The Postcondition Auditor: A Rigorous Analysis of LLM Prompting Strategies

Phase 1 Project Report

Software Systems Development Course

IIIT Hyderabad

TEAM NO 32	
NAME	**ROLL NO**
Naveen Mishra	2025201084
Radha Krishna Nallagatla	2025201069
Vipin Yadav	2025201013
Sriram Puruchuri	2025201074
Mohd Shahid Kaleem	2025204008

Table 1: Team Details

1 Project Understanding and Scope

1.1 Problem Statement Analysis

The Postcondition Auditor project addresses a critical challenge in automated software verification: **evaluating the effectiveness of Large Language Models (LLMs) in generating accurate and reliable postconditions**. This research initiative, conducted under the Software Engineering Research Centre (SERC) framework, moves beyond mere demonstration to provide quantifiable, evidence-based conclusions on LLM performance.

The core problem revolves around the **automated generation and evaluation of postconditions** - logical assertions that specify the expected state after function execution. Traditional manual verification approaches are time-intensive and error-prone, making LLM-assisted automation an attractive research direction. However, the reliability and accuracy of LLM-generated postconditions remain uncharted territory requiring systematic investigation.

1.2 Research Objectives and Significance

Our primary objective is to design a **novel evaluation framework** that conducts definitive comparative analysis of three distinct LLM prompting strategies:

- **Naive Prompt**: Direct, zero-shot instruction approach
- **Few-Shot Prompt**: Incorporates three hand-curated exemplar postconditions
- **Chain-of-Thought (CoT) Prompt**: Explicit reasoning about function purpose, invariants, and edge cases

The significance lies in establishing **quantitative benchmarks** for LLM postcondition generation across three critical dimensions: correctness (validity), completeness (strength), and soundness (reliability). This research contributes to the broader field of automated software verification and LLM applications in software engineering.

1.3 Dataset and Constraints

The experiment utilizes a **curated subset of 50 functions** from the Most Basic Programming Problems (MBPP) dataset, each accompanied by function signatures, implementations, and comprehensive input-output test cases. This dataset provides sufficient complexity while maintaining experimental manageability within our 8-week timeline.

2 Team Composition and Persona Analysis

2.1 Team Structure and Philosophy

Our 5-member team adopts an **independent responsibility model** designed to maximize parallel development and minimize inter-dependencies. Each team member will assume primary ownership of a distinct project component while maintaining collaborative oversight of the overall system architecture.

2.2 Proposed Responsibility Matrix

Team Member	Primary Responsibility
Naveen Mishra	LLM Integration & Prompting Strategy Implementation
Radha Krishna Nallagatla	Property-Based Testing & Correctness Evaluation
Vipin Yadav	Mutation Analysis & Completeness Assessment
Sriram Puruchuri	Static Analysis & Soundness Verification
Mohd Shahid Kaleem	Data Pipeline & Results Analysis Framework

Table 2: Independent Responsibility Assignment

This structure ensures that each member can progress independently while contributing to a cohesive final system. Regular integration checkpoints will maintain system consistency and identify potential interface issues early in development.

3 Solution Architecture and Implementation Strategy

3.1 System Architecture Overview

The Postcondition Auditor implements a **modular pipeline architecture** consisting of five core components:

1. **Data Ingestion Module**: MBPP dataset processing and function extraction
2. **Prompting Strategy Engine**: Three-strategy postcondition generation using DeepSeek-Coder
3. **Tripartite Evaluation Framework**: Parallel execution of correctness, completeness, and soundness assessments
4. **Analysis & Reporting Module**: Statistical analysis and visualization generation
5. **Integration & Orchestration Layer**: Workflow coordination and result aggregation

3.2 Evaluation Framework Deep Dive

3.2.1 Correctness Evaluation (Property-Based Testing)

Implements **hypothesis-driven testing** where each generated postcondition undergoes validation against 1000 randomly generated inputs. A postcondition achieves validity only through zero test failures, establishing a stringent correctness threshold. The metric output is the **Percentage of Valid Postconditions** for each prompting strategy.

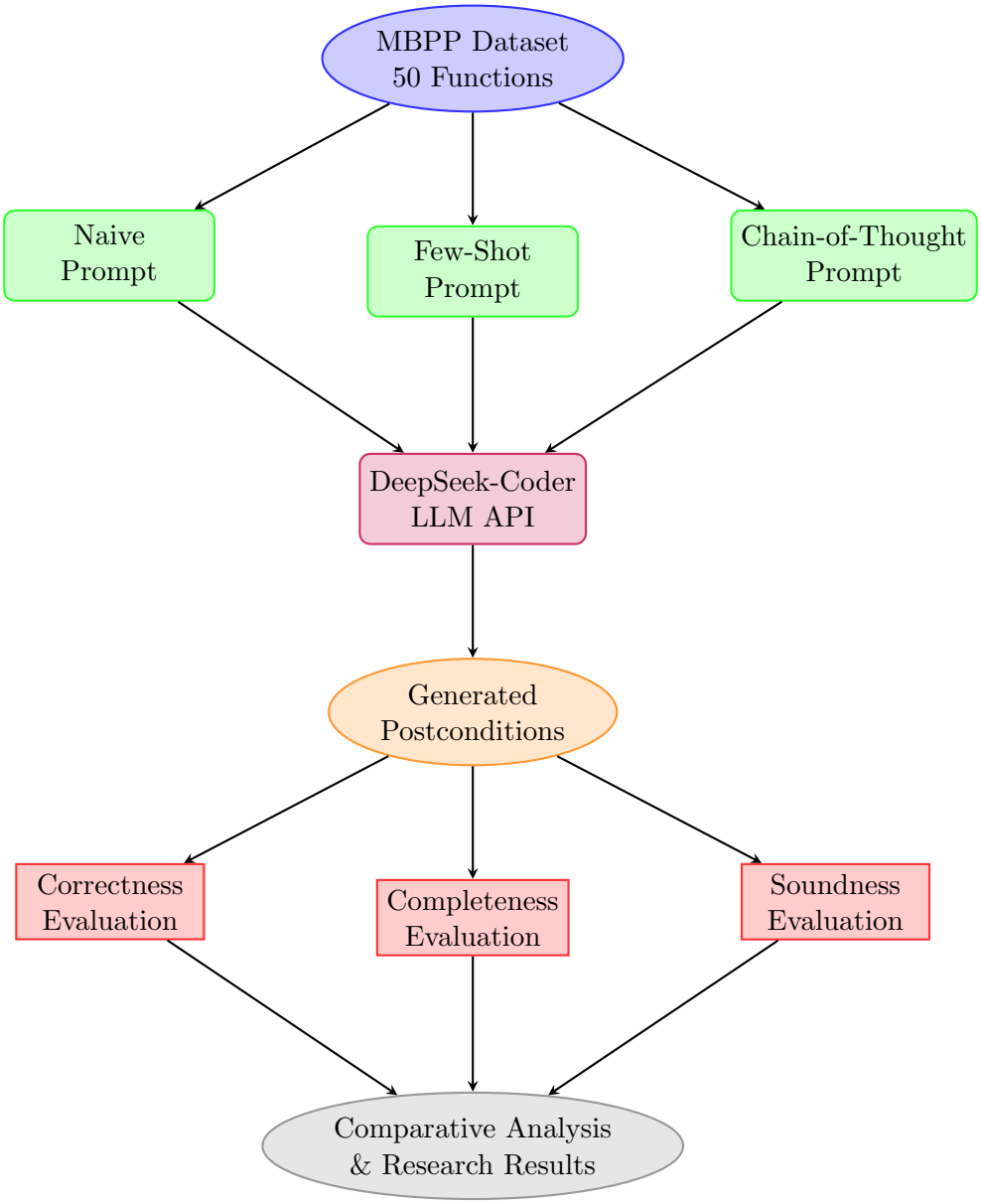
3.2.2 Completeness Assessment (Mutation Analysis)

Employs **systematic mutation testing** where each function generates 5 plausible mutants (boundary condition alterations, operator modifications, off-by-one errors). The **Mutation Kill Score** measures a postcondition's ability to detect these intentional defects, with the Average Mutation Kill Score serving as the primary completeness metric.

3.2.3 Soundness Verification (Hallucination Audit)

Implements **static analysis** using Python's AST library to parse generated postconditions and identify hallucinated variables. Any identifier not present in function parameters, result keywords, or built-ins triggers a hallucination flag. The **Hallucination Rate** quantifies strategy reliability.

3.3 Solution Architecture Diagram



4 Technology Stack and Implementation Details

4.1 Core Technology Selection

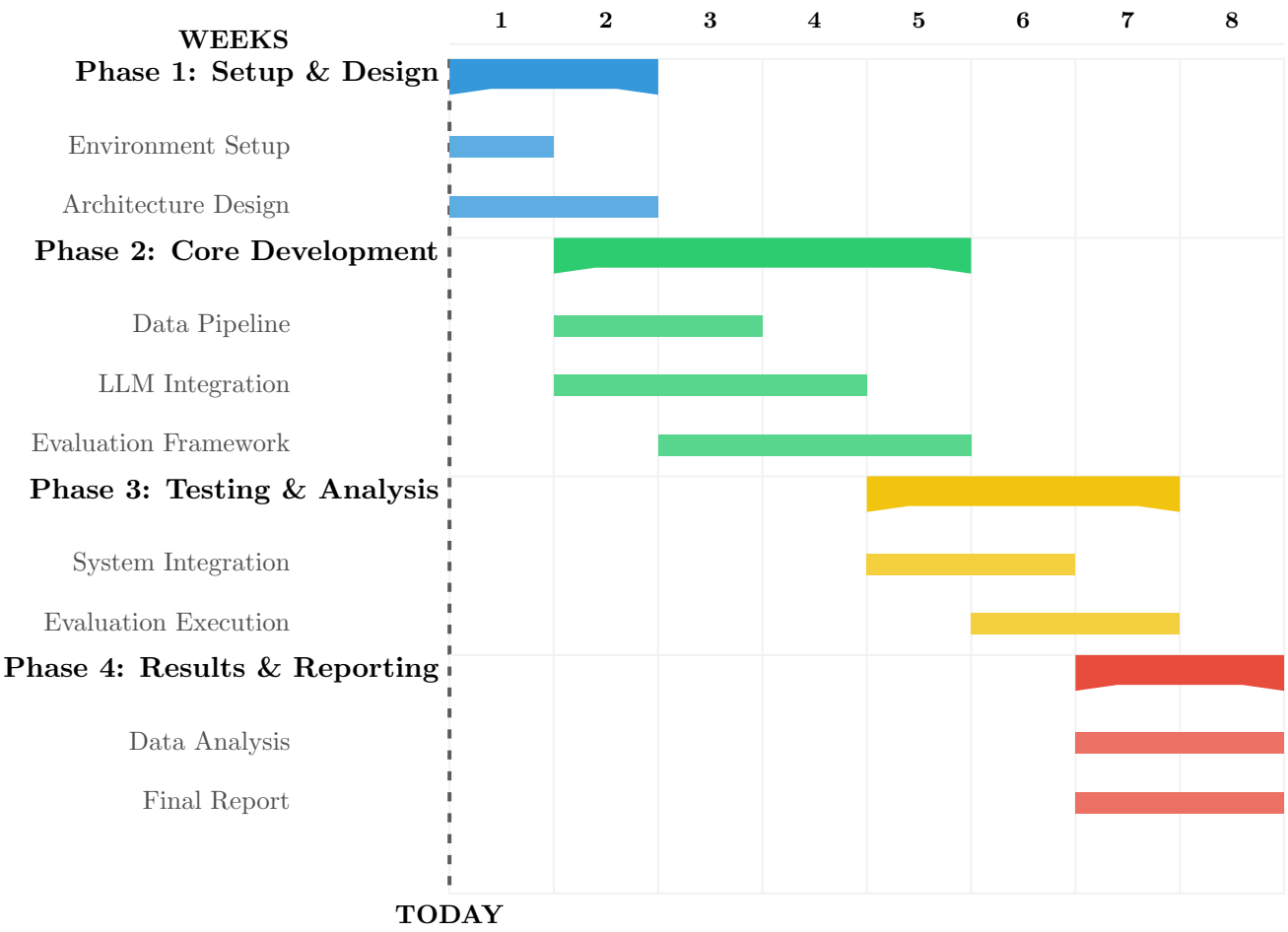
- **Programming Language**: Python 3.10+ (for extensive library ecosystem and LLM integration capabilities)
- **LLM Model**: DeepSeek-Coder (for consistent evaluation baseline)
- **Testing Framework**: Pytest + Hypothesis (property-based testing implementation)
- **Mutation Testing**: Mutmut (systematic mutation generation and analysis)
- **Static Analysis**: Python AST library (postcondition parsing and validation)
- **Data Analysis**: Pandas, Seaborn, Matplotlib (statistical analysis and visualization)
- **Version Control**: Git with collaborative branching strategy

4.2 Development Environment and Infrastructure

The project employs a **containerized development approach** using Docker for environment consistency across team members. CI/CD pipeline integration ensures automated testing and code quality maintenance throughout development cycles.

5 Project Timeline and Milestones

5.1 8-Week Development Schedule



5.2 Detailed Milestone Breakdown

Weeks 1-2: Foundation Phase

- Development environment standardization and Docker containerization
- MBPP dataset acquisition, analysis, and preprocessing pipeline
- System architecture finalization and interface specification
- DeepSeek-Coder API integration and authentication setup

Weeks 3-4: Core Implementation Phase

- Three prompting strategies implementation and testing
- Property-based testing framework development
- Mutation analysis pipeline construction
- Static analysis module for hallucination detection

Weeks 5-6: Integration and Validation Phase

- End-to-end system integration and interface testing
- Pilot evaluation runs with subset of functions
- Performance optimization and error handling implementation
- Quality assurance and unit testing completion

Weeks 7-8: Evaluation and Analysis Phase

- Full-scale evaluation execution across all 50 functions
- Statistical analysis and visualization generation
- Comparative analysis report compilation
- Final presentation preparation and documentation

5.3 Risk Mitigation and Contingency Planning

Key identified risks include LLM API rate limiting, mutation testing complexity, and evaluation framework scalability. Mitigation strategies include API quota management, simplified mutation strategies as fallbacks, and performance profiling for optimization opportunities.

6 Expected Outcomes and Success Metrics

The project's success will be measured through **quantitative comparative analysis** demonstrating statistically significant differences between prompting strategies across the three evaluation dimensions. Expected deliverables include a reproducible evaluation pipeline, comprehensive statistical analysis, and actionable insights for LLM-assisted software verification practices.

The research contribution extends beyond immediate course requirements, potentially informing future automated verification research and establishing benchmarks for LLM postcondition generation evaluation.