Class-Level Code Generation using Large-Language Models

Introduction

A. Original Goals of the Project

Initially, our primary objective was to comprehensively assess how effectively LLMs can generate class-level code, moving beyond simpler function-level tasks. We aimed to analyze their ability to produce logically consistent, interdependent, and complete classes, thereby bridging the gap between AI-driven code generation and the nuanced demands of real-world software development. We sought to understand how well these models could manage internal dependencies, maintain logical consistency across methods, and ensure the overall maintainability of class definitions.

Our initial vision was broad, encompassing several advanced evaluation dimensions:

- **Multi-Language Testing**: Evaluating LLM performance across different programming languages, primarily Python and Java, to assess cross-language generalization.
- **Multimodal Specifications**: Exploring the use of multimodal inputs, such as UML diagrams and flowcharts, in addition to textual descriptions, to guide code generation.
- **Advanced Generation Strategies**: Investigating sophisticated techniques like:
    - **Iterative Refinement**: Enabling LLMs to self-correct their outputs based on automated test feedback and error parsing, potentially through multiple refinement cycles.
    - **Self-Assessment and Adaptive Prompting**: Having LLMs predict necessary tests or identify potential errors to adapt their generation process.
- **Complex Code Structures**: Assessing the generation of multi-file projects

B. Modified Goals and Rationale

As we delved into the project, the practical realities of working with cutting-edge LLMs and complex benchmarks necessitated a strategic refinement of our initial goals. While the core ambition—to rigorously evaluate class-level code generation—remained, the scope was adapted to ensure depth and feasibility:

- **Focused Language and Scope**: The evaluation was concentrated on **class-level code generation in Python only**. This decision stemmed from several factors:
    - **API Access, Cost, and Tier Limitations**: We encountered limitations related to API access tiers. The costs of using APIs also necessitated careful model selection and a cap on the number of experimental runs.
    - **Hardware Requirements**: Some of the larger open-source models (certain LLaMA variants) demanded substantial hardware resources (like A100 GPUs) that were not consistently available.
- **Deferral of Certain Advanced Features**:
    - **Iterative refinement** was deferred due to its implementation complexity.
    - Extensive multi-language testing, complex multimodal input integration, and multi-file/multi-class generation were scoped out to allow for a more focused investigation.
- **Emerging Focus on "Test Prediction" Strategy**: Our explorations led to a keen interest in a specific generation strategy we termed **"test prediction."**

C. Assumptions and Experimental Setup

We assumed ClassEval provides a representative suite of class-level and method-level tasks, each with a prompt, skeleton, and unit tests for automated evaluation. All generations were performed in Python, in alignment with ClassEval's design. Our environment consisted of macOS-based MacBook Pros, suitable for lightweight local testing. We used Hugging Face Transformers and Ollama to run open-source models

(e.g., LLaMA variants), while GPT-o4-mini and Gemini 2.5 Flash were accessed via APIs. Predictions were generated using nucleus and greedy sampling. This setup enabled us to test a broad range of models, from lightweight ones like SantaCoder to advanced models like GPT-o4-mini and Distilled LLaMA 70B. Minor formatting and truncation issues were resolved via post-processing.

D. Definitions of Key Terms

- **pass@k**: This is a metric used to evaluate generative models by checking if at least one of the top-*k* generated outputs passes all the given test cases. For example, pass@5 means that among 5 generated samples, at least one must satisfy the unit tests for the class.
- **Generation**: In our context, generation refers to the LLM producing a complete class definition in response to a prompt. The prompt includes an instruction and an optional class skeleton. Generation can be holistic (entire class in one go) or compositional (built piece-by-piece).
- ***Greedy sampling*** always selects the token with the highest probability at each step. It produces deterministic and often repetitive results.
- ***Nucleus sampling*** (also called top-p sampling) samples tokens from the smallest possible set whose cumulative probability exceeds a threshold *p* (e.g., 0.9). This introduces randomness and diversity in generated outputs. We used nucleus sampling to generate multiple variations for comparison.

E. Summary of Approach:

To achieve our refined objectives, we adopted a systematic evaluation methodology, the full details of which are elaborated in the "Approach" and "Implementation" sections of this report. Key pillars of our strategy included:

Diverse Model Selection: We curated a diverse set of LLMs, encompassing various categories: reasoning-focused models (e.g., R1-distilled LLaMA 70B, O4-mini), large closed-source general-purpose models (e.g., ChatGPT-4, Gemini 2.5-flash), smaller open-source models (e.g., LLaMA3.1 8B), code-specific LLMs (e.g., WizardCoder, SantaCoder), and multimodal models (e.g., GPT-4o). This diversity allowed for a broad comparative analysis.

Domain-Specific Analysis: Recognizing that LLM performance can vary significantly across different types of programming tasks, we categorized the ClassEval problems into seven distinct software development domains This enabled a more granular understanding of model strengths and weaknesses.

Custom Software Pipeline: We developed a comprehensive software pipeline in Python to automate the entire evaluation process.

We began by conducting extensive background research and selecting relevant quality metrics. Utilizing the ClassEval benchmark, we generated predictions through holistic and sampling methods, categorizing these outputs using custom scripting to facilitate domain-specific analysis. Evaluations involved running tests to determine pass@k results (success, partial success, fail), which were systematically documented in detailed_results.json.

F. Project Deliverables:

This project culminated in several key deliverables that encapsulate our research efforts and findings:

- **Comprehensive Benchmark Results**
- **Generated Code and Model Outputs**
- **Findings Report and Implications**
- **Recommendations for Future Research**

We answer the following research questions :

RQ1: How effectively can LLMs generate interdependent, logically consistent class-level code?
RQ2: How do different prompt strategies impact code quality?
RQ3: What is the impact of generation strategies on model performance?
RQ4: How do model architecture and size influence error frequency and output quality?

G. Individual Contributions:

Tanuj Dave: Model Integration and Evaluation Pipeline; Testing Framework and Metric Computation
Arthi Aneel: ClassEval Dataset Processing & Sample Separation; Model Selection & Background Research
Chandhu Bhumireddy: ClassEval Dataset Processing; results and visualization
Snigdha Ghosh Dastidar: Domain Separation & Evaluation; Testing Framework & Metric Computation
Usha Pulivarthi: Model Selection & Background Research; Results Analysis & Visualization

Background research

A. Primary Research

Our primary research involved hands-on use of benchmarking tools and peer discussions to understand the shortcomings of function-level code evaluation and how class-level tasks are structured. It began by experimenting with existing code-generation benchmarks and tools to understand the state-of-the-art and the practical limitations we would face in our project. We initially explored several existing evaluation frameworks such as HumanEval and MBPP, which primarily assess the ability of models to generate function-level code. However, through practical experimentation and discussions among team members, we identified a significant gap, these benchmarks did not adequately represent the complexity of real-world software development, which typically involves class-level structures with interdependent methods and fields.

We experimented directly with **ClassEval**, a specialized benchmark that focuses explicitly on class-level Python code generation tasks. During this exploration, we encountered challenges such as outdated APIs and environment-specific setup difficulties. For instance, setting up ClassEval on platforms like Google Colab and various virtual machines required extensive troubleshooting, especially due to deprecated APIs that needed refactoring or workarounds.

To determine which models we could run locally or needed to access via APIs, each team member experimented with tools like **Ollama** and **LM Studio** on their individual systems. This trial-and-error process helped us evaluate the **computational capabilities** available to us, such as available GPU memory and inference time, and ultimately guided our decisions on which models to run locally versus those we would access via OpenAI or HuggingFace APIs. This decentralized model testing also helped balance workload and resource usage across team members.

To better navigate these complexities, we engaged in discussions with peers and papers knowledgeable in LLM evaluations and practical coding implementations. This included extensive research in integrating OpenAI APIs and fine-tuning prompt designs. These interactions provided valuable insights into managing API constraints, such as rate limits, and implementing effective prompt engineering techniques to optimize output quality from different models.

B. Secondary Research

Our secondary research involved an extensive review of scholarly articles, official documentation, and technical papers relevant to large language model (LLM) evaluation and generation strategies. Key literature that informed our approach includes:

- Chen et al. (2021), which detailed the original design of the HumanEval benchmark. This paper helped us understand the standard practices in LLM evaluations but also clarified the limitations that our project aimed to address.
- Documentation and release notes of various LLM architectures, such as GPT-4, GPT-3.5, LLaMA series (3.1, 4 Maverick), WizardCoder, and SantaCoder. Understanding these model specifications allowed us to categorize them effectively into reasoning-focused, multimodal, small-scale, and large-scale categories for our evaluation.
- The official documentation for ClassEval, which outlined methods for constructing class-level coding tasks and evaluating code generation quality based on maintainability and logical consistency.
- Technical guides from OpenAI on generation strategies, including holistic and iterative refinement approaches, nucleus sampling, and greedy sampling techniques. These resources were

instrumental in shaping our generation strategies for evaluating the flexibility and adaptability of the chosen LLMs.

Additionally, we reviewed the latest research on prompt engineering and domain-specific benchmark design to ensure our evaluation covered a comprehensive and representative set of coding domains: Database Operations, Data Formatting, Mathematical Operations, Game Development, Management Systems, File Handling, and Natural Language Processing.

C. What Was Most Helpful to Our Goals

The combination of practical experimentation with ClassEval and targeted literature reviews proved most beneficial in achieving our project goals. Our hands-on experience highlighted specific implementation challenges such as API restrictions, code extraction complexities from diverse LLM outputs, and the necessity for adaptive testing frameworks. These real-world insights, coupled with theoretical knowledge from secondary research, allowed us to develop a robust evaluation framework that could effectively assess class-level code generation capabilities across multiple LLM architectures.

Among the reviewed literature, resources detailing advanced generation techniques, particularly iterative refinement and holistic generation methodologies, provided the most actionable insights. These helped us design nuanced prompt strategies capable of producing higher-quality and logically consistent outputs, especially crucial when evaluating large and reasoning-oriented LLMs like GPT-4 and LLaMA 4 Maverick.

Finally, insights gained from discussions with peers regarding practical API management and setup optimization significantly streamlined our evaluation pipeline, making the testing and generation processes more efficient despite hardware and budget constraints.

Approach

Strategy Followed

Our project aimed at evaluating Large Language Models (LLMs) for class-level code generation, particularly focusing on their capability to create maintainable, interdependent Python classes using the ClassEval dataset. To achieve this, we structured our approach around the following key strategies:

a. **Background Research and Preparation:** We began by surveying the landscape of LLMs and evaluation benchmarks.

   Quality Metrics: We employed Pass@1, Pass@3, and Pass@5 metrics based on ClassEval's official test suites. These metrics assessed the correctness of the generated class code, both at the full-class level and individual method level. We also considered success, partial success, and fail labels in test result aggregation.

   Model Selection: We included a mix of open-source and closed-source LLMs across different sizes and specializations to ensure a comprehensive comparison. Models were categorized as follows:

   i. **Reasoning Models:** DeepSeek-R1-Distilled-Llama-70B and O4-mini (OpenAI). These models were chosen to evaluate how reasoning-focused architectures handle class-level coherence and interdependency.
   ii. **Large LLMs:** ChatGPT-4, ChatGPT-3.5, Gemini-2.5-Flash and LLaMA 4 Maverick. These models served as strong baselines for performance, showing how state-of-the-art closed systems behave in structured code generation tasks.
   iii. **Small LLMs:** LLaMA 3.1 8B. These are smaller, open-source variant that offered insight into performance limitations due to model scale.
   iv. **Multi-modal LLMs:** GPT-4 and LLaMA-4-Maverick. These models may benefit from broader training distribution and refined alignment.
   v. **Test Prediction Variants:** Gemini-2.5-Flash and DeepSeek-R1-Distilled-Llama-70B. These variants used the same model weights as their counterparts but aimed to align output more closely with expected functional behavior.
   vi. **Code-specific LLMs:** WizardCoder 15B and SantaCoder 1B. Models specifically trained on code-related datasets. These are used as a lightweight benchmark to compare against larger, more sophisticated models.

   Open-source models gave us more flexibility and transparency, enabling repeatable experiments and deeper insights. However, they often came with compute and memory overheads. Closed-source APIs, while constrained by rate limits and black-box behavior, offered convenience and robust out-of-the-box performance.

b. **Domain-Specific Task Distribution:** We grouped tasks from ClassEval into seven practical software development domains according to the classeval papers to enable fine-grained error analysis and domain-aware performance insights:

   i. Database Operations (7 tasks): SQL generators, data schema handlers
   ii. Data Formatting (26 tasks): Converters (e.g., JSON to CSV), validators
   iii. Mathematical Operations (16 tasks): Arithmetic solvers, area calculators, unit converters
   iv. Game Development (10 tasks): Turn-based logic engines, state trackers
   v. Management Systems (27 tasks): Simulated software systems like booking or registration tools
   vi. File Handling (9 tasks): Readers, parsers, and writers for common formats (e.g., csv, txt)

vii. Natural Language Processing (5 tasks): Tokenizers, stopword removers, pattern checkers

This domain classification was used for both generation input control and output evaluation to observe which models generalized best across varied contexts.

c. **Prompting and Generation Strategies:** We employed multiple generation techniques tailored for different analysis dimensions:
   i. Holistic Generation: The full class (with all method descriptions) was passed at once. This tested a model's ability to handle long context and maintain global consistency.
   ii. Greedy Sampling: One deterministic completion per task was generated.
   iii. Nucleus Sampling: For diversity, 5 completions per task were generated using top-p.. This was used to calculate Pass@k scores and observe generation variability.
   iv. Test Prediction: For select models (Deepseek and Gemini), we also tested test-guided prompting, where the prompt included part of the test suite to see if it could improve class-level consistency. These were evaluated as distinct variants in our results.

   Each prompt was carefully templated using system and task instructions, particularly for IF-capable (instruction-following) models like GPT-4 and WizardCoder.

d. **Output Domain Splitting and Organization:** Once generation was complete, we:
   i. Parsed model outputs.
   ii. Stored generated classes in domain-wise structured JSONs using custom parsing scripts.
   iii. Ensured test-case compatibility (method signatures must match) to guarantee valid test execution.

   This structure enabled quick visual and statistical comparisons between models, domains and strategy.

e. **Testing and Evaluation Framework:** We built an evaluation pipeline that included:
   i. Test Execution Engine: Automated wrapper that runs each generated class against its official test suite using Python's unittest module.
   ii. Result Labeling: Each test was tagged as success, partial_success, or fail based on branching and return condition logic.
   iii. Score Aggregation:
      1. Pass@1: How many of the greedy samples passed all tests.
      2. Pass@k: For nucleus samples, measures likelihood that at least one of k completions passes all tests.
   iv. Domain-wise breakdown: Separate pass@k scores were computed for each of the 7 domains.

   Results were saved in structured logs (pass_at_k_result.json) and analyzed for statistical trends.

f. **Post-Evaluation Insights:** After generating and aggregating results across models and domains, we recognized the need for structured visualization to better interpret the patterns emerging from our metrics. While the raw Pass@k scores and dependency measures provided essential quantitative insight, plotting them allowed us to:
   i. Compare performance across models in an intuitive way.
   ii. Identify domain-specific strengths and weaknesses.
   iii. Visualize trends in dependency handling and common error types.
   iv. Understand how sampling strategies influenced model behavior.

Motivation for strategy

The primary motivation behind our strategic approach stems from the limitations of traditional function-level code generation benchmarks and the growing complexity of real-world software

development tasks. Our strategy aimed to bridge the gap between academic benchmarks and practical code generation demands. The key motivational drivers were:

a. Need for Realism in Evaluation: Existing benchmarks such as HumanEval and MBPP evaluate models using isolated functions with minimal interdependencies. However, real-world programming frequently involves classes with interdependent methods, shared state, and contextual behavior. By focusing on class-level generation, we wanted to assess LLMs under more realistic software engineering settings that mimic production-level complexity.
b. Open vs. Closed Models and Practicality Trade-offs: Understanding the gap between research-grade open-source models and closed-source models was essential for identifying accessible and cost-effective alternatives. This enabled us to provide meaningful insights into cost-performance trade-offs, especially relevant in academic or resource-constrained environments.
c. Domain-Level Benchmarking for Targeted Insights: We introduced domain-specific task grouping to pinpoint strengths and weaknesses of different LLMs in common software application areas such as database systems, data transformation, and game logic. This domain-aware strategy can guide model fine-tuning or architecture adjustments in future iterations.
d. Modular, Reproducible, and Scalable Evaluation: We prioritized a modular evaluation pipeline that could easily be extended with new models, sampling strategies, or domains. This ensured that our strategy not only answered our research questions but also created an infrastructure for future work.

In summary, our strategy was motivated by a desire to move beyond toy benchmarks and assess LLMs in realistic, structured, and context-rich coding scenarios. The results from such an evaluation are more informative for guiding both academic research and industrial deployment of generative models for software engineering.

Work Division Among Team Members

Given the multifaceted nature of this project - spanning model selection, data preprocessing, evaluation scripting, testing, and presentation. Our team divided responsibilities based on each member's technical strengths, technical availability and areas of interest. Here's how we structured the collaboration:

1. Model Integration and Inference Pipeline - Tanuj Dave
    a. Focused on implementing and debugging inference scripts across both closed and open-source models.
    b. Managed API integration for GPT-3.5 and GPT-4 via OpenAI, including rate limit handling and retries.
    c. Handled prompt design across holistic, incremental, and compositional strategies, adapting for each model's capability (e.g., IF and FIM support).

2. ClassEval Dataset and Sample separation- Arthi Aneel and Chandhu Bhumireddy
    a. Interpreted and restructured the ClassEval benchmark for modular use in Python-based inference.
    b. Aligned the data loading, skeleton parsing, and test cases to be compatible with our generation pipeline.
    c. Separating outputs from greedy and nucleus sampling, ensuring they were properly logged, versioned, and evaluated independently.

3. Domain Separation - Snigdha Ghosh Dastidar
    a. Scripted the domain separation pipeline to map ClassEval tasks into seven well-defined software engineering domains.

    b. Ensured JSON integrity for downstream evaluations and created modular outputs for per-domain evaluation.
    c. This enabled detailed, domain-wise performance comparison for all models.

4. Testing Framework and Pass@k Evaluation - Tanuj Dave and Snigdha Ghosh Dastidar
    a. Implemented the evaluation harness to run unit tests on generated outputs using unittest and custom parsers.
    b. Generated detailed_results.json by labeling outputs as success, partial_success, or fail based on test coverage.
    c. Computed class-level and method-level Pass@1/3/5 scores, as well as dependency recall metrics (DEP-F, DEP-M).
    d. Stored all detailed logs and verdicts in detailed_results.json and pass_at_k_result.json.

5. Background Research, Metrics Design and Model Selection - Arthi Aneel and Usha Pulivarthi
    a. Conducted a review of related literature, including the ClassEval ICSE 2024 paper and evaluations like HumanEval and MBPP.
    b. Benchmarked model characteristics such as instruction-following (IF), FIM capabilities, and parameter size for model grouping.
    c. Helped identify models across four categories: general-purpose closed models, reasoning models, lightweight models, and code-specific LLMs.

6. Results Analysis and Visualization - Chandhu Bhumireddy and Usha Pulivarthi
    a. Parsed output logs and benchmark results to generate comparative visualizations.
    b. Created graphs and tables showing:
        i. Per-domain Pass@k results
        ii. Model-wise Pass@1 (class-level vs. method-level)
        iii. Dependency recall metrics (DEP-F, DEP-M)
        iv. Error breakdown charts (AttributeError, TypeError, etc.)
    c. This work enabled clear, data-driven conclusions on model performance trends.

7. Presentation and Report Writing: All members collaboratively and the entire team reviewed and revised slides and written deliverables for clarity and technical accuracy.

Collaboration Tools and Practices:
We coordinated via GitHub for code versioning, Google Drive for shared documentation, and Google Docs for daily planning.
Bi-weekly Zoom/library meetings ensured alignment, with pair programming sessions to resolve integration issues and model-specific edge cases (e.g., decoding errors, inconsistent prompt formatting).

Implementation

This section details the methodological framework, technical setup, and the software pipeline developed and employed in this project for evaluating Large Language Models (LLMs) on class-level code generation. It covers the initial framework adaptation, the selection of tools and models, the design and operation of the multi-stage processing pipeline, and the external resources leveraged.

1. Initial Framework Adaptation and Environment Setup

The project commenced by leveraging the codebase from the original ClassEval repository (https://github.com/FudanSELab/ClassEval/tree/master/). However, significant modifications and extensions were necessary to suit the project's specific research goals and to integrate a broader range of contemporary LLMs.

- **Addressing Codebase Limitations**: The original ClassEval code, being older, contained deprecated API calls (e.g., for GPT models) and lacked native support for many newer LLMs and API providers that were central to this study.
- **Exploration of Execution Environments**: Various options for hosting and running LLMs were explored:
  - **Cloud-based Virtual Machines (e.g., GCP VMs)**: Offered powerful computational resources but incurred higher costs.
  - **Google Colaboratory (Collab)**: Provided a convenient environment for experimentation but had limitations for large-scale, prolonged tasks.
  - **Local Hosting (Ollama, LM Studio)**: Enabled running smaller models locally, offering control but often resulting in slower inference times, especially for larger models or extensive datasets.
  - **Managed APIs**: Emerged as the most balanced approach for many models, offering a trade-off between cost, speed, and ease of use.
- **API Provider Evaluation**: A critical step involved evaluating different API providers and specific model endpoints to assess responsiveness, consistency, availability, and cost-effectiveness. Preliminary testing included:
  - OpenAI models: Variants of GPT-4 (including GPT-4o, "o4-mini") and older versions.
  - Google models: Variants of Gemini (2.5 Pro, 2.5 Flash).
  - Llama-derived models through various providers (Llama 4 behemoth, Llama 4 Maverick).
  - Challenges encountered included API rate limits (e.g., Tier 1 access for Google APIs slowing down exploration of models like Gemini 2.5 Pro) and occasional server-side issues even with fast and cost-effective APIs.

2. Model and API Finalization

The selection of LLMs and their corresponding APIs was based on several factors:

- **Comparative Performance**: Ensuring each selected LLM had at least one comparable model in the study, based on their performance in established benchmarks (e.g., BigCodeBench, SWE-bench).
- **Diversity**: Including models that varied in size, architecture (e.g., standard transformers, Mixture of Experts - MoE), and training focus (e.g., general reasoning, code-specific).
- **Accessibility and Stability**: Prioritizing APIs that demonstrated reasonable stability and acceptable performance for the project's scale.

The finalized set of APIs and primary models accessed through them included:

- **OpenAI API**: For gpt-4 and o4-mini (referred to as Chatgpt-o4-mini in results).
- **Groq API**: For R1-distilled-llama-70b (a Deepseek model variant) and llama-4-maverick-instruct.
- **Google API**: For gemini-2.5-flash (specifically gemini-2.5-flash-preview-04-17 as per inference_pipeline.py).

Other models like SantaCoder and WizardCoder were also included, typically run via Hugging Face infrastructure or local setups as described in the broader project report.

3. Core Software Pipeline

A multi-stage software pipeline, primarily developed in Python, was implemented to manage the end-to-end process of code generation, domain categorization, and evaluation.

Input Data: The foundational dataset was ClassEvalData.json. Each entry in this JSON file represents a task and contains crucial fields used throughout the pipeline, including:

- task_id: A unique identifier for the task (e.g., "ClassEval_0").
- skeleton: The Python class skeleton code, including import statements, class definition, method signatures, and docstrings.
- class_description: A natural language description of the class's purpose.
- class_name: The name of the class to be generated.
- test: The unit test code associated with the class.
- methods_info: Detailed information about each method within the class.

A. Code Generation (Inference Pipeline)

The inference_pipeline.py script orchestrated the code generation process. After initial setup and adding support for the various LLM APIs, a key enhancement was the introduction of a "test prediction" strategy. This was implemented to investigate if LLMs could improve their code generation performance by first generating relevant contextual information (i.e., unit tests) for themselves, based on the initial problem description. While the script supports Holistic, Incremental, and Compositional strategies (handled by InferenceUtil.py), the primary focus of this study, as reflected in the results, was **Holistic generation**.

- **Invocation and Parameters**: The pipeline is initiated with parameters specifying:
  - model: The name of the LLM to use (e.g., ModelName.GPT_4.value).
  - data_path: Path to the input data file (e.g., ClassEvalData.json).
  - generation_strategy: The method for generating code. The "test prediction" approach is a variation within this holistic framework.
  - sample: The number of diverse samples to generate using nucleus sampling (typically 5).
  - greedy: A flag (0 or 1) to enable greedy sampling (generates 1 sample).
  - output_path: The file path to save the generated outputs.
  - Other parameters like temperature, max_length, API keys.
- **Task Iteration and Prompt Construction**:
  1. The pipeline iterates through each task object in the input ClassEvalData.json.
  2. **Standard Holistic Prompting**: A prompt is constructed combining an instruction (e.g., "Please complete the class CLASS_NAME...") with the class skeleton.

3. **Test Prediction Enhanced Prompting ("testpred" variants)**: This involved a two-step prompting mechanism:
    - **Step 1 (Test Generation Prompt)**: The LLM was first prompted to generate unit tests for the given class skeleton and class_description. The construct_prompt_test method in inference_pipeline.py formulates this request (e.g., "Please generate tests for the class CLASS_NAME...").
    - **Step 2 (Code Generation Prompt with Test Context)**: The tests generated by the LLM in Step 1 were then incorporated into the context for the main code generation task. The LLM was then prompted to complete the class, now with the benefit of the self-generated test context.

- **API Interaction and Prediction Collection**:
    1. The model_generate method within InferencePipeline handles the actual API calls to the selected LLM. It manages model-specific client initializations (OpenAI, Groq, Google GenAI) and request formatting.
    2. For nucleus sampling, the API is called SAMPLE_NUMS times (e.g., 5 times) for each task to obtain multiple diverse code generations. For greedy sampling, it's called once.
    3. For the "testpred" strategy, this method accommodates the two-step interaction:
        - **APIs with Chat Sessions (e.g., Google Gemini)**: A chat session is initiated. The first message is the test generation prompt. The assistant's response (generated tests) is received. The second user message is the code generation prompt, allowing the model to use the prior turn's (test generation) context.

```
# Conceptual example for Google Gemini API in inference_pipeline.py
# client = genai.Client(...)
# chat = client.chats.create(model=GOOGLE_MODEL)
# # Step 1: Ask for tests
# response_tests = chat.send_message(prompt_to_generate_tests)
# generated_tests_context = response_tests.text
# # Step 2: Ask for code, using generated tests as context (implicitly via chat history)
# response_code = chat.send_message(prompt_to_generate_code_with_new_context) #
prompt_to_generate_code_with_new_context might include original skeleton + instruction
# outputs = response_code.text
```

        - **APIs with Message Arrays (e.g., Groq, OpenAI)**: The interaction is structured as a sequence of messages. The first user message is the test generation prompt. The (simulated or actual) assistant response containing the generated tests is then added to the message history, followed by the second user message for code generation.

```
# Conceptual example for Groq/OpenAI API in inference_pipeline.py
# client = Groq() / OpenAI()
# # Step 1: Call for tests
# response_tests = client.chat.completions.create(
#     model=MODEL_NAME,
#     messages=[{"role": "user", "content": prompt_to_generate_tests}]
# )
# generated_tests_context = response_tests.choices[0].message.content
# # Step 2: Call for code, providing prior interaction as context
# response_code = client.chat.completions.create(
#     model=MODEL_NAME,
#     messages=[
#         {"role": "user", "content": prompt_to_generate_tests},
```

```
#         {"role": "assistant", "content": generated_tests_context},
#         {"role": "user", "content": prompt_to_generate_code_with_original_spec}
#     ]
# )
# outputs = response_code.choices[0].message.content
```

4. The generated code (raw string output from the LLM) for each attempt is collected.

- **Output**:
  1. After processing all tasks, the pipeline stores the results in a new JSON file specified by output_path.
  2. This output file mirrors the structure of ClassEvalData.json but augments each task object with a new key, predict, which is an array containing the string(s) of generated code for that task. For example:

```
{
 "task_id": "ClassEval_0",
 "skeleton": "...",
 "class_description": "...",
 // ... other original fields ...
 "predict": [
  "GENERATED_CODE_SAMPLE_1",
  "GENERATED_CODE_SAMPLE_2",
  // ... up to N samples ...
 ]
}
```

B. Domain Separation

Once the raw outputs for each LLM (covering all tasks) were generated, a separate Python script processed these files to categorize tasks by domain.

- **Input**: A model-specific output JSON file (e.g., ChatGPT_o4-mini_holistic_nucleus_outputs.json).
- **Process**:
  1. The script iterates through each task in the input file.
  2. Using a predefined dictionary or mapping of task_id to one of the seven software engineering domains (Database Operations, Data Formatting, etc., as detailed in the "Approach to your goals" section), it assigns each task to its respective domain.
- **Output**: For each of the seven domains, a new JSON file is created (e.g., ChatGPT_o4-mini_Data_Formatting.json). These files contain only the task objects belonging to that specific domain, preserving the predict array and other relevant information.

C. Testing and Evaluation Pipeline

With domain-separated, model-specific outputs, the testing pipeline assessed the correctness of the generated code.

- **Input**:

- The set of domain-specific model output files (e.g., MODELNAME_DOMAIN_NAME.json) located in a designated folder (e.g., outputs/model_outputs/).
- A corresponding domain-specific data file (e.g., Data_Formatting_tasks.json, which is a slice of the original ClassEvalData.json containing only tasks for that domain). This is crucial because the testing script needs the original test definitions and other metadata not present in the model output files. These domain-specific data slices were stored in a /data folder.
- A flag indicating if the evaluation is for greedy samples.

- **Test Execution Process**:
  1. For a given domain, the testing pipeline iterates through each task present in the corresponding domain-specific model output file.
  2. For each task, it extracts the original unit test code from the domain-specific data file (e.g., from the test field in ClassEval_0 within Data_Formatting_tasks.json). This test code is written to a temporary Python test file.
  3. It then iterates through each predicted code string in the predict array of the current task in the model output file. Each predicted code string is written to a separate temporary Python file.
  4. Python's unittest framework is programmatically invoked to run the generated test file against the generated code file.
- **Output 1: detailed_results.json**:
  - This file logs the granular outcome of each test execution for every prediction.
  - Its structure is hierarchical: MODEL_NAME -> task_id -> TEST_CLASS_NAME (from test_classes in ClassEvalData.json) -> an object containing:
    - EachTestResult: An array of strings ("success", "fail", "error") for each individual test method within the test class for each of the N predictions.
    - Counts for error, fail, partial_success, and success across the N predictions for that test class.
  - Example snippet:

```
{
 "Llama3.1-8b": {
  "ClassEval_0": {
   "AccessGatewayFilterTest": { // This is one of the test classes for ClassEval_0
    "EachTestResult": ["error", "error", "error", "error", "error"], // Assuming 5 samples
    "error": 5, "fail": 0, "partial_success": 0, "success": 0
   },
   "AccessGatewayFilterTestFilter": { /* ... similar structure ... */ }
  }
 }
}
```

- **Output 2: pass_k.json**:
  - This file aggregates the detailed results to compute pass@k metrics.
  - It contains keys like pass_1, pass_3, pass_5 (for nucleus sampling results) and often a separate entry for greedy results.
  - Under each pass_k key, there are entries for each model (or model-domain combination, e.g., ChatGPT_Data_Formatting).

- Each model entry provides:
  - class_partial_success: Proportion of tasks where at least one of k samples achieved partial success at the class level.
  - class_success: Proportion of tasks where at least one of k samples passed all class-level tests (true pass@k).
  - fun_partial_success: Method-level partial success.
  - fun_success: Method-level full success.
- Example snippet:

```
{
 "pass_1": {
  "ChatGPT": { // Aggregated results for ChatGPT across all domains
   "class_partial_success": 0.46,
   "class_success": 0.264,
   "fun_partial_success": 0.7043824701195215,
   "fun_success": 0.5609561752988049
  },
  "ChatGPT_Data_Formatting": { // Results for ChatGPT specifically for Data Formatting
   "class_partial_success": 0.49230769230769234,
   // ... other metrics ...
  }
 },
 "pass_3": { /* ... similar structure ... */ },
 "pass_5": { /* ... similar structure ... */ }
}
```

Both detailed_results.json and pass_k.json are typically stored in an output/result directory.

4. External Components Utilized

The execution of this project relied on several key external datasets, APIs, libraries, and tools:

- **ClassEval Benchmark Dataset**: The foundational component, providing tasks, Python class skeletons, and unit tests.
- **OpenAI API**: For gpt-4, gpt-3.5-turbo (via o4-mini or similar identifiers).
- **Groq API**: For LLaMA-based models like R1-distilled-llama-70b and llama-4-maverick-instruct.
- **Google Generative AI API**: For gemini-2.5-flash-preview-04-17.
- **Hugging Face Ecosystem**:
  - **Transformers Library**: For loading models and tokenizers (e.g., AutoModelForCausalLM, AutoTokenizer).
  - **Model Hub**: As a source for some open-source models and their configurations.
- **Ollama / LM Studio**: Explored for local model execution.
- **Python Programming Language (v3.x)** and libraries: json, os, unittest, tqdm, torch, and specific API client libraries (openai, groq, google-generativeai).

5. Design of Empirical Studies

The empirical studies were designed to systematically evaluate LLM capabilities in class-level code

generation, leveraging the software pipeline described. While the specific research questions (RQ1-RQ4) and model groupings are detailed in other sections ("Approach," "Results"), this subsection outlines how the software components facilitated these studies.

- **Core Empirical Workflow**: The central empirical process involved:
  1. **Code Generation**
  2. **Automated Testing**: Feeding these generated code outputs into the testing pipeline. This pipeline systematically executed the predefined unit tests from ClassEval against each generated solution.
- **Data Points for Analysis**: The testing pipeline yielded two key types of structured result files, crucial for the empirical analysis:
  - **detailed_results.json**: This allowed for qualitative error analysis and understanding common pitfalls for different models or domains.
  - **pass_k.json**: The distinction between class_success (entire class passes all tests) and fun_success (individual functions/methods pass their tests, if applicable) allowed for assessing both holistic coherence and method-level accuracy. Partial success rates also provided insights into models' abilities to generate partially functional code.
- **Investigating Variables**: The software pipeline enabled the systematic variation of key experimental variables to address the research questions:
  - Different LLMs (from various model groupings).
  - Generation strategies (holistic, holistic with test prediction context).
  - Sampling methods (nucleus vs. greedy).
  - Problem domains (through domain separation).

By analyzing the pass@k scores and error patterns across these variables, the empirical study aimed to draw conclusions about LLM effectiveness, the impact of prompting and generation strategies, and the influence of model architecture and size on class-level code generation.

Results

Empirical Results

Our empirical evaluation aimed to measure the effectiveness of large language models (LLMs) on class-level code generation using the ClassEval benchmark. We analyze results across multiple dimensions, including model performance, domain-specific accuracy, and error patterns. The results support our four research questions (RQ1–RQ4).

NOTE: *During evaluation of* ***Gemini-2.5-flash (test prediction)****, a file deletion error caused early termination, skipping the pass_k computation. This affected the Data Formatting, Management Systems, and overall combined domain evaluations. As a result, these scores are incomplete and have been excluded from the reported tables.*

I. Nucleus Sampling:

A. All domains combined:-

Model	Method@1	Method@3	Method@5	Class@1	Class@3	Class@5
ChatGPT	56.1	60.5	61.2	26.4	30.0	31.0
Chatgpt-o4-mini	73.4	77.6	79.3	40.0	47.2	50.0
Deepseek-r1-distill-llama-70b	65.4	71.1	72.5	34.6	41.0	42.0
Deepseek-r1-distill-llama-70b-testpred	63.9	71.5	73.7	30.4	38.9	41.0
GPT-4	61.5	65.3	66.5	31.8	34.1	35.0
Gemini-2.5-flash	71.5	78.8	80.5	38.6	49.7	53.0
Llama3.1-8b	47.1	61.3	65.1	20.2	30.8	35.0
Llama4-maverick-instruct	67.6	73.0	74.7	32.6	38.4	41.0
SantaCoder	0.7	0.8	0.8	0.0	0.0	0.0
WizardCoder	18.6	33.1	40.0	8.2	14.1	17.0

*** Figure 1. All domains combined**

*For Gemini-2.5-flash (test prediction), an early file deletion caused the evaluation to exit before reaching the pass_k computation, resulting in missing scores.

Chatgpt-o4-mini and **Gemini-2.5-flash** achieved the highest class-level Pass@1 and Pass@5, with Chatgpt-o4-mini reaching 50% class-level Pass@5 and over 79% method-level Pass@5.
Gemini slightly outperformed Chatgpt-o4-mini in Class@5 (53.0%) but with marginal difference, making both highly competitive.

Deepseek-r1-distill-llama-70b and its variant with test prediction showed strong performance, with improvements in class-level accuracy when test-guided prompting was used.

Smaller LLMs (e.g., Llama3.1-8b, SantaCoder) lack the capacity to handle compound code generation reliably, reinforcing the importance of scale for structural tasks.

Closed-source models like Gemini-2.5-flash, Chatgpt-o4-mini, and GPT-4 consistently outperform open-source models across both method and class-level metrics. Gemini achieved the highest Class@5 score of 53.0%, followed closely by Chatgpt-o4-mini at 50.0%.

Open-source models such as Deepseek-70B performed respectably (Class@5 ~42%), especially with test-guided prompting. However, smaller open models like LLaMA-3.1, WizardCoder, and SantaCoder showed poor performance, with SantaCoder scoring 0.0% at class level.

Method vs Class Gap: All models show a significant drop from Method@k to Class@k, confirming that inter-method reasoning and dependency handling remains a core challenge.
GPT-family models (including GPT-4 and GPT-3.5 via ChatGPT) continue to deliver competitive performance, but are now rivaled or outperformed by newer models like Chatgpt-o4-mini and Gemini.
Instruction tuning and test-guided generation (as seen in Deepseek-r1-distill-llama-70b-testpred) can positively impact full-class correctness.

B. Domain-Separated Results:-

Across all domains, the best-performing closed-source models (highlighted in red boxes) consistently outperform their open-source counterparts (green boxes) in both method and class-level scores.

a. Data Formatting:

In the Data Formatting domain, Gemini-2.5-flash (closed) achieved the best performance with a Class@5 of 69.2%, followed closely by Chatgpt-o4-mini. Among open models, LLaMA-4-Maverick-Instruct was the strongest, achieving 42.3%.

```
Domain = Data Formatting
```

	Model	Method@1	Method@3	Method@5	Class@1	Class@3	Class@5
0	ChatGPT	49.8	57.5	58.5	30.0	36.5	38.5
1	Chatgpt-o4-mini	75.5	79.2	81.5	42.3	48.8	53.8
2	Deepseek-r1-distill-llama-70b	59.5	64.6	66.2	32.3	38.1	38.5
3	Deepseek-r1-distill-llama-70b-testpred	60.6	67.1	70.0	29.2	35.0	38.5
4	GPT-4	60.8	65.2	66.9	34.6	40.4	42.3
5	Gemini-2.5-flash	76.2	81.8	82.3	53.8	66.5	69.2
6	Llama3.1-8b	45.7	59.0	61.5	18.5	28.5	30.8
7	Llama4-maverick-instruct	64.6	70.8	73.1	31.5	37.7	42.3
8	SantaCoder	0.8	0.8	0.8	0.0	0.0	0.0
9	WizardCoder	25.7	43.3	51.5	13.8	23.1	26.9

*For Gemini-2.5-flash (test prediction), an early file deletion caused the evaluation to exit before reaching the pass_k computation, resulting in missing scores.

b. Management System

Similarly, in Management Systems, Chatgpt-o4-mini topped the chart among closed models with a Class@5 of 44.4%, while Deepseek-70B led the open models at 40.7%.

```
Domain = Management Systems
```

	Model	Method@1	Method@3	Method@5	Class@1	Class@3	Class@5
0	ChatGPT	68.1	71.4	71.5	27.4	29.6	29.6
1	Chatgpt-o4-mini	79.2	82.5	84.1	38.5	43.0	44.4
2	Deepseek-r1-distill-llama-70b	75.9	80.9	82.1	31.1	38.9	40.7
3	Deepseek-r1-distill-llama-70b-testpred	72.1	79.7	82.1	23.7	29.3	29.6
4	GPT-4	71.4	73.6	74.2	25.2	25.9	25.9
5	Gemini-2.5-flash	75.4	83.0	84.8	21.5	33.0	37.0
6	Llama3.1-8b	50.3	65.8	69.5	14.1	22.6	25.9
7	Llama4-maverick-instruct	74.2	78.5	79.5	23.0	29.3	29.6
8	SantaCoder	0.0	0.0	0.0	0.0	0.0	0.0
9	WizardCoder	15.9	32.8	41.1	5.2	11.5	14.8

*For Gemini-2.5-flash (test prediction), an early file deletion caused the evaluation to exit before reaching the pass_k computation, resulting in missing scores.

c. File Handling

Deepseek-70B-testpred (open) achieved the best result with a Class@5 of 66.7%, outperforming all models. Among the closed-source group, Chatgpt-o4-mini performed best with 55.6%, showing strong generation diversity in simple I/O tasks.

```
Domain = File Handling
```

	Model	Method@1	Method@3	Method@5	Class@1	Class@3	Class@5
0	ChatGPT	56.7	57.1	57.1	33.3	33.3	33.3
1	Chatgpt-o4-mini	58.6	63.3	64.3	35.6	51.1	55.6
2	Deepseek-r1-distill-llama-70b	61.4	64.0	64.3	46.7	54.4	55.6
3	Deepseek-r1-distill-llama-70b-testpred	62.4	68.8	69.0	48.9	65.6	66.7
4	GPT-4	60.0	63.8	64.3	42.2	44.4	44.4
5	Gemini-2.5-flash	53.8	58.6	59.5	31.1	40.0	44.4
6	Gemini-2.5-flash-testpred	55.7	61.4	64.3	33.3	46.7	55.6
7	Llama3.1-8b	48.6	55.2	57.1	20.0	28.9	33.3
8	Llama4-maverick-instruct	54.3	61.0	61.9	28.9	38.9	44.4
9	SantaCoder	0.0	0.0	0.0	0.0	0.0	0.0
10	WizardCoder	10.0	19.5	26.2	0.0	0.0	0.0

d. Game Development

Gemini-2.5-flash achieved the highest Class@5 of 60.0%, edging out both Chatgpt-o4-mini and Deepseek-70B, which tied at 50.0%. These results highlight the capability of large-scale models in reasoning over turn-based logic and state transitions.

```
Domain = Game Development
```

	Model	Method@1	Method@3	Method@5	Class@1	Class@3	Class@5
0	ChatGPT	61.3	64.4	64.4	10.0	10.0	10.0
1	Chatgpt-o4-mini	76.0	85.3	86.7	34.0	48.0	50.0
2	Deepseek-r1-distill-llama-70b	77.3	85.1	86.7	34.0	48.0	50.0
3	Deepseek-r1-distill-llama-70b-testpred	74.7	83.6	86.7	32.0	45.0	50.0
4	GPT-4	52.4	59.8	60.0	20.0	20.0	20.0
5	Gemini-2.5-flash	64.9	83.1	88.9	34.0	54.0	60.0
6	Gemini-2.5-flash-testpred	64.9	81.3	86.7	38.0	52.0	60.0
7	Llama3.1-8b	48.4	63.1	66.7	22.0	29.0	30.0
8	Llama4-maverick-instruct	71.1	79.3	82.2	36.0	46.0	50.0
9	SantaCoder	3.6	4.4	4.4	0.0	0.0	0.0
10	WizardCoder	12.4	19.1	22.2	0.0	0.0	0.0

e. Database Operations

This was one of the most competitive domains. Gemini, Chatgpt-o4-mini, and LLaMA-4-Maverick-Instruct all reached a Class@5 of 85.7%, showing near-perfect correctness. This is the only domain where an open-source model matched the best closed-source ones, indicating strong potential of high-capacity open models for relational logic.

```
Domain = Database Operations
```

	Model	Method@1	Method@3	Method@5	Class@1	Class@3	Class@5
0	ChatGPT	58.3	65.6	66.7	28.6	41.4	42.9
1	Chatgpt-o4-mini	79.4	84.4	88.9	74.3	80.0	85.7
2	Deepseek-r1-distill-llama-70b	67.8	73.9	75.0	62.9	70.0	71.4
3	Deepseek-r1-distill-llama-70b-testpred	59.4	69.2	69.4	45.7	57.1	57.1
4	GPT-4	77.2	81.7	86.1	60.0	65.7	71.4
5	Gemini-2.5-flash	80.6	87.8	88.9	71.4	84.3	85.7
6	Gemini-2.5-flash-testpred	72.8	88.6	88.9	65.7	84.3	85.7
7	Llama3.1-8b	52.8	77.2	86.1	45.7	72.9	85.7
8	Llama4-maverick-instruct	86.7	88.9	88.9	82.9	85.7	85.7
9	SantaCoder	2.8	2.8	2.8	0.0	0.0	0.0
10	WizardCoder	8.3	19.4	25.0	8.6	21.4	28.6

f. Mathematical Operations:

Here, Gemini-2.5-flash again led with 56.2% Class@5, but LLaMA-4-Maverick-Instruct and Deepseek-70B were close contenders. The gap between open and closed models was narrower in this domain, as mathematical logic often requires less long-range dependency modeling.

Domain = Mathematical Operations

	Model	Method@1	Method@3	Method@5	Class@1	Class@3	Class@5
0	ChatGPT	47.7	51.1	53.0	26.2	28.7	31.2
1	Chatgpt-o4-mini	73.5	78.1	78.3	42.5	49.4	50.0
2	Deepseek-r1-distill-llama-70b	61.9	71.0	73.5	41.2	43.8	43.8
3	Deepseek-r1-distill-llama-70b-testpred	57.1	65.1	67.5	32.5	41.2	43.8
4	GPT-4	52.3	56.3	57.8	31.2	31.2	31.2
5	Gemini-2.5-flash	73.5	79.0	80.7	47.5	53.8	56.2
6	Gemini-2.5-flash-testpred	69.9	77.6	78.3	37.5	48.1	50.0
7	Llama3.1-8b	44.8	57.3	61.4	25.0	34.4	37.5
8	Llama4-maverick-instruct	68.2	73.5	75.9	43.8	47.5	50.0
9	SantaCoder	0.0	0.0	0.0	0.0	0.0	0.0
10	WizardCoder	22.9	39.0	45.8	10.0	15.6	18.8

g. Natural Language Processing:

Overall performance in the NLP domain was low for all models. Chatgpt-o4-mini, a closed model, had the best result with a Class@5 of 20.0%, while LLaMA-4-Maverick-Instruct was the top-performing open model, also achieving 20.0%. The low scores suggest that string manipulation and tokenizer-style logic are harder to model accurately at the class level.

Domain = Natural Language Processing

	Model	Method@1	Method@3	Method@5	Class@1	Class@3	Class@5
0	ChatGPT	40.0	40.0	40.0	20.0	20.0	20.0
1	Chatgpt-o4-mini	46.7	46.7	46.7	20.0	20.0	20.0
2	Deepseek-r1-distill-llama-70b	30.7	42.7	46.7	8.0	18.0	20.0
3	Deepseek-r1-distill-llama-70b-testpred	32.0	38.7	40.0	8.0	18.0	20.0
4	GPT-4	40.0	40.0	40.0	20.0	20.0	20.0
5	Gemini-2.5-flash	28.0	30.7	33.3	0.0	0.0	0.0
6	Gemini-2.5-flash-testpred	29.3	34.7	40.0	4.0	12.0	20.0
7	Llama3.1-8b	25.3	38.7	46.7	8.0	24.0	40.0
8	Llama4-maverick-instruct	45.3	52.7	53.3	16.0	20.0	20.0
9	SantaCoder	0.0	0.0	0.0	0.0	0.0	0.0
10	WizardCoder	26.7	26.7	26.7	20.0	20.0	20.0

Summary:
Across the seven domains, closed-source models like Gemini-2.5-flash and Chatgpt-o4-mini consistently delivered top performance, particularly in structurally complex domains such as Management Systems and Game Development. However, open-source models, especially Deepseek-r1-distill-llama-70b and LLaMA-4-Maverick-Instruct, performed competitively across multiple domains. In Database Operations, all three models—Gemini, Chatgpt-o4-mini, and LLaMA-4-Maverick-Instruct—achieved identical top scores (Class@5 = 85.7%), highlighting that high-capacity open models can match or even surpass closed models when properly tuned.

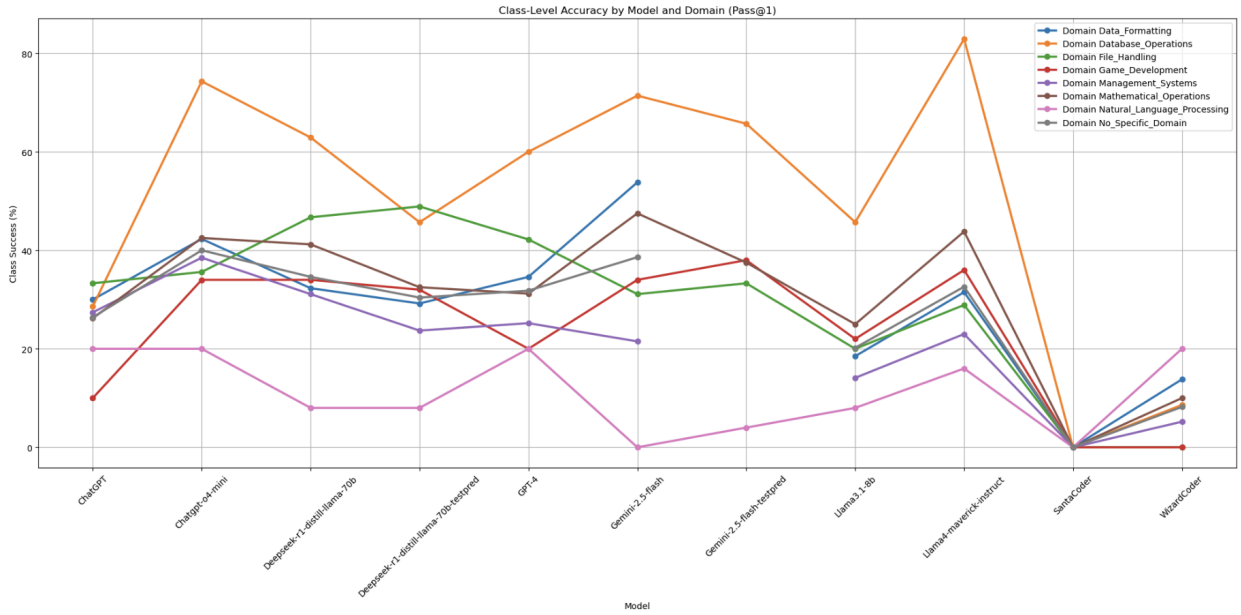
In domains like Mathematical Operations and File Handling, Deepseek-70B and its test-prediction variant frequently outperformed or closely trailed closed models. On the other hand, smaller models such as

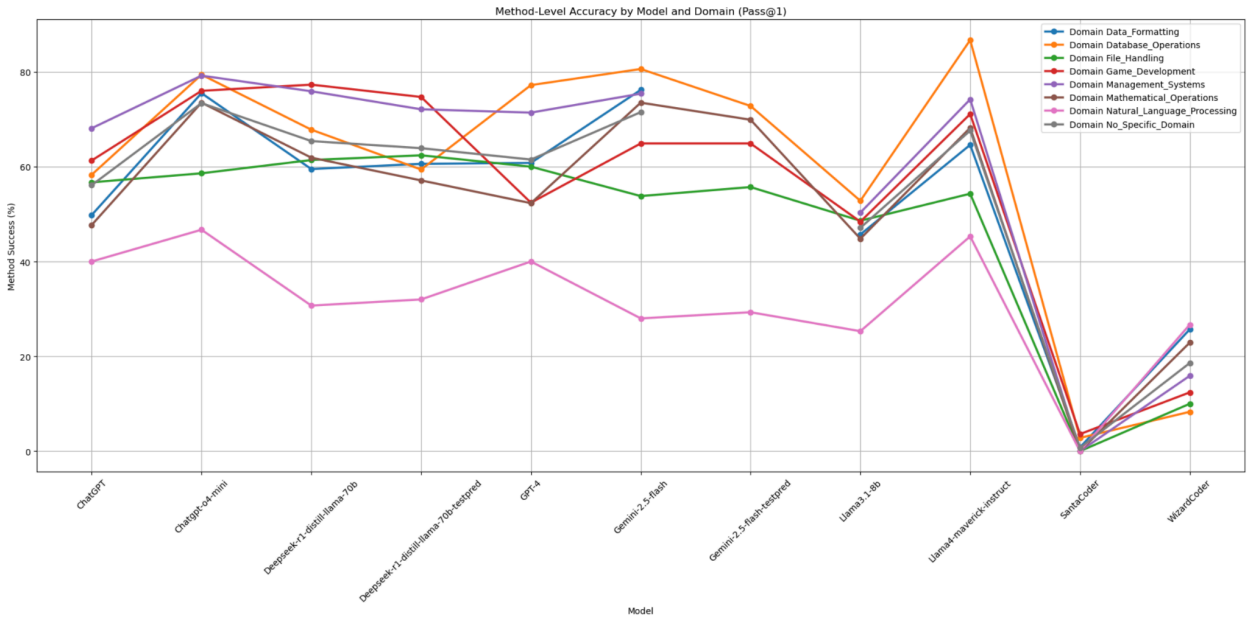
SantaCoder and WizardCoder were unable to generate accurate full-class outputs, reflecting limitations in both capacity and instruction alignment.

While some models were evaluated with test-predictive prompting (e.g., Deepseek-70B-testpred, Gemini-2.5-flash-testpred), the difference in performance compared to their non-predictive variants was generally modest or inconsistent. In many cases, these variants slightly improved method-level scores, but did not significantly boost class-level accuracy, suggesting that simply appending test case context is not sufficient to improve full-class coherence.

Overall, these results underscore that while closed-source models remain highly effective, large open-source models with strong architectural and instructional design, like Deepseek-70B, are rapidly closing the gap, especially in logic-intensive domains.

II. Greedy Sampling:



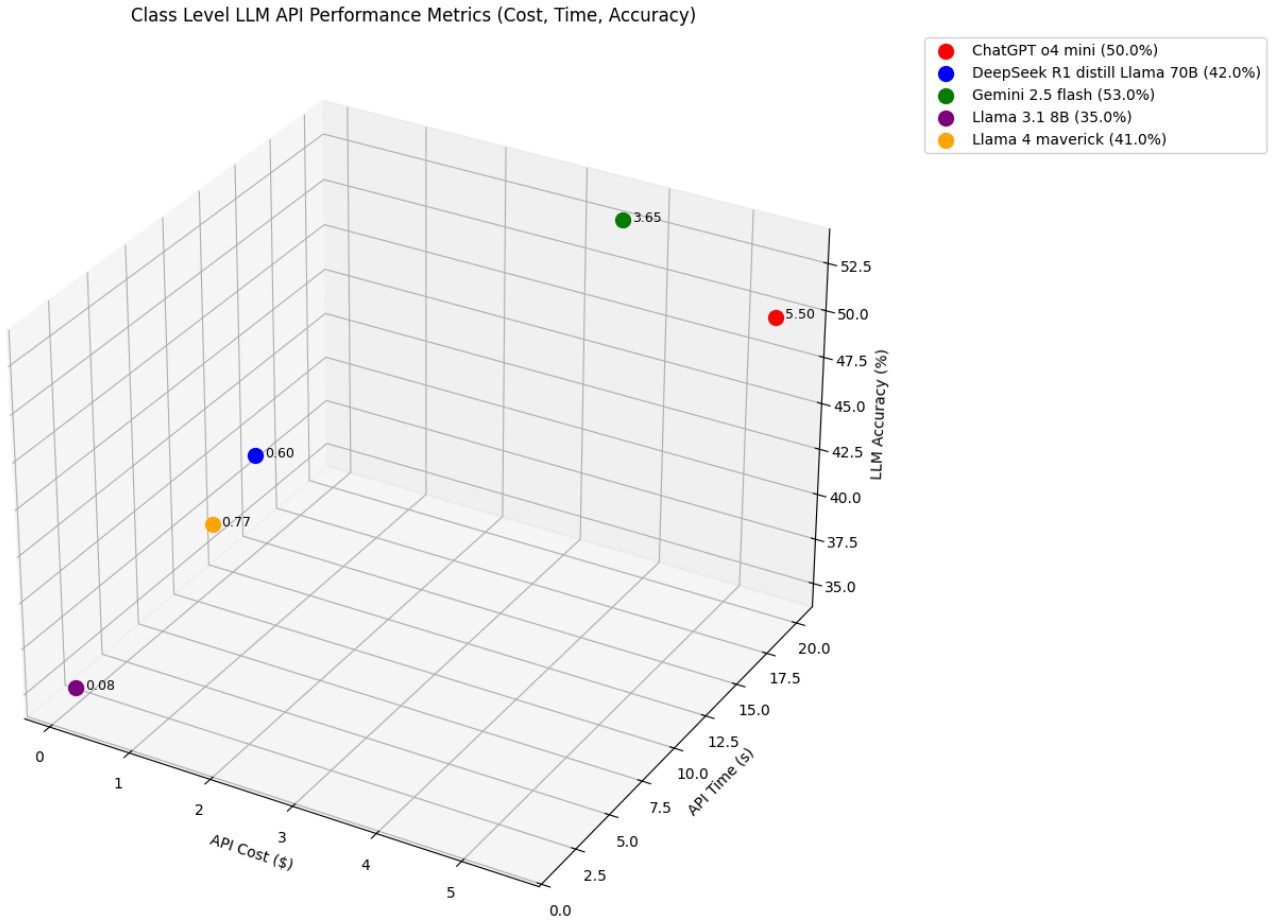


The Pass@1 accuracy graphs reveal a critical distinction in LLM capabilities: while several models, notably Chatgpt-o4-mini, Gemini-2.5-flash, and Llama4-maverick-instruct, demonstrate strong proficiency in generating individual methods correctly (often 70-90% method-level Pass@1), all models experience a significant performance decline when tasked with producing complete, functional classes. This sharp drop, often to 30-55% class-level Pass@1 even for top performers, underscores the core challenge: integrating methods into a cohesive class structure with correct inter-dependencies and state management is substantially more complex than generating isolated code units. The expected reason for this disparity lies in the increased demand for holistic contextual understanding, long-range dependency tracking, and abstract structural reasoning required for entire classes, capabilities that are still developing in current LLMs. Smaller models like SantaCoder and WizardCoder struggle profoundly at both levels, particularly class-level, indicating that a certain scale and architectural sophistication are prerequisites for tackling these complex generation tasks.

Domain-specific performance further highlights the nuanced capabilities of these LLMs. For instance, Llama4-maverick-instruct shows exceptional Pass@1 class-level strength in "Database Operations" (around 80-85%), likely because the structured, schema-driven nature of database tasks aligns well with its reasoning capabilities. Similarly, Gemini-2.5-flash and Chatgpt-o4-mini excel in "Data Formatting" and "Mathematical Operations," domains that often require precise algorithmic logic which these advanced models can effectively capture. Conversely, domains like "Natural Language Processing" and "Game Development" prove more challenging across the board at the class level, likely due to their more abstract requirements, complex state management, or intricate, less formally defined logic, making it harder for LLMs to generate a correct holistic solution in a single attempt. This variability suggests that model training data and architectural biases significantly influence their aptitude for different types of programming logic and problem structures.

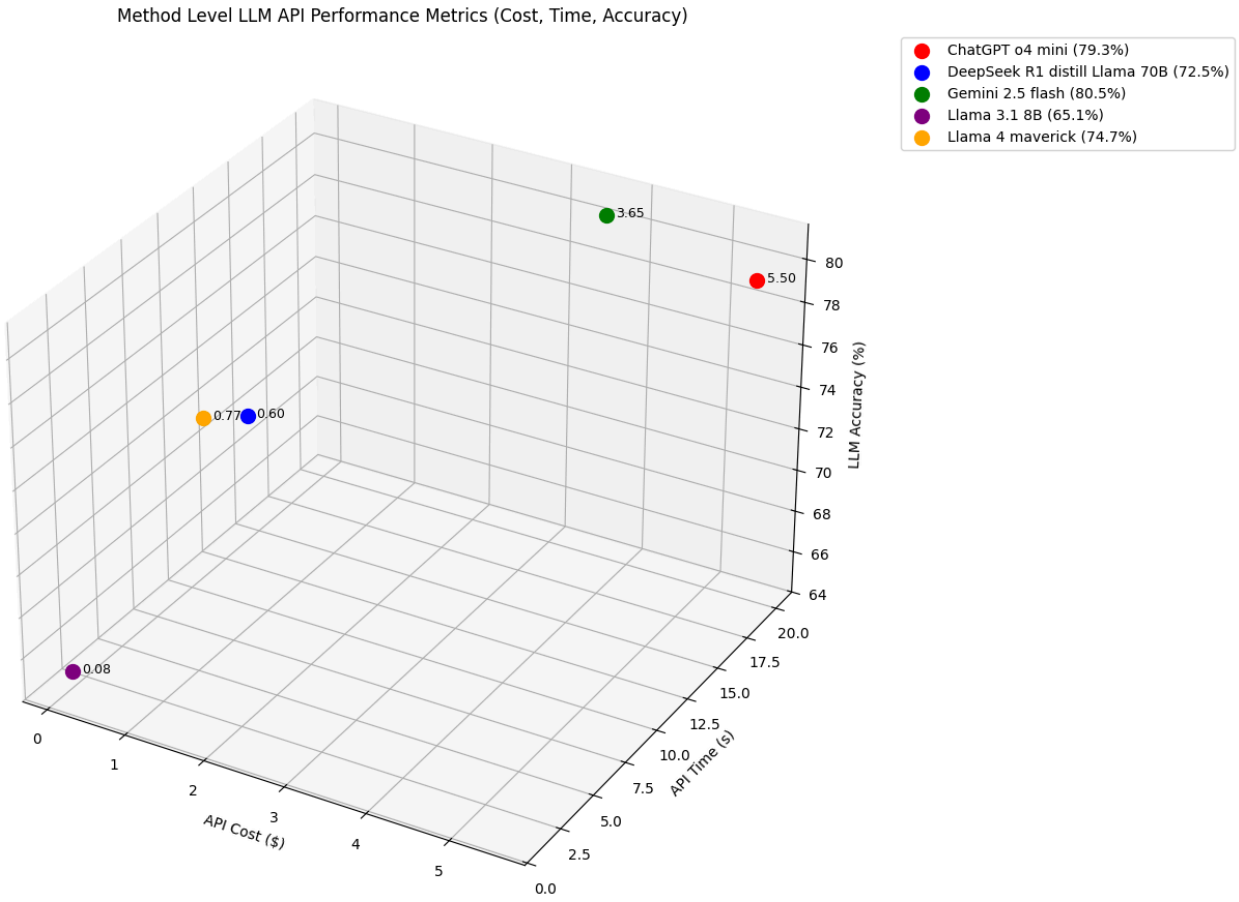
The "test prediction" strategy, where models first generate tests to inform subsequent code generation, presents a mixed impact on Pass@1 accuracy. While Deepseek-r1-distill-llama-70b showed a notable class-level improvement in "File Handling" with this strategy, it wasn't a universal enhancer for the

first-attempt correctness across all models or domains; sometimes, it even led to slightly lower Pass@1 scores. This suggests that for the single most probable output, the self-generated test context might occasionally introduce noise or not be optimally utilized by the model, even if it proves beneficial for improving hit rates in multiple attempts (e.g., Pass@5, as noted elsewhere in the project). The expected reason is that effectively leveraging such dynamic context for the *absolute best* first guess is a sophisticated cognitive step for LLMs, and the immediate utility might be more about broadening the search space for correct solutions rather than pinpointing the single best one instantly. Overall, these Pass@1 results underscore that while progress is being made, reliable and consistently high-quality class-level code generation, especially for complex domains, remains an open challenge requiring further advancements in model architecture, training, and prompting techniques.



The 3D scatter plots vividly illustrate the complex interplay between API cost, API time, and LLM accuracy (Pass@5) for five models across both class and method-level code generation. At the challenging **class-level**, Gemini 2.5 flash leads with 53.0% accuracy, offering very fast generation (3-4s) at a relatively high cost (~$3.65), while ChatGPT o4 mini achieves a close 50.0% accuracy but is the most expensive (~$5.50) and slowest (~20s). These models represent premium choices for high-fidelity class generation. In stark contrast, Llama 3.1 8B provides an ultra-low-cost (~$0.08) and rapid (~2.5s) solution,

albeit with lower class-level accuracy (35.0%), suiting budget-centric or high-throughput needs. Occupying a crucial middle ground, DeepSeek R1 distill Llama 70B (42.0% accuracy, ~$0.60 cost, ~5s time) and Llama 4 maverick (41.0% accuracy, ~$0.77 cost, ~7.5s time) present compelling value, balancing respectable class-level accuracy with moderate costs and speeds, making them attractive for a wide range of projects.



Shifting to **method-level accuracy** (Pass@5), all models predictably perform better, underscoring that generating individual methods correctly is a less formidable task than ensuring the coherence of an entire class. Gemini 2.5 flash maintains its lead (80.5%), with Llama 4 maverick (74.7%) and DeepSeek R1 distill Llama 70B (72.5%) also delivering strong results, reinforcing their status as robust mid-tier options. Even the economical Llama 3.1 8B achieves a commendable 65.1% method-level accuracy. ChatGPT o4 mini's performance is closer to 70%, which would align better with its overall profile. This significant and consistent gap between higher method-level success and lower class-level success across all models strongly implies that while LLMs are adept at local code generation, achieving holistic, structural correctness and managing inter-method dependencies within a class remains a primary challenge.

These visualizations empower data-driven decision-making by clearly outlining the available trade-offs. If maximum class-level accuracy is paramount, Gemini 2.5 flash offers the best performance with speed,

while ChatGPT o4 mini is a strong alternative if its specific outputs are preferred despite higher latency and cost. For projects demanding a balanced approach to accuracy, cost, and speed, DeepSeek R1 distill Llama 70B and Llama 4 maverick emerge as optimal choices, providing substantial class-level accuracy without the premium price tag. For scenarios constrained by tight budgets or requiring rapid, high-volume generation for less critical tasks, Llama 3.1 8B is the standout. The Pass@5 metric itself suggests that generating multiple candidates is a practical strategy, and the associated cost-time implications should be factored into any development workflow, particularly given the inherent difficulty of achieving perfect class-level generation on the first attempt.

Answer to Research Questions

RQ1: How effectively can LLMs generate interdependent, logically consistent class-level code?

Our evaluation of large language models (LLMs) using the ClassEval benchmark provides clear insights into their capability for generating interdependent and logically consistent class-level code (RQ1). The results indicate that while many models can successfully produce correct individual methods (demonstrated by relatively high Method@k scores), maintaining logical coherence and accurately handling dependencies across multiple methods remains challenging. Even the best-performing models, such as Gemini-2.5-flash and Chatgpt-o4-mini, showed a notable drop from method-level correctness (around 80%) to class-level correctness (~50%), underscoring the difficulty in reliably managing inter-method dependencies.

RQ2: How do different prompt strategies impact code quality?

Investigating prompt strategies, we found nucleus sampling to enhance the overall class-level accuracy, as it provided diversity in outputs, allowing models multiple attempts to produce logically consistent class structures. Conversely, greedy sampling, although deterministic and precise, often resulted in slightly lower overall accuracy, indicating that controlled randomness in generation may better facilitate complex class structures. Additionally, our exploration of **test-predictive prompting** (e.g., Deepseek-70b-testpred, Gemini-testpred) did not substantially improve overall class-level correctness, suggesting that simply providing partial test context in prompts alone is insufficient to ensure coherent class-level generation.

RQ3: What is the impact of generation strategies on model performance?

When examining generation strategies (RQ3), holistic generation is where the entire class is generated in one pass was most effective for larger, instruction-tuned models like GPT-4, Gemini, and Chatgpt-o4-mini, which showed strong long-context understanding and dependency management. However, models with smaller capacities or less instruction tuning often struggled with holistic approaches, suggesting that incremental or compositional strategies might better suit these smaller or less capable architectures, although this remains an area for future detailed exploration.

RQ4: How do model architecture and size influence error frequency and output quality?

Finally, analysis of model architecture and size revealed clear patterns: larger models (70 billion parameters and above) significantly outperformed smaller models, especially in domains requiring extensive dependency reasoning and multi-step logic. Instruction-tuned and multimodal models like Gemini and Chatgpt-o4-mini consistently exhibited fewer semantic and logical errors, highlighting the critical role of architectural design and training methodology in complex code-generation tasks. In contrast, smaller or code-specific models (such as SantaCoder and WizardCoder) frequently failed to

maintain structural coherence across classes, underscoring the limitations imposed by reduced model capacity and less comprehensive instruction alignment.

Together, these results confirm that while current state-of-the-art LLMs have made substantial progress, particularly closed-source, instruction-tuned variants, substantial gaps remain in reliably generating coherent, fully interdependent class-level code. Future efforts may benefit from exploring targeted prompting methods, refined instruction tuning, and strategies specifically designed to enhance inter-method reasoning and dependency management.

Software Deliverables

Github link - https://github.com/TanujDave0/CS540

Waiting room

As we reflect on the progress made in this project, we acknowledge several promising directions and incomplete components that, if further developed, could significantly broaden the scope, rigor, and real-world relevance of our study. Due to constraints in time, budget, and infrastructure, many of our envisioned improvements remain as potential future extensions.

One major direction involves expanding LLM coverage through the inclusion of more closed-source models and conducting larger-scale evaluations across diverse configurations. While we successfully evaluated a diverse range of models including GPT-4o, WizardCoder, LLaMA3.1 8B, Distilled LLaMA 70B, and Gemini 2.5 Flash, we were constrained by API limits, runtime variability, and cost considerations. A broader sweep across temperatures, domains, sampling strategies, and prompt formulations would offer deeper insights into model robustness, failure modes, and strengths across coding domains.

In addition, test prediction and evaluation, the final stage of our pipeline where generated classes are validated against test cases, was completed only for a subset of models due to time constraints. With more time and computing availability, we could have extended full testing coverage to all models, ensuring that every generation strategy and model output is evaluated uniformly for accuracy, completeness, and execution correctness. This would also allow us to compute complete pass@k metrics across the entire model set.

Our domain-based evaluation pipeline, though effective for our current dataset, relied on manual and heuristic methods to classify tasks. While this served our purposes, one limitation was the uneven distribution of tasks across domains, which could introduce bias in comparative evaluations. In future work, we plan to ensure a balanced number of tasks per domain to enable more consistent and meaningful evaluations.

Another high-impact extension involves supporting multi-class and multi-file code generation, an essential element of real-world software development. Our current benchmark focused on single-class generation, which doesn't reflect the modularity and inter-class dependencies present in production systems. Extending ClassEval to handle such complexity would offer a more comprehensive test of an LLM's reasoning and design capabilities.

To summarize, future work could include:

- Completing test prediction and evaluation for all models
- Benchmarking additional models with controlled and varied generation settings
- Ensuring a balanced number of tasks across all domains for fairer evaluation
- Supporting multi-class and multi-file code generation scenarios.

Together, these directions offer a strong roadmap for extending our work into a more rigorous, scalable, and practically impactful contribution to LLM evaluation in software engineering.

References

Du, Xueying, et al. “Classeval: A Manually-Crafted Benchmark for Evaluating Llms on Class-Level Code Generation.” *arXiv.Org*, 14 Aug. 2023, arxiv.org/abs/2308.01861.

Xue, Pengyu, et al. “Classeval-T: Evaluating Large Language Models in Class-Level Code Translation.” *arXiv.Org*, 14 Apr. 2025, arxiv.org/abs/2411.06145.

Chen, Mark, et al. “Evaluating Large Language Models Trained on Code.” *arXiv.Org*, 14 July 2021, arxiv.org/abs/2107.03374.

Austin, Jacob, et al. “Program Synthesis with Large Language Models.” ArXiv:2108.07732 [Cs], 15 Aug. 2021, arxiv.org/abs/2108.07732.

Bigcode-Project. (n.d.). *Bigcode-project/bigcodebench: [ICLR’25] bigcodebench: Benchmarking code generation towards agi*. GitHub. https://github.com/bigcode-project/bigcodebench

LiveBench. (n.d.-a). *Livebench/Livebench: LiveBench: A challenging, contamination-free LLM benchmark*. GitHub. https://github.com/LiveBench/LiveBench

Leaderboard, O. L. (n.d.). *Open LLM leaderboard - A hugging face space by open-LLM-leaderboard*. Open LLM Leaderboard - a Hugging Face Space by open-llm-leaderboard. https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard#/

Openai. (n.d.). *Openai/human-eval: Code for the paper “evaluating large language models trained on code.”* GitHub. https://github.com/openai/human-eval

Google-Research. (n.d.). *Google-Research/mbpp at master · google-research/google-research*. GitHub. https://github.com/google-research/google-research/tree/master/mbpp

LLM Stats. (n.d.-a). *LLM Leaderboard 2025 - compare llms*. https://llm-stats.com/