)

        if abs(mid_value) < tolerance:
            return mid

        if poly(xs, left) * mid_value < 0:
            right = mid
        else:
            left = mid

    return (left + right) / 2
```

TDS failed because:

1. Models wrote tests based on docstring examples in Phase 1.

```
assert round(find_zero([1, 2]), 2) == -0.5
```

2. Actual HumanEval tests are much stricter.

```
assert math.fabs(poly(coeffs, solution)) < 1e-4
```

3. Models' self-written tests used `round(..., 2)` (0.01 precision) but HumanEval requires <1e-4.

4. Models could not detect the precision gap during self-verification in Phase 3.

5. The implementation used 1e-6 tolerance which passes self-tests but fails HumanEval.

One solution (uses 1e-10):

```
def find_zero(xs: list):
    begin, end = -1., 1.
    while poly(xs, begin) * poly(xs, end) > 0:
        begin *= 2.0
        end *= 2.0
    while end - begin > 1e-10:
        center = (begin + end) / 2.0
        if poly(xs, center) * poly(xs, begin) > 0:
            begin = center
        else:
            end = center
    return begin
```

3.4.3 Analysis of Error Pattern

All Failures categorized:

- Naming errors: **0**
- Numerical precision: **5**
- Logic errors: **0**

Comparison with Part 2 Patterns:

- Part 2: 60% naming, 40% precision
- TDS: 0% naming, 100% precision.

Key Finding: TDS achieved the goal of eliminating naming errors but failed to improve numerical precision issues.

3.5 Comparative Analysis

3.5.1 Model-Specific Performance

Claude Sonnet 4.5:

- CoT: 100% Pass@1 (0 failures)
- Self-Debug: 100% Pass@1 (1 failure in problem8)
- TDS: 100% Pass@1 (2 failures in problem8)
- **Conclusion**: Claude is works well with all strategies, TDS worked perfectly for Claude

GPT-5:

- CoT: 100% Pass@1 (1 failure in problem8)
- Self-Debug: 90% Pass@1/2, 100% Pass@3 (3 failures: 2 in problem1, 1 in problem9)
- TDS: 90% Pass@1/2/3 (3 failures: all in problem8)
- **Conclusion**: GPT-5 is more sensitive to strategy choice; TDS did not improve compared to self-debug strategy.

3.5.2 Ranking strategies

By success rate:

1. CoT: 98.3% (59/60 solutions)
2. Self-Debug: 93.3% (56/60 solutions)
3. TDS: 91.7% (55/60 solutions)

3.6 Possible improvements

Enhancement 1: Explicit Precision Guidance

```
Phase 2 Addition:
"For numerical algorithms:
- Always use 1e-10 tolerance for bisection/iteration
- Verify that the solution meets <1e-4 precision requirement
- Warning: Docstring examples may use rounding that hides true precision needs"
```

Enhancement 2: Stricter Test Requirements

```
Phase 1 Addition:
"Tests must be more rigorous than docstring examples:
- For numerical problems: Test with math.fabs(result) < 1e-4
- Include 10+ random test cases
- Test boundary conditions at precision limits"
```

Enhancement 3: Stronger Verification

```
Phase 3 Addition:
"Verification checklist:
- Using ≥1e-10 tolerance for numerical algorithms?
- Will solution pass <1e-4 precision tests?
- Have I tested beyond docstring examples?"
```

Overall Conclusion

4.1 Summary of findings

Through this comprehensive evaluation of 3 prompting strategies across 180 code generation tasks, I found that:

Best Strategy is: Chain-of-Thought

• 100% Pass@1 across both models with only 1 failure in 60 solutions (98.3% success rate).

Best Model is: Claude Sonnet 4.5

• 100% Pass@1 across all three strategies with only 3 total failures across 90 solutions.

Novel Strategy (TDS): Partially successful

• Achieved primary goal of removing all naming errors, however it did not improve precision.

• Overall: 95% Pass@1.

Appendix

Problem Mapping

Problem ID	HumanEval ID	Function Name	Description	Difficulty
problem1	HumanEval/0	has_close_elements	Check if numbers close	Easy
problem2	HumanEval/1	separate_paren_groups	Parse parentheses	Medium
problem3	HumanEval/31	is_prime	Primality test	Easy
problem4	HumanEval/10	make_palindrome	Create palindrome	Medium
problem5	HumanEval/54	same_chars	Compare character sets	Easy
problem6	HumanEval/61	correct_bracketing	Validate brackets	Easy
problem7	HumanEval/108	count_nums	Count digit sums	Medium
problem8	HumanEval/32	find_zero	Polynomial roots	Hard
problem9	HumanEval/105	by_length	Sort and name numbers	Medium
problem10	HumanEval/163	generate_integers	Generate integer range	Easy