Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

European Master in Software Engineering

Master Thesis

Evaluation of Code Generation Capabilities in Large Language Models

Author: Eduardo Mena Caballero

Madrid, August, 2025

Master Thesis
European Master in
Software Engineering

Title: Evaluation of Code Generation Capabilities in Large Language Models

August, 2025

Author: Eduardo Mena Caballero

Supervisor:	*Co-supervisor:*
Bogdan Korel	Óscar Dieste
Illinois Institute of Technology	Universidad Politécnica de Madrid

Abstract

Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation, yet their effectiveness varies depending on multiple factors, including correctness, efficiency, and maintainability. This study presents a comprehensive evaluation framework for assessing LLM-generated code based on a diverse set of metrics. These include correctness, execution time, cyclomatic complexity, maintainability, style compliance, redundancy, and code similarity, among others. Additionally, generated unit tests are analyzed for coverage and effectiveness. To enable a flexible assessment, a benchmarking system is developed, allowing users to assign different weights to each metric based on the desired attributes of the generated code. Furthermore, the generated code is compared against reference implementations to determine whether LLMs can improve upon existing solutions. A functional analysis is performed using unit tests generated by the models. The results of this study provide valuable insights into the strengths and limitations of LLMs in software development, guiding future improvements and practical applications.

Contents

List of Tables

List of Figures

List of Equations

1 Introduction

1.1 Background on LLMs and Code Generation

Large Language Models, also known as LLMs [33], represent one of the most significant advances in Artificial Intelligence research and Natural Language Processing over the past decade[1]. LLMs, such as OpenAI's GPT series, Google's Gemini, and Grok, have gained a lot of attention due to their wide capabilities in understanding and generating human-like text. Trained on extensive datasets composed with billions of tokens collected from diverse sources, these models have showcased remarkable proficiency in numerous tasks including translation, summarization, question answering, and notably, code generation.

Code generation, in the context of software development, refers to the automatic production of source code by machine learning models based on natural language prompts provided by users. The effectiveness of code generation models can substantially improve developer productivity, reduce time-to-market for software products, and democratize programming by lowering barriers for non-experts. Recently, LLM driven code generation tools, such as GitHub Copilot and OpenAI's Codex, have seen a quick adoption across industries and academic places due to their potential to revolutionize the software development lifecycle.[3] [4]

However, despite the promising capabilities of LLMs in code generation, there exist several challenges and limitations. Generated code quality varies significantly based on several factors such as model size, training data quality, prompting techniques, and task complexity. Additionally, some issues related to correctness, efficiency, readability, and maintainability[2] of generated code continue to present some difficulties for the global adoption of these technologies in critical and large-scale software systems.

Historically, code generation has been approached through various paradigms and methods, ranging from traditional rule-based systems and template-driven approaches to statistical and neural network-based methodologies. Early approaches focused on manually defined patterns and templates, offering limited flexibility and adaptability to simple problems. The introduction of statistical methods marked a transition to more adaptive and probabilistic code generation capabilities. However, it was the creation of deep learning and transformer architectures, specifically transformer-based LLMs, that revolutionized the landscape of automated code generation.

Transformer models, initially designed for natural language understanding tasks, possess unique properties that make them particularly suitable for programming language understanding and generation. These properties include self-attention mechanisms, and big contextual memory, enabling the models to capture in a effectively way syntactic and semantic characteristics in programming languages.

Despite the capabilities shown by modern LLMs, assessing the quality and reliability of generated code are still a complex and critical issue. Traditional software quality assessment methods and metrics may not fully capture the differences associated with LLM-generated code. This highlights the necessity for specialized evaluation frameworks that analyzes and benchmark LLM outputs across various dimensions relevant to software engineering.

This project aims to address these concerns by developing an extensive evaluation framework tailored specifically for LLM-generated code. By employing a range of detailed metrics and comparative analyses, evaluating the quality and functional behaviors of these generated code. This study seeks to present an in-depth understanding of the strengths and limitations of contemporary LLM models in software development contexts.

1.2 Motivation for Evaluating LLM-Generated Code

The motivation for evaluating LLM-generated code emerge from several practical and theoretical considerations. As software becomes essential to critical applications in sectors such as healthcare, finance, and autonomous systems, ensuring the correctness, reliability, and maintainability of code becomes more important. Any errors or defects introduced during the automated code generation process could lead to significant risks, provoking system failures or security vulnerabilities.

Furthermore, despite the trend and enthusiasm around LLM-driven code generation tools, there remains uncertainty about their effectiveness across diverse programming scenarios. Users, both experienced developers and juniors, require clear and objective benchmarks to make decisions about employing these technologies. Without proper testing, people might trust LLMs too much or not enough, which could slow down progress in the development of a project.

Another important motivation is the potential to significantly improve developer productivity. While some studies suggest promising results, making assessments using comprehensive and multidimensional metrics can provide insights into the conditions under which LLM-generated code offers advantages. Those insights could guide software engineering practices and prompting strategies for LLMs, and help optimize their integration into real-world development workflows.

Finally, evaluating LLM-generated code also contributes to a broader scientific understanding of language models capabilities and limitations. By identifying areas of strength and weaknesses, researchers and experts can strategically focus future developments, refine models, and improve performance and safety. This evaluation not only is used for immediate practical needs but also contributes to advancing AI research and its application to software engineering.

1.3 Goals and Scope

The primary goal of this thesis is to develop a robust and comprehensive evaluation framework for assessing code generated by Large Language Models. This framework is designed to quantify various aspects of code quality and functionality, providing stakeholders with clear insights into the capabilities and limitations of current LLM technologies. Specific objectives of the study include:

- Developing a detailed and extensive set of metrics covering correctness, efficiency, maintainability, readability, and reusability of generated code.
- Designing a flexible benchmarking system that allows users to customize metric weights based on their specific requirements and preferences.

- Conducting comprehensive experiments with diverse LLMs across a wide range of coding tasks.
- Analyzing correlations and interactions between different evaluation metrics to identify critical factors influencing code quality.
- Comparing LLM-generated code against human-written reference implementations to evaluate whether LLMs can reliably enhance or substitute traditional software engineering practices.

Since the LLM models analyzed (GPT-4o [34], GPT-4-mini-high [35], Gemini 2.5 [36], Grok 3 [37]) are neither optimized nor trained to produce large projects, for example, applications of several hundred or thousands of lines, the study focuses on scripts and modules of medium complexity, up to about 200 to 500 lines of code. Although there are specialized tools, such as Firebase Studio, capable of outlining complete structures with thousands of lines and multiple files, such solutions fall outside the scope of this work, which aims to rigorously and homogeneously evaluate the coding capabilities of more conventional LLMs under controlled loads.

The scope of this research encompasses a selection of known LLMs and a variety of representative coding tasks that range from simple algorithm implementations to complex software functionalities. The study's results aim to generalize well beyond specific models or tasks, providing insights that can inform future research and practical applications in automated code generation.

2 Related Work

2.1 Prior Research on Code Generation using LLMs

Previous research on code generation using LLMs has primarily focused on exploring the capability of transformer-based architectures, such as GPT models, Codex, and other state of the art neural network frameworks.[5] [6] Studies have highlighted the impressive performance of these models in generating syntactically correct and semantically comprehensive code snippets from natural language descriptions[1]. For instance, early evaluations of GitHub Copilot demonstrated significant promise in real-world coding environments, significantly accelerating developer workflows.[10] [11] [12] [13] [14]

2.2 Existing Benchmarks and Evaluation Methodologies

Numerous benchmarks and evaluation methodologies have been proposed to measure the quality of generated code.[8] Existing benchmarks such as HumanEval and APPS have been widely adopted, focusing primarily on evaluating the correctness of generated code via unit testing and functional equivalence. Other methodologies have also integrated metrics related to complexity, readability, and maintainability, although comprehensive and standardized frameworks remain relatively scarce.[15] [16] [17] [18]

2.3 Comparison with Traditional Software Engineering Practices

Comparing LLM-generated code against traditional software engineering practices reveals critical insights into their practical applicability. Traditional practices focus on structured programming principles, rigorous testing, and iterative development, ensuring high reliability and maintainability. On the other hand, LLM-generated code often requires a more careful validation and post-processing due to potential issues with readability, redundancy, and maintainability. Despite these differences, LLM-generated code offers substantial benefits in rapid prototyping, automating code, and facilitating innovative software development paradigms. Systematic comparative studies are essential to balance these advantages with traditional quality assurances.

3 Evaluation Framework

3.1 Overview of the evaluation pipeline

This subsection will provide an overview of the complete pipeline that automates the evaluation of code generated by various LLM models. Next, the key steps for execution will be shown, along with a brief description of each phase.

3.1.1 Selection of test programs

To ensure an accurate analysis, it's essential that the benchmark encompasses a wide range of programs. To this end, the benchmark includes programs from various functional domains. These include numerical and data utilities, such as a calculator or a program that predicts the next day for a specified date. It also includes search and traversal algorithms, such as a binary search or a DFS search, sorting algorithms, and divide-and-conquer patterns. In addition, simple simulators with state and event logic, such as flappy bird and tictactoe, and integration tasks with external libraries, such as a program for QR code generation or programs for basic audio manipulation, were added. This allows us to evaluate the handling of basic structures, state management, and interaction with APIs.

The chosen programs also span a wide range of sizes and complexities, from scripts of less than 20 lines to modules of more than 200 lines, and different programming styles such as object-oriented programs with various classes and methods, and programs with recursive approaches. Each program includes tests to validate edge cases. This diversity ensures that each LLM model is rigorously tested for functionality, readability, performance and error robustness.

3.1.2 Code generation for each model

To generate the code with the LLMs, a specific prompt structure was employed to ensure a fair evaluation of each model's response under equal conditions. In this phase, a zero-shot instruction prompt with manually executed role prompting at the interface of each model was used.

A **zero-shot instruction prompt** is a prompting technique in which the model receives only clear, complete instructions about the task to be performed without previous output examples.

Role prompting, on the other hand, involves assigning the model a role before presenting the task to influence its response style and priorities. Beginning with "You are a software developer..." prompts the model to adopt the associated tone, vocabulary, and best practices. Thus, the output tends to be more technical, precise, and adapted to the desired profile.

The prompt used is as follows:

> *You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked*

*[**task**]. Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone, they are runned by an evaluator externaly as following: (inside run_tests/evaluate_coverage); exec(main_code.py); exec(test_code.py). Each, the tests and script classes must have its own main class as following: if __name__ == "__main__": main().*

After executing the prompt for each program-model combination, three artifacts are extracted: the solution, the test suite, and the reference code, if available.

It is important to know that the unit tests used in this framework were not provided externally or manually developed. They were generated directly by the LLMs in response to the same prompt. This approach enables the evaluation to simultaneously assess two critical aspects of the models, their ability to produce functional code, and their capacity to generate meaningful and complete test cases. This generation process provides a more realistic measure of their performance in autonomous coding scenarios.

3.1.3 Execution and collection of metrics

For the execution of the framework, a CodeEvaluator module has been created which runs through all the output directories with the code generated by the LLM and, for each file, generates a set of quality and functional metrics derived from the automatic execution of the tests. All these metrics generated by the framework are introduced in a CSV format file called *code_evaluation.csv* which will be used later to graphically evaluate the results.

3.2 Framework Flowchart

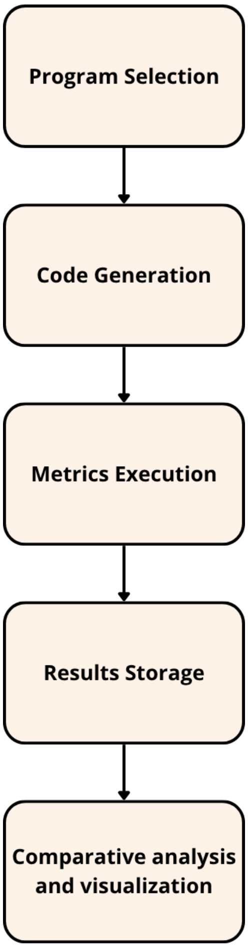

Figure 1: Framework Flowchart

3.3 Metrics for code assessment

This subsection presents a set of metrics. These metrics will be used to objectively and quantitatively assess the quality and functionality of the code generated by LLM models. Each metric provides a different perspective, ranging from functional accuracy to efficiency and maintainability. Together, they enable comparisons of solutions, detection of weaknesses, and guidance of improvements.

3.3.1 Correctness

Correctness is a metric that measures how well the generated code meets the requirements defined in the unit tests. To properly exploit this metric, it can be broken down into the following sub-metrics: Total Tests, Passes, Failures, Errors, and Pass Rate. These sub-metrics provide a series of results with which to analyze the functionality of the code generated by the different LLMs. Obtaining a value of 100% passed tests indicates that all basic functionalities and limit cases are correctly created. Each time a script produced by an LLM is evaluated, it is compared against a set of tests covering typical cases. We will use the following formula to calculate the pass rate:

$pass\_rate = \frac{\text{passed tests}}{\text{total tests}} \times 100$

If the result is greater than 90%, it is considered that the model has generated a robust solution. On the other hand, if the result is less than 70%, it can be said that aspects of the code could be improved.[19] [20]

3.3.2 Execution time

Execution time is a metric that measures how long it takes for the generated code to execute from start to finish. Termination can occur when the program finishes or when an exception is thrown. This metric directly measures algorithmic efficiency in terms of time and possible bottlenecks. The lower the execution time, the faster the LLM-generated solution responds.

Depending on the complexity and length of each program, different execution time ranges can be expected. For example, for a simple script such as an array search, a value less than 0.05 indicates that the program is highly efficient, while a value greater than 0.5 suggests excessive program complexity.[31]

3.3.3 Cyclomatic complexity

Cyclomatic complexity, a metric proposed by Thomas McCabe in 1976, measures the number of linearly independent paths through a block of code. It measures the number of linearly independent paths through a block of code. It primarily considers decisions (if, for, while, etc.) and their nesting depth. A higher degree of complexity directly affects execution time negatively.

Depending on the type of program generated, a different result will be expected. Typically, a value between one and five indicates a simple, easy to follow flow. However, when it exceeds ten or fifteen, the program becomes difficult to understand and test.

In the case of this framework, we are going to use Radon's `cc_visit` (imported from `radon.complexity`).[32]

3.3.4 Code length (LOC)

The lines of code (LOC) metric measures the number of lines occupied by code generated by the LLM. This metric measures three different aspects:

- Total number of lines: It usually records the total number of lines, including comments and blank lines.
- Non-empty lines: It usually records all lines containing text, including comments and code.
- Lines of executable code: It includes non-empty lines that are not comments.

This metric helps evaluate the conciseness of a solution. For example, a very long script for a simple program may indicate over-complexity. [27]

3.3.5 Memory usage (number of data variables)

This metric measures the number of variables declared in the code, including single assignments. Each assignment or declaration increases the value by one. Having a large number of variables can lead to increased memory consumption. In some cases, it may also result in a sparse layout of data rather than grouping it into structures.

When interpreting the results, the size and complexity of the script generated by the different LLM models must be considered. For example for a medium sized script a range of less than 20 variables would be considered a memory light program, while one with more than 50 variables would be considered high in memory which would mean that there is a possible data fragmentation or repetition of variables.

3.3.6 Number of methods

This metric counts the number of methods in the code generated by the LLM. A method is defined as a function within a class. This metric enables evaluation of the generated solution's degree of modularity and object orientation. A low value suggests monolithic classes that concentrate too much logic, while a high value may indicate over-engineering with too many small responsibilities distributed among multiple methods, which increases their complexity.

3.3.7 Comment-to-code ratio

This metric measures the proportion of lines devoted to comments against the total number of lines of code. An optimal level of comments for improved comprehension and maintainability is between 15% and 30%. A value of less than 10% may indicate poor documentation, which hinders comprehension and maintainability. A value greater than 50% may reflect an excess of unnecessary comments.

3.3.8 Test coverage

This metric quantifies the percentage of lines of code executed when running the unit test suite. It measures the extent to which our tests effectively cover all program paths and blocks. A value close to 100% means that nearly all of the code is being verified. When evaluating code generated by LLMs, a value greater than 80% indicates excellent coverage and high confidence in the robustness of the implementation. Coverage between 80% and 60% indicates adequate coverage with gaps that could be reinforced. A value below 60% means there are too many untested branches, leaving large areas unattended with a high risk of undetected bugs.

3.3.9 Code similarity

There are two approaches to this metric: cosine similarity and Jaccard similarity. These are two approaches commonly used in natural language processing (NLP) for comparing the lexical content of two or more texts.[24] [25]

- Cosine similarity measures the angle between two vectors in a high-dimensional space. Each vector represents the content of a text after applying TF-IDF (term frequency–inverse document frequency), a numerical statistic that reflects the importance of a word within a document relative to a collection of documents. A cosine value close to 1 indicates that the texts have very similar term patterns and lexical structures, while a value close to 0 indicates little to no similarity.

The ranges used to evaluate these metrics are as follows: A value greater than 0.8 indicates very high similarity with respect to the reference code. A value between 0.5 and 0.8 indicates some similarity with notable variations. A value less than 0.5 indicates low similarity.

- Jaccard similarity compares two sets of elements by measuring the ratio of the intersection to the union. A value of 1 means the sets are identical, and a value of 0 means the sets have no tokens in common.

3.3.10 AST Similarity

Abstract Syntax Tree (AST) similarity measures how similar the syntactic structure of the generated code is to the reference code.

AST or Abstract Syntax Tree is a hierarchical representation of the source code where each node reflects a language construct, such as statements, expressions or operations, and their logical relationships. In Python it is obtained with ast.parse(), traversed or transformed with ast.walk() or NodeVisitor, and allows to analyze the program structure or compare its form with another implementation in a name- and style-independent way. [16]

The result obtained from AST is between 0 and 1. If the value is close to 1, it means that the more similar the code structure is. On the other hand, a value close to 0 indicates that the syntactic organization differs significantly. Intermediate ratios, such as a value of 0.5, indicate that there is some overlap in key components but also differences in program form.

3.3.11 Dependency complexity

This metric measures how many import statements the code contains, for example, how many models, external packages or libraries it needs to execute. A high number of dependencies can complicate the complexity of the script and increase the chances of failures and make the traceability of changes difficult at the time of maintenance. [30] In the case of this study a value of less than 5 imports means that the solution does not use a lot of libraries making the script easy to deploy and maintain. If the value is between 5 and 10 imports it is said that there is a reasonable balance. And if the value is over 10 imports it means that there is a risk of external dependencies that can cause compatibility issues.

3.3.12 Redundant code ratio

This metric measures the proportion of repeated lines within the set of significant lines in the script—that is, lines containing code. Each non-empty line is compared with the others; those that appear more than once are considered duplicates. A value near 0 indicates minimal or no repetition. As this value increases, it indicates an increase in duplicated parts. [29]

In this study, a value less than 0.1 indicates almost no duplicate lines. A value between 0.1 and 0.2 indicates some duplication and suggests refactoring. Finally, if the value is greater than 0.2, then there are many duplicate lines.

3.3.13 Style compliance (PEP8 adherence)

PEP 8 stands for Python Enhancement Proposal 8. Published in 2001, it is a standard coding style guide for the Python language. This style guideline is recognized worldwide for helping to write high-quality, readable, and maintainable Python code. The guide is still maintained and updated.[23]

This metric measures the number of violations of the PEP8 style guide in the code. These errors can cover a wide range of style characteristics, such as lines that are too long, incorrect indentation, incorrect spacing, misspelled names, and so on. Each error indicates a deviation from conventions and standards recognized by the community. Code without style errors is more uniform, readable, and easier to maintain.

In this study, a score of 0 to 5 errors indicates that the code is nearly fully compliant with style standards. A value between 6 and 20 indicates that minor details could be refined. If the score is greater than 20, then the code should be revised due to multiple styling errors.

For this evaluation `pycodestyle.StyleGuide` library is used.

3.3.14 Code reusability and maintainability index

This metric returns as a result a numerical value between 0 and 100. It combines several code factors, such as size, cyclomatic complexity, and comments, to estimate how easy it is to understand or modify a program. A higher Manageability Index implies cleaner, more modular code and usually correlates with higher component reusability. [28]

A program will be considered excellent in terms of manageability if the value is greater than 85. A result between 65 and 85 indicates reasonably structured code with room for improvement. Finally, if the value is less than 65, there are signs of high coupling, complex logic, and difficulty refactoring or reusing the code.

4 Benchmarking System

4.1 Design of the scoring system

This subsection describes how we constructed the mechanism that transforms raw metric values into comparable, easily interpretable composite scores for each model.

1. Normalization of Metrics

   The raw metrics (x) have different units and ranges. Some range from 0 to infinity (e.g., runtime), while others range from 0 to 100 (e.g., maintainability). To compare them, we are going to apply min-max scaling, which rescales them all to the interval [0, 1].

   We define:

   $$x' = \begin{cases} \dfrac{x - x_{\min}}{x_{\max} - x_{\min}}, & \text{if higher is better,} \\ \dfrac{x_{\max} - x}{x_{\max} - x_{\min}}, & \text{if lower is better.} \end{cases} \tag{1}$$

   This normalization ensures that the best observed value is assigned a value of 1, while the worst is assigned a value of 0. This maintains proportionality for intermediate values.

2. Calculation of scores by category

   After normalizing each metric m to m', we aggregate two sets of scores:

   - Score_quality (Quality of the code)

   $$\text{Score}_{\text{quality}} = \sum_i w_{q_i}\, q'_i, \quad \sum_i w_{q_i} = 1 \tag{2}$$

   - Score_functional (Functionality)

   $$\text{Score}_{\text{functional}} = \sum_j w_{f_j}\, f'_j, \quad \sum_j w_{f_j} = 1 \tag{3}$$

   Each q_i is the normalized value of quality metric i in the range $[0, 1]$. w_{q_i} is the weight assigned to that metric.

   In the case of functional score, f'_j represents the normalized functional metric j, and w_{f_j} its corresponding weight. Thus, the functionality score is produced. These variables allow us to control the influence of each metric before combining the quality and functionality scores into the overall score.

3. Global Score

   The overall score balances the two main components, quality and functionality, to generate a single value that allows for direct comparison of different models. The following formula is used for this purpose:

$$\text{Score}_{\text{global}} = 100 \times \Big(\alpha\,\text{Score}_{\text{quality}} + (1-\alpha)\,\text{Score}_{\text{functional}}\Big)$$

   This formula calculates a weighted average of the two category scores and scales to a range of 0-100:

   The parameter $\alpha \in [0,\ 1]$ represents the relative importance of quality versus functionality. I'll use $\alpha = 0.5$ by default to give them equal weight.

4.2 Weight assignment mechanism for metrics

In the designed benchmark, each metric receives a weight w that reflects its relevance according to software quality standards and academic evidence. To set the quality weights we have relied on ISO/IEC 25010, where maintainability and its sub-characteristics, such as cyclomatic complexity, duplication, documentation or PEP8 compliance, have a prominent role to calibrate the influence of complexity and comments. Similarity metrics, such as AST, Cosine and Jaccard, come from code evaluation frameworks such as CodeBLEU and recent studies validate their ability to measure structural and semantic fidelity.

For functional weights, ISO/IEC/IEEE 29119 has been followed, which prioritizes functional correctness and test coverage as key risk mitigation mechanisms, assigning them the highest percentages. Execution efficiency is based on the performance efficiency feature of ISO/IEC 25010, and the number of calls acts as a proxy for modular coupling, also covered in that standard.

With this approach, each weight is not arbitrary but emanates from recognized normative frameworks and empirical references.

The weights assigned to each metric are as follows:

Table 1: Code Quality (`quality_weights`)

Metric	**Weight**	**Description**
`style_errors`	0.10	Penalizes PEP8 style violations without over-penalizing minor infractions.
`total_cc`	0.10	Reflects the overall cyclomatic complexity of the control flow.
`maintainability_index`	0.15	Synthesizes readability, modularity, and documentation quality.
`redundant_code_ratio`	0.10	Penalizes duplicated lines of code.
`comment_ratio`	0.10	Encourages balanced and useful documentation.
`cosine_similarity`	0.10	Measures lexical similarity to the reference implementation.
`jaccard_similarity`	0.10	Measures token-level overlap with the reference implementation.
`ast_similarity`	0.20	Reinforces adherence to the expected code structure; most heavily weighted.

Table 2: Functionality (`functional_weights`)

Metric	**Weight**	**Description**
`pass_rate`	0.40	Correctness is critical, hence the highest weight.
`coverage_percentage`	0.30	Ensures that tests cover the majority of code paths.
`execution_time`	0.20	Performance matters, though secondary to correctness.
`method_calls`	0.10	Proxy for coupling and modular use of helper calls.

4.3 User-defined customization of evaluation criteria

The relevance of certain metrics depends on the specific purpose of the generated code.

For example:

- In the case of educational tools designed to teach good programming practices, we could emphasize metrics such as the feedback ratio and maintainability index to reward well-documented and modularly structured solutions.

- For production applications, where reliability and performance in a real environment are priorities, it would make sense to reinforce metrics such as pass rate, test coverage, and execution time. This would ensure that the code is robust and efficient.

These adaptations can be easily made by modifying the quality_weights or functional_weights dictionaries, or the mixing factor α in the benchmark script.

However, to ensure a fair and comparable study, I will use the weights defined in the previous section. This ensures that all models are evaluated under the same criteria, and that the results reflect only differences in code quality and functionality without introducing bias through weight adjustments.

5 Experimental Setup

In this section, we will provide a detailed description of the environment, data, and general project flow for evaluating LLMs for code generation. To ensure reproducibility and clarity, we will first detail the selected models and their configuration, and finally the complete execution and analysis methodology.

5.1 Selection of LLM models for evaluation

For this evaluation, I chose four of the most advanced and popular LLMs as of the project's development date. These LLMs cover different sizes, architectures, and philosophies of language generation and processing. Their key features and strengths are described below.

GPT-4o, also known as GPT4 Omni, is OpenAI's multimodal model that was launched in May 2024. Trained on text, images, and audio, it can process and generate coherent responses in all three formats using the same neural network. GPT-4o has demonstrated substantial advancements in multimodal comprehension benchmarks and general reasoning tasks, surpassing GPT-4 Turbo in latency and cost per token.

- Native multimodality: Text, images, and audio are processed by a single model.
- Context length: 128k tokens, ideal for very long prompts
- Performance: 2 times faster and 50% cheaper than GPT-4 Turbo
- Use cases: complex code generation, image annotation, and real-time audio processing.

GPTo4-mini-high is a *mini* version of GPT-4o launched in April 2025, that enables high concurrency at a low cost. It retains much of GPT-4o's reasoning and multi-modality capabilities, but with a significantly smaller number of parameters. This reduces latency and cost per token without sacrificing task scheduling quality.

- Reduced size: It has a small size of approximately 8 billion parameters, similar to LLaMA 3 with also 8B parameters.
- Speed: Intended for fast responses in large-scale applications.
- Advanced context length: 200k tokens vs. 128k in GPT-4o, with a maximum output of 100k tokens.
- Strengths: Fast and efficient reasoning. Excellent for coding and visual tasks.

Google's **Gemini 2.5** is the latest version of Gemini, launched in March 2025. It is optimized for reasoning and multimodality, and it can process up to 1 million tokens of context. It incorporates reasoning techniques and can natively process text, audio, images, and videos. It excels at analyzing large code repositories and documents.

- Context window: 1,000,000 tokens, enabling the analysis of entire projects.
- Multimodality: Supports text, audio, images, and videos with a single endpoint.

- Integrated reasoning: "Deep Think" mode provides a step-by-step breakdown of the problem.
- Use cases include generation of complex algorithms, interactive simulations, and analysis of massive datasets. There are also improvements in advanced math and science tasks.

Grok 3, the third model created by xAI, launched on February 2025, was trained with 10 times more computing power than Grok 2 on the Colossus supercomputer. It uses large-scale reinforcement learning to improve reasoning in two modes: "Think," which shows the chain of thought, and "Big Brain," which is used for highly complex problems. Additionally, Grok 3 integrates DeepSearch, an agent that can perform real-time internet and social media searches on X,formerly known as Twitter, to validate facts and enrich answers with the users interaction in this social network.

- Compute power: 10x GPUs in Colossus vs. Grok 2
- Reasoning modes: "Think" (exposes chain of thought) and "Big Brain" (backtracking and refinement).
- DeepSearch: searches and synthesizes live information from the web for up-to-the-minute responses.
- Use cases: include interactive debugging, test generation, and real-time information retrieval.

In terms of quality versus cost, an overall comparison of the four selected models shows that GPT-4o and Gemini 2.5, in general terms, tend to be at the high end of the spectrum. This is thanks to their reasoning power and consistency in code generation, albeit at the cost of higher latency and cost per token. GPTo4-mini-high and Grok 3 are designed to provide faster responses and lower execution cost, sacrificing only a small part of the fidelity in cases of high complexity.

Regarding the handling of extreme contexts, which require the models' maximum potential, Gemini 2.5 is the leader. It has a window of up to one million tokens, allowing it to process large projects or documents without partitioning the prompt. GPTo4-mini-high follows with a limit of 200,000 tokens, and GPT-4o follows with 128,000. While not reaching those numbers, Grok 3 maintains a high limit intended for multi-shift conversational chats and real-time searches.

Each model has its own area of expertise.

- GPT-4o, for example, is highly effective in high-level multimodal projects involving text, images, and audio.
- GPTo4-mini-high integrates perfectly into massive pipelines that require cost efficiency and very low latencies.
- Gemini 2.5 excels at analyzing large amounts of code and documents thanks to its specialized training and extensive context window.

- Grok 3 is excellent at interactive debugging, step-by-step reasoning, and using its DeepSearch agent to obtain real-time data.

These four models were chosen because they are the most current and widely used by the developer community as of the project date. They all have public APIs, support a variety of use cases, and receive constant updates from their vendors.

5.2 Methodology for running and analyzing results

To ensure a reproducible and systematic workflow, the execution and analysis process is divided into three phases: data generation, preprocessing, and visualization and statistical analysis.

5.2.1 Data generation

As described in Subsection 3.1, data generation is based on the same prompt design, which, as mentioned above, follows zero-shot instructions with role-prompting patterns. This prompt was manually applied to each program-model combination. Three artifacts were extracted from each run: the solution script, the unit tests, and the reference code when available. This ensured that all models started from the same instruction and parameters.

Once the code has been generated by the different models, a series of files containing the reference code, the generated code, and the unit tests is created. To correctly differentiate each program and model used, the name of each file contains the program name and the model used.

After preparing all the files, the script that evaluates the metrics explained in previous sections is executed, including the execution of the unit tests.

These values and the metrics are concatenated in a CSV format file called code_evaluation.csv.

5.2.2 Data preprocessing

Once the code_evaluation.csv file has been generated correctly, a brief data cleaning and transformation process is performed to ensure the data is in the desired format. For example, all metric columns are converted to a numeric type, which forces invalid and empty values to NaN. This ensures that subsequent calculations and plot generation use consistent data.

5.2.3 Visualization and statistical analysis

After cleaning the DataFrame, the script generates a series of graphs for each program and category of metrics (quality and functionality). Each group shows only the corresponding metrics. Additionally, we perform two levels of analysis: an individual analysis of each program with its bar plots and a global analysis that gathers all the execution data and presents comparative box plots by model. We also conduct a global analysis that compiles all execution data and presents comparative box plots by model.

To interactively integrate all generated graphs and facilitate visualization, we have also developed a Streamlit dashboard. The dashboard loads the generated plots directly and organizes them in tabs by category (Quality and Functionality) and by program, including a global ”All” summary. The dashboard allows for dynamic navigation between models and metrics, configurable display size, and scrolling and zooming within each interactive graph. These features facilitate comparative analysis, eliminating the need to manually open each file in the file system.

6 Results and Analysis

In this section, the results of the LLM model evaluation are presented in four sections. In Subsection 6.1, we conduct a comparative analysis of the code generated by each model using quality and style metrics. In Subsection 6.2, we examine functional performance through unit tests, detailing success rates, failure types, and edge cases. In Subsection 6.3, we explore the relationships between the obtained metrics to identify connections between code quality and performance. Finally, in Subsection 6.4, we unify the results by means of a global index generated by the created framework, offering conclusions.

6.1 Quality analysis of LLM-generated code

6.1.1 AST Similarity

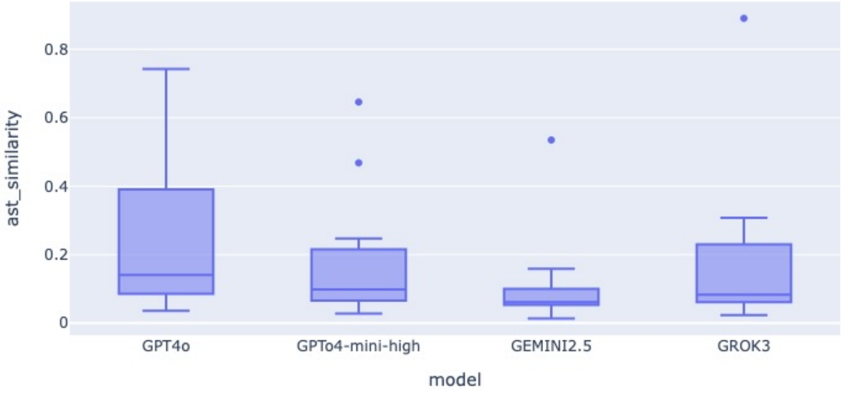

Figure 2: Overall AST Similarity Results

In the AST Similarity Analysis, GPT4o exhibits the greatest variability, with a median of approximately 0.15, an interquartile range (IQR) between 0.08 and 0.40, and whiskers extending to 0.75. This indicates that GPT4o sometimes copies the structure almost literally and sometimes rewrites it completely. GPTo4-mini-high has an aproximate median of 0.10, an IQR between 0.05 and 0.22, and whiskers with values below 0.25. It also has two outliers of about 0.45 and 0.65. It preserves a similar skeleton without extreme copying.

On the other hand, GEMINI 2.5 has the lowest median, of about 0.08 and a IQR between 0.03 and 0.10. It almost always restructures syntax and prioritizes originality. GROK3 has a median and IQR similar to GPTo4-mini-high, but it has a high outlier value of 0.9. GROK3 combines moderate consistency with some cases of nearly exact reproduction.

In summary, GPT4o and Grok3 can achieve high structural fidelity irregularly. GPTo4-mini-high offers the most reliable balance, while GEMINI 2.5 is the most creative option, albeit with lower AST similarity.

6.1.2 Total Cyclomatic Complexity

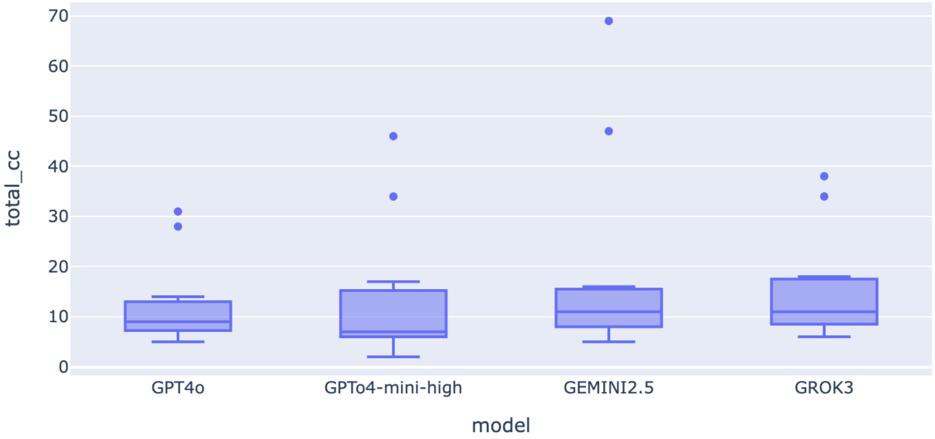

Figure 3: Overall Total Cyclomatic Complexity Results

In the cyclomatic complexity analysis, GPT4o has the lowest median and the most compact distribution, having an IQR between 7 and 13 and whiskers between 5 and 14, with a couple of outliers. This indicates that, consistently, GPT4o generates code with relatively controlled logical complexity. GPTo4-mini-high, despite having a similar median as GPT4o, shows a much larger IQR with values between 6 and 15 and whiskers ranging between 2 and 17, as well as very high outliers. This suggests that in certain cases it produces significantly more complex structures.

For its part, GEMINI 2.5 has a higher median with a value of 11 and maintains an intermediate IQR and with whiskers up to 16, but has outliers that reach very high values such as 47 and 69, reflecting occasions in which the logic becomes noticeably more complicated. GROK3, also with a median of 11 and a wider IQR and whiskers between 6 and 17, also adds slightly smaller outliers that show a similar pattern: moderate complexity in most examples, with occasional examples of high intricacies.

Overall, if one seeks to minimize cyclomatic complexity consistently, GPT4o is the most reliable choice. In contrast GPTo4-mini-high introduces more scatter and GEMINI 2.5 and GROK3 can lead to high complexity peaks in specific cases.

6.1.3 Comment Ratio

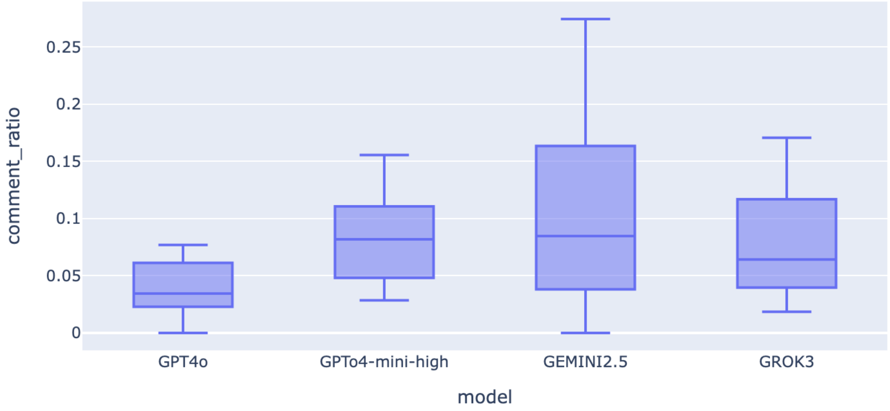

Figure 4: Overall Comment Ratio Results

In the comment ratio analysis, GPT4o shows the lowest median with a value of 0.03 and a contained dispersion, with an IQR between 0.02 and 0.06 and whiskers between 0 and 0.075, indicating that, quite consistently that it generates code with very few comments. In contrast, GPT4o-mini-high increases that documentary level to a median of 0.08 and exhibits a moderate IQR with whiskers reaching 0.15, reflecting a more stable comment ratio.

GEMINI 2.5, on the other hand, exhibits the greatest variability, with a median of 0.085, wide IQR and whiskers reaching almost 0.27. This points to results ranging from almost uncommented fragments to highly commented ones. Finally, GROK3 falls in between with a median of 0.065, intermediate IQR and whiskers up to 0.17, with some outliers.

In summary, if you are looking to maximize documentation, GEMINI 2.5 is the choice. For medium level and reliability, GPT4o-mini-high is more balanced. And GPT4o produces the most concise code in comments.

6.1.4 Cosine Similarity

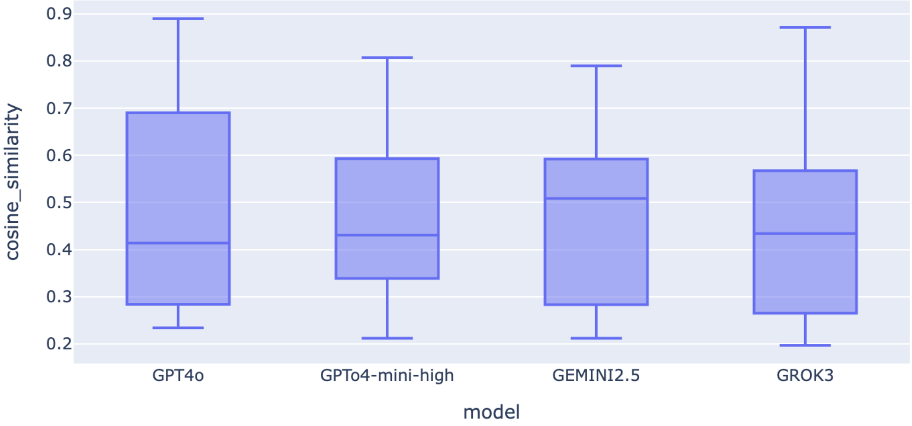

Figure 5: Overall Cosine Similarity Results

In the cosine similarity analysis, GPT4o has a median of 0.41. The IQR values are between 0.29 and 0.68 and the whiskers are between 0.24 to 0.89. This shows cases of low similarity as well as very high peaks. On the other hand, GPT4o-mini-high has a higher median of 0.43 and a narrower IQR of values between 0.21 and 0.81. This means greater consistency in semantic similarity. GEMINI 2.5 stands out with the highest median, at 0.51, and an intermediate IQR. Its whiskers reach 0.79, suggesting that much of its output shares semantics closer to the reference code. GROK3 shows a median similar to GPT4o-mini-high, with an IQR between 0.26 and 0.57 and longer whiskers. This means that some generated code has high levels of semantic similarity.

In summary, GEMINI 2.5 is the strongest choice if the goal is to maximize semantic similarity. For lower levels of similarity, GPT4o-mini-high is a better alternative. If more variability is tolerated in exchange for possible extreme similarity peaks, GPT4o or GROK3 may be suitable in specific cases.

6.1.5 Jaccard Similarity

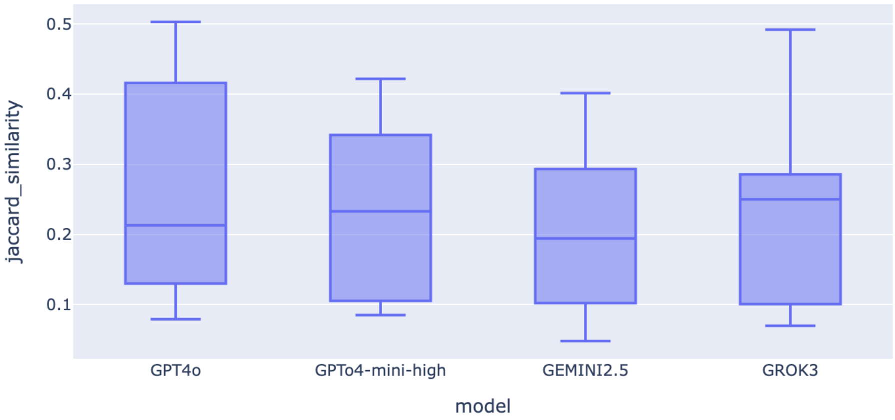

Figure 6: Overall Jaccard Similarity Results

In the Jaccard Similarity analysis, GPT4o shows a median close to 0.22 with an IQR with values between 0.13 and 0.41 and whiskers ranging from 0.08 to 0.50, which shows a high variability. This is because in some cases the token overlap is very low, and in others it reaches values close to 0.5. GPT4o-mini-high, on the other hand, offers a somewhat higher median value of 0.23 but a more contained IQR and whiskers between 0.085 and 0.42 approximately, showing a more stable lexical similarity. GEMINI 2.5 exhibits the lowest median with a value of 0.19 and a moderate IQR with whiskers between 0.05 and 0.40, indicating somewhat less overall overlap. GROK3 exhibits the highest median with IQR around 0.10 and 0.28 and whiskers between 0.07 and 0.49. This achieves more overlap but also with cases of extreme values.

In summary, if token to to token overlap is to be maximized, GROK3 is the most reliable option. On the other hand, if very high peaks are tolerated in exchange for irregularity, GPT4o can give the maximum values. For consistent similarity, GPT4o-mini-high is more balanced. As for GEMINI 2.5, it sacrifices some Jaccard to diversify the implementation.

6.1.6 Maintainability Index

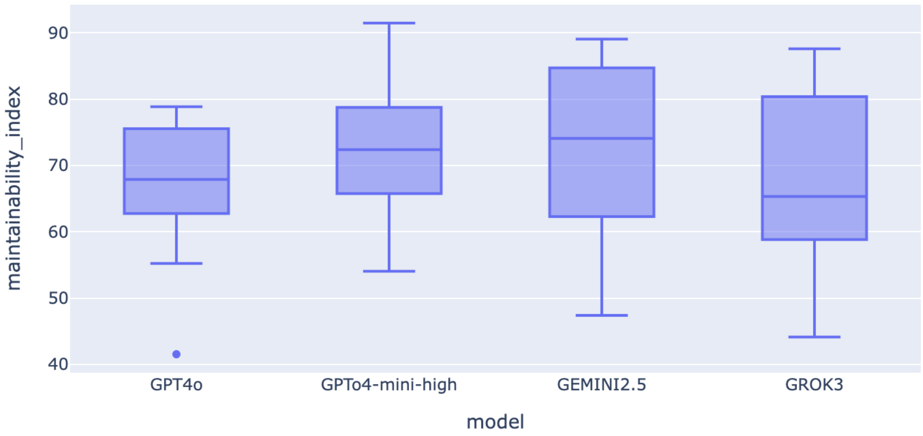

Figure 7: Overall Maintainability Index Results

In the Maintainability Index analysis, GPT4o shows a median around 68, with a narrow IQR with values of 63 and 75 and whiskers varying between 55 and 79, along with an outlier around 42. This indicates that it generates code with a decent level of maintainability and few extreme outliers. GPTo4-mini-high, on the other hand, has a higher median with a value of 72 and maintains a moderate IQR, although its whiskers are more extended with values between 54 and 92, suggesting that it produces very maintainable code, but occasionally there are cases with very high outliers. GEMINI 2.5 has the highest median value of 74, but with a wide IQR and whiskers between 47 and 89, reflecting variability. This means that it often delivers very maintainable code, although there are also examples where it drops below 50. GROK3, on the other hand, is at a lower median with a value of 65 and with a wide IQR and whiskers from 44 to 88, showing both cases of good maintainability and more complex examples to refactor.

Overall, if the goal is to maximize maintainability, GEMINI 2.5 is the most powerful although less predictable. On the other hand, for high and relatively constant performance, GPT4-mini-high offers the best balance. And if outliers are to be avoided, GPT4o is the most stable option.

6.1.7 Redundant Code Ratio

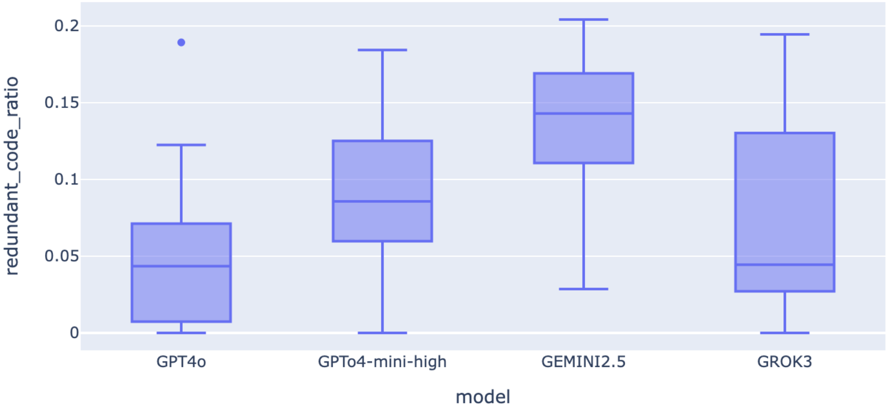

Figure 8: Overall Redundant Code Ratio Results

In the redundant code ratio analysis, GPT4o has the lowest median with a value of 0.045, a low variable IQR and whiskers ranging from 0 to 0.12, with a single isolated outlier around 0.19. This indicates that, consistently, GPT4o generates very little logic duplication or repeated fragments. GPTo4-mini-high, on the other hand, raises its median to 0.085 and shows a wider IQR, with values between 0.06 and 0.12, with whiskers reaching 0.18, suggesting that it tends to introduce somewhat more redundant code on a stable basis, although without too frequent extremes. For its part, GEMINI 2.5 achieves the highest median than the previous models with an IQR around 0.11 and 0.17 and whiskers ranging from about 0.03 to 0.20, reflecting a pattern of greater repetition of code blocks but also some variability. GROK3 offers a low median, similar to GPT4o, but with a much wider IQR with values between 0.025 and 0.13 and whiskers ranging up to 0.19, indicating very unpredictable outputs. In some cases they are as clean as GPT4o, others with redundancies.

In summary, if the priority is to minimize duplication, GPT4o is the most reliable choice. On the other hand, for a moderate and predictable level, GPT4-mini-high is more suitable. If variability is to be accepted, both GEMINI 2.5 and GROK3 give the greatest diversity of profiles.

6.1.8 Style Errors

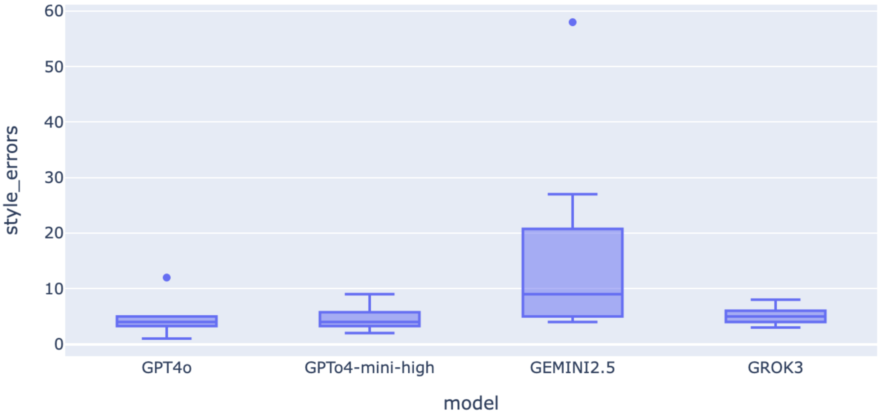

Figure 9: Overall Style Errors Results

In the analysis of style errors, GPT4o shows a low median of around four or five errors, with a narrow IQR of values between three and five and whiskers ranging from one to six, plus an isolated outlier of 12. This indicates that its output consistently meets style standards with few deviations. GPTo4-mini-high exhibits similar behavior, with a median of 4, a somewhat wider IQR ranging from 3 to 6, and whiskers extending from 2 to 9. It introduces few errors in general, but occasionally incurs somewhat higher style violations. In contrast, GEMINI 2.5 exhibits the greatest variability, with a median of nine, a substantial IQR, and whiskers extending up to 27, along with an extreme outlier approaching 58. GROK3 falls between the other models with a median of 5, a moderate IQR, and whiskers between 3 and 8. There are no extreme outliers, suggesting a reasonable level of style consistency.

In summary, GPT4o and GPTo4-mini-high are the most reliable choices if one seeks to minimize style violations. GROK3, on the other hand, offers a stable balance. GEMINI 2.5, despite its creative potential, requires style reinforcement.

6.2 Functional analysis from unit test performance

6.2.1 Execution Time

Figure 10: Overall Execution Time Results

In the execution time analysis, GPT4o has a median of about three seconds, an IQR between one and six seconds, and whiskers reaching up to 14 seconds. There are no outliers, indicating that all executions consistently stay within that range. GPTo4-mini-high, has a median of about two seconds, an IQR between zero and six seconds, and whiskers up to eight seconds. However, it incorporates two outliers around 17 and 18 seconds, indicating occasionally slower runs.

GEMINI 2.5 has the lowest median with 1 second, it also has a narrow IQR and whiskers of 4 seconds. However, it has an outlier close to 18 seconds. GROK3 has the lower median with a value 0.8 seconds. This means that is the fastest one. It has an IQR between 0 and 3 seconds, and whiskers up to 4 seconds. However, it also shows a significant outlier. These results suggest that these three models they may occasionally run slowly.

In summary, if the code is needed to ensure that execution time always stays within a controlled range, GPT4o is the most reliable choice. If the lowest median time is desired, with the possibility of isolated delays, then GEMINI 2.5 and GROK3 are the fastest. For a balance between speed and moderate dispersion, GPTo4-mini-high is a valid option.

6.2.2 Methods Calls

Figure 11: Overall Method Calls Results

In the Method Calls analysis, GPT4o has a median around 14 calls, an IQR of approximately 12 to 20, whiskers ranging from 6 to 20 calls, and two outliers around 30 and 31 calls. This indicates that, in most cases, GPT4o maintains a reasonable use of invocations, but in a couple of examples it generates more methods, which can be seen directly related to the complexity of the requested program. GPTo4-mini-high has a slightly higher median of about 15 calls, a narrower IQR, whiskers covering 6 to 18 calls, and also two prominent outliers of about 29 and 43 calls, revealing fairly predictable behavior except for isolated moments of excess.

In contrast, GEMINI 2.5 raises its median to 16 calls but exhibits a very wide IQR and whiskers reaching up to 51 calls, with no outliers marked outside that range. This reflects that it can produce both very contained solutions and slide toward an excessive number of calls. GROK3, with a median of 14 and whiskers ranging from 6 to 40 calls, shows a similarly prominent dispersion, although with no peaks outside its whiskers.

In summary, if the priority is to maintain a stable and controlled number of invocations, GPT4o and GPTo4-mini-high are the most reliable options. On the other hand, if structural flexibility is sought and wide variations in call flow length are less important, GEMINI 2.5 and GROK3 offer the widest range.

6.2.3 Coverage

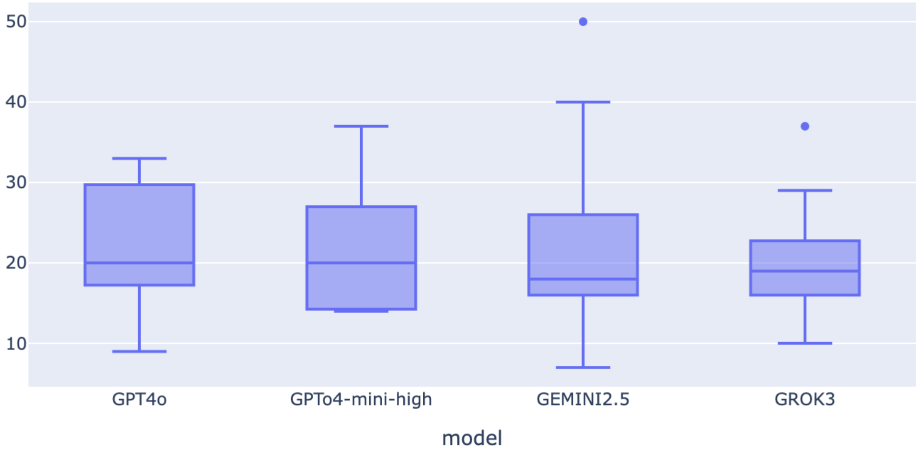

Figure 12: Overall Coverage Percentage Results

In the Coverage Percentage analysis, GPT4o shows a median close to 20%, with an IQR of approximately 17 to 30% and whiskers ranging from 9% to 33%. It shows no outliers outside this range, indicating very consistent behavior. GPTo4-mini-high, on the other hand, also has a median around 20%, a somewhat narrower IQR and whiskers ranging up to 37%.

As for the other models, GEMINI 2.5 has a slightly lower median value of 18%, with a moderate IQR ranging from 16 to 26% and whiskers ranging from 7% to 40%, but includes an outlier of around 50%, reflecting that, although it usually provides reasonable coverage, in one particular case it triggered the tests noticeably. GROK3 has a median close to 19%, an IQR around 16 and 23%, whiskers ranging from 10% to 29% and an outlier around 37%, showing a similar dispersion to GPTo4-mini-high but with a punctually higher ceiling.

In summary, if maximum predictability in the percentage of coverage is sought, GPT4o is the most stable option. If the objective is to achieve occasional very high coverages, GEMINI 2.5 and GPT4-mini-high offer such peaks, while GROK3 balances a mid-range with the ability to exceed 30% in isolated cases.

6.2.4 Pass Rate

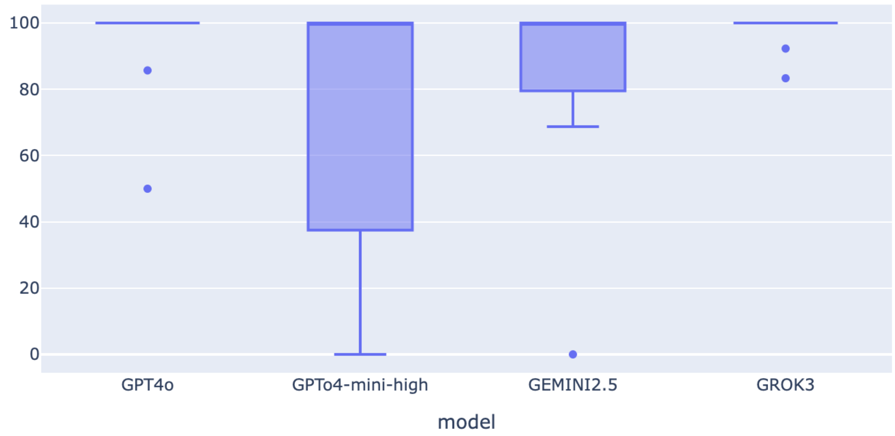

Figure 13: Overall Pass Rate Results

In the Pass Rate analysis, GPT4o achieves a median of 100% with an IQR of almost 100% and whiskers that remain at 100%, with no outlier within that range, indicating that in the vast majority of cases it passes all tests. The only two outliers of note are around 85% and 50%, reflecting a couple of runs with partial failures. GPTo4-mini-high, in contrast, shows the greatest dispersion with its median hovering around 70%, the IQR ranging between 37 and 93%, and the whiskers cover from 0%, indicating total failure, to around 98%. This is evidence that, although it sometimes passes all or almost all tests, in other examples it fails completely.

For its part, GEMINI 2.5 offers a very high median value of 98%, an intermediate IQR between 80 and 100% and whiskers between 68% and 100%, but includes an isolated outlier at 0%, indicating a single case of total failure. GROK3, similar to GPT4o, maintains IQR and whiskers at 100%, with a median of 100%, and only two moderate outliers at 83% and 93%, reflecting very occasional drops.

In summary, if you need to guarantee full passage of the tests consistently, GPT4o and GROK3 are the most reliable options. GEMINI 2.5 combines high median with a single extreme failure, while GPTo4-mini-high is the least predictable, varying between 0% and 98% pass rate.

6.3 Correlation between different evaluation metrics

6.3.1 Methodology

We considered each metric as a set of N observations and calculated its Pearson correlation coefficient [38] using `df_prog[metrics].corr()`, retaining only those associations with $p < 0.05$. When we detected that any metric deviated from normality, we repeated the calculation with Spearman's method (`corr(method='spearman')`), to capture monotonic relationships. In this way, the statistical description exactly matches the code fragment and the generated heatmap.

6.3.2 Correlation Matrix

Figure 14 shows the resulting heat map. Each cell indicates the value of r, where red shades indicate strong negative correlations and blue shades indicate positive correlations.

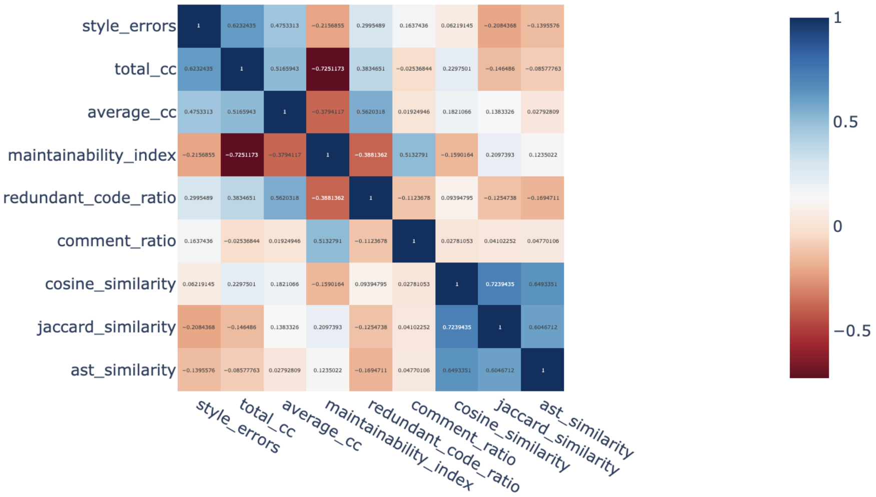

Figure 14: Quality Metrics Correlation Matrix

6.3.3 Main Findings

The cyclomatic complexity indicator, `total_cc`, show a strong negative correlation with `maintainability_index`, with a value of $r = -0.72$, confirming that the higher the number of logical branches, the lower the maintainability of the code. In addition, the `redundantcode_code_ratio` correlates negatively with maintainability with a value of $r = -0.39$, evidencing that the excess of duplicated blocks impairs maintainability.

On the other hand, comment density, known as `comment_ratio` and `maintainability_index` exhibit a moderate positive correlation, with a value of $r = +0.15$, indicating that a higher level of documentation tends to be accompanied by more maintainable code. At the opposite extreme, `style errors` correlate negatively with maintainability, with a value of $r = -0.12$ and positively with complexity with a value of $r = +0.42$, suggesting that well ordered code is also more compliant with style guidelines.

Finally, lexical and semantic similarity metrics turn out to be closely related. `cosine_similarity` and `jaccard_similarity` are related with a value $r = +0.73$, while both have moderate correlations with `ast_similarity` with a value of $r = +0.55$, implying that they share much of the variance when measuring overlap with the reference code.

6.4 Global Composite Score and Ranking

Based on the composite index defined in Section 4, Table 3 presents the aggregate quality, functionality and global scores for each model. Models are ranked according to their global score, which combines static analysis performance (quality) and test execution (functional) with the chosen weighting scheme.

Model	**Quality Score**	**Functional Score**	**Global Score**
GPT4o	0.7333	0.7366	73.4955
GPTo4-mini-high	0.7281	0.2830	50.5516
GEMINI2.5	0.3061	0.6980	50.2032
GROK3	0.5054	0.4563	48.0873

Table 3: Composite quality, functional and global scores for each LLM

As can be seen, GPT4o leads the ranking with an overall score of 73.50, driven by its balanced performance in both quality, with a score of 0.7333, and functionality with a value of 0.7366. GPTo4-mini-high ranked second with 50.55, thanks to its high static analysis score of 0.7281 despite its lower functional score with a value of 0.2830. GEMINI2.5 and GROK3 follow closely with overall scores of 50.20 and 48.09, respectively. In the case of GEMINI2.5, it is seen to obtain good test scores with a score of 0.6980, but a lower quality with a value of 0.3061, while GROK3 shows a moderate balance in both dimensions as it has a score of 0.5054 in quality and 0.4563 in functionality.

These results highlight the trade-offs inherent in composite ranking. A model with very high quality metrics may be penalized by weak functional performance, and vice versa. Therefore, the chosen weighting scheme plays a critical role in determining which LLM best matches the desired balance between code correctness, readability and execution reliability.

7 Conclusions

This final section presents a global reflection on the results obtained and the implications derived from the evaluation of code generated by large language models. This section is organized in five different subsections. The first subsection summarizes the main findings of the study, highlighting the different performances of the analyzed models. Subsection 7.2 talk about the methodological and practical limitations that may affect the results. Subsection 7.3 discusses the implications of these findings in software development. Subsection 7.4 mention possible future directions for improving the quality and reliability of code generated using different LLMs. Finally, Subsection 7.5 shows some recommendations which objective is guiding further research in this evolving field.

7.1 Summary of key findings

The study has allowed us to study and identify significant differences in the quality and functionality of the code generated by the different language models analyzed. In global terms, GPT-4o is positioned as the most balanced model, obtaining the highest score in both quality metrics and functional performance. Its code presents low cyclomatic complexity, low redundancy, good compliance with PEP8 style and a high success rate in unit tests.

On the other hand, GPTo4-mini-high has shown remarkable performance in static analysis metrics, such as structural similarity, also known as AST, and maintainability, although with a high variability in the hit rate in functional tests, which limits its practical consistency.

Gemini 2.5 stands out for its ability to generate highly semantic code with good levels of test coverage and success rate, but with marked weaknesses in structural aspects such as code duplication and the number of style errors, which affects its overall score. In contrast, Grok 3 is characterized by intermediate performance in most metrics, without particularly standing out in any aspect, although it maintains acceptable functional robustness.

Correlation analysis between metrics further reveals significant patterns. On the one hand, a strong negative relationship between cyclomatic complexity and maintainability, as well as a positive association between comment density and code quality. A high consistency between lexical and structural similarity metrics is also confirmed, validating their joint use as indicators of fidelity with respect to reference solutions.

These results not only validate the usefulness of multi-metric evaluation frameworks, but also highlight the importance of tailoring model selection to the specific requirements of a coding task.

Taken together, these findings demonstrate that, although current LLM models can generate functional and, in many cases, high quality code, there are substantial differences in their behavior depending on the type of task, the relationship between quality and performance, and consistency in test execution. Consequently, choosing the most suitable model for code generation should not rely just on global rankings, but also consider the balance between structural quality, execution reliability, and documentation needs depending on the intended application.

7.2 Limitations of the study

Although this study provides a comprehensive evaluation of the code generated by language models, there are several limitations that must be taken into account to properly contextualize the results. The 4 models analyzed, GPT-4o, GPTo4-mini-high, Gemini 2.5 and Grok 3, are generalist models, not designed exclusively for code generation. It is true that they are capable of producing good quality functional scripts and unit tests of considered complexity, their architecture and training objectives are focused towards versatility, encompassing tasks such as natural language understanding, reasoning and multimodal processing. As a result, their performance in more complex software development scenarios such as the creation of large-scale modular systems or complete applications, is limited.

During the course of this work, new tools specialized in the automatic generation of complete code structures have emerged, such as Firebase Studio, which make it possible to build applications with thousands of lines distributed in multiple files and folders. However, the present study has deliberately focused on the language models most commonly used on a day-to-day basis by the developer community, whose versatility makes them especially relevant in general programming contexts. These models, not being exclusively designed for large-scale code generation, present certain limitations in the creation of complex projects with multiple files, but they are the most representative of current practical, everyday use. Therefore, the proposed evaluation framework has focused on scripts and modules of medium to high complexity, contained in a single file, which allows a homogeneous and realistic comparison of their capabilities in typical tasks of individual software development or rapid prototyping.

Another limitation is related to the benchmark design used. Although the selected programs cover a wide variety of domains and levels of difficulty, and have been carefully chosen to represent common tasks in software development, it is important to recognize that they do not cover all the complexity and diversity of real-world scenarios. As for the design of the prompts, we deliberately opted for a zero-shot strategy with assigned role, which allowed us to guarantee homogeneity in the evaluation between models and to eliminate biases derived from previous examples. However, this choice leaves out other more sophisticated prompting techniques, such as few-shot, chain-of-thought or the use of external context retrieval tools, which in future studies could complement and enrich the analysis.

Finally, it should be noted that the entire evaluation was based on automatically extracted quantitative metrics. Human judgment has not been incorporated to evaluate subjective aspects such as readability, elegance or maintainability of the code in collaborative contexts, although it should be noted that metrics have been included to evaluate these aspects with respect to different standards. While the metrics employed are supported by software engineering standards and have been statistically analyzed, certain subjective nuances of professional development may have been left out of the scope of this analysis.

These limitations indicate that, although the study provides comparative conclusions of value, the results should not be interpreted as an absolute representation of the capabilities of each model in all possible programming contexts, although it does provide a fairly objective representation supported by several reliable metrics.

7.3 Implications for software development

The results obtained in this study have relevant implications for modern software development, especially in contexts where language models are used as automatic assistants in programming tasks. First, the ability of LLMs to generate not only functional code but also unit tests demonstrates their potential as stand-alone tools in continuous development, rapid prototyping or assisted code generation environments.

The strong negative correlation observed between total cyclomatic complexity and maintainability index suggests that models tend to generate functional but structurally more complex solutions, which may affect long-term code maintainability. This finding underscores the need to integrate structural quality metrics into monitoring systems when using AI-generated code in production.

Likewise, automatic test generation reveals a significant advantage in terms of productivity, although it also shows limitations in coverage, accuracy and depth of the generated cases. From what has been seen in this study, LLMs can act as a support in the early stages of testing, but still require human validation to guarantee a complete evaluation of the system behavior.

Finally, the correlation between generated comments and maintainability points to a positive effect of understanding the produced code. However, the quality and semantic relevance of comments remains an area for improvement in critical applications.

Overall, these results indicate that LLMs can be very useful support tools in specific tasks of the software life cycle, provided that they are used with a clear knowledge of their limitations and under adequate control and review mechanisms.

7.4 Future directions for improving LLM-based code generation

The results obtained show several opportunities for improving language model-assisted code generation. If we look at the technical aspects, a key direction is the optimization of prompts, incorporating advanced approaches such as few-shot prompting, chain-of-thought or the use of contextualized examples. These strategies could improve the semantic coherence, structural organization and functional coverage of the generated code. In addition, fine-tuning of models on specialized repositories, with good programming practices and well-documented test cases, would allow LLMs to be adapted to specific contexts, such as educational environments, critical domains or specific languages.

As for the evaluation process, it is proposed to move towards more complete contexts that combine quantitative metrics with qualitative validations, including human reviews or formal verification techniques. Integrating static analysis or automatic error detection would help to improve the reliability of the generated code. In the medium term, these advances will allow LLMs to evolve from passive assistants to more autonomous and specialized agents within the software lifecycle, as we are already seeing with Firebase Studio or Cursor.

7.5 Recommendations for further research

This paper has approached the evaluation of code generation by LLMs from a quantitative perspective and based on structural and functional metrics. However, there are numerous avenues to extend this project.

One possibility is to apply this same study to other programming languages, such as Java or C#, which would allow us to analyze how the performance of the models varies according to the language.

Another interesting line would be to study the interaction between the user and the model in real environments, analyzing how human developers use, correct or reject the proposed code.

In addition, it would be valuable to conduct studies dedicated to new emerging tools focused exclusively on assisted programming, such as Firebase Studio or Cursor, which combine code generation with visual interfaces or development environments integrated with LLMs.

Finally, it is recommended to develop continuous evaluations that measure the progress of the models throughout future versions, as well as studies that combine automatic metrics with human feedback, thus providing a more complete view of the impact of these technologies in professional development contexts.

8 References

[1] *A Survey on Large Language Models for Code Generation*, Anonymous, arXiv:2406.00515, 2024. https://arxiv.org/abs/2406.00515

[2] *Large Language Models for Code Generation: A Comprehensive Survey of Challenges, Techniques, Evaluation, and Applications*, Anonymous, arXiv:2503.01245, 2025. https://arxiv.org/abs/2503.01245

[3] *Github Copilot Adoption Trends: Insights from Real Data*, Opsera, 2025. https://www.opsera.io/blog/github-copilot-adoption-trends-insights-from-real-data

[4] *OpenAI Codex*, Wikipedia, 2025. https://en.wikipedia.org/wiki/OpenAI_Codex

[5] *A Comparative Study on Code Generation with Transformers*, Anonymous, arXiv:2412.05749, 2020. https://arxiv.org/abs/2412.05749

[6] *Codex Open AI: The Ultimate Guide to AI-Powered Coding in 2025*, Emelia.io, 2025. https://emelia.io/hub/codex-open-ai

[7] *Python Code Generation Using Transformers*, GeeksforGeeks, 2024. https://www.geeksforgeeks.org/python-code-generation-using-transformers/

[8] *Framework for evaluating code generation ability of large language models*, Wiley, 2024. https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0357

[9] *Enhance your GitHub Copilot Adoption*, KPMG, 2024. https://kpmg.com/us/en/articles/2024/enhance-github-copilot-adoption.html

[10] *Introducing Codex*, OpenAI, 2025. https://openai.com/index/introducing-codex/

[11] *CodeGen*, Hugging Face, 2025. https://huggingface.co/docs/transformers/model_doc/codegen

[12] *Analysis of Code and Test-Code generated by Large Language Models*, Robin Beer, Alexander Feix, Tim Guttzeit, Tamara Muras, Vincent Müller, Maurice Rauscher, Florian Schäffler, Welf Löwe. arXiv:2408.16601v1, 2024.

[13] *Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation*, Davide Tosi. Future Internet 2024, 16, 188.

[14] SonarSource, “LLMs for Code Generation: A summary of the research on quality,” 2024. https://www.sonarsource.com/learn/llm-code-generation/

[15] Klu.ai, “HumanEval Benchmark,” https://klu.ai/glossary/humaneval-benchmark

[16] Papers With Code, “HumanEval Benchmark (Code Generation),” https://paperswithcode.com/sota/code-generation-on-humaneval

[17] Athina AI, "Top 3 Benchmarks to Evaluate LLMs for Code Generation," 2025. https://hub.athina.ai/top-benchmarks-to-evaluate-llms-for-code-generation/

[18] Carnegie Mellon University, "Navigating Challenges with LLM-based Code Generation using Software-specific Insights," 2025. https://kilthub.cmu.edu/articles/thesis/Navigating_Challenges_with_LLM-based_Code_Generation_using_Software-specific_Insights/29316002

[19] Thomas J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308–320, 1976. https://ieeexplore.ieee.org/document/1702388

[20] Miquel Canal, "Radon: Python tool for code metrics," 2024. https://radon.readthedocs.io/en/latest/api.html#radon.complexity.cc_visit

[21] PyCQA, "pycodestyle 2.11.1 documentation," 2024. https://pycodestyle.pycqa.org/en/latest/

[22] Ned Batchelder, "Coverage.py: Code coverage measurement for Python," 2024. https://coverage.readthedocs.io/en/latest/

[23] Guido van Rossum et al., "PEP 8 – Style Guide for Python Code," Python Software Foundation, 2001. https://peps.python.org/pep-0008/

[24] scikit-learn developers, "sklearn.metrics.pairwise.cosine_similarity," 2024. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html

[25] NLTK Project, "Jaccard Distance," 2024. https://www.nltk.org/howto/collocations.html#jaccard-distance

[26] Python Software Foundation, "ast — Abstract Syntax Trees," Python 3.12.3 documentation, 2024. https://docs.python.org/3/library/ast.html

[27] Wikipedia, "Source lines of code," 2024. https://en.wikipedia.org/wiki/Source_lines_of_code

[28] Microsoft Docs, "Maintainability Index," 2024. https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index

[29] Stack Overflow, "How to detect duplicate code in Python?," 2024. https://stackoverflow.com/questions/30212413/how-to-detect-duplicate-code-in-python

[30] Python Software Foundation, "Modulefinder — Find modules used by a script," Python 3.12.3 documentation, 2024. https://docs.python.org/3/library/modulefinder.html

[31] Oregon State "Measuring Execution Time" https://web.engr.oregonstate.edu/~sinisa/courses/OSU/CS261/CS261_Textbook/Chapter04.pdf

[32] Radon "Using radon programmatically" https://radon.readthedocs.io/en/latest/api.html#radon.complexity.cc_visit

[33] Amazon "What is LLM (Large Language Model)¿' https://aws.amazon.com/what-is/large-language-model/

[34] OpenAI, "Hello GPT-4o", 2024. https://openai.com/index/hello-gpt-4o/

[35] OpenAI, "Introducing OpenAI o3 and o4-mini", 2025. https://openai.com/index/introducing-o3-and-o4-mini

[36] Google Deepmind, "Gemini 2.5 Pro" 2025. https://deepmind.google/models/gemini/pro/

[37] xAI, "Grok 3 Beta — The Age of Reasoning Agents", 2025. https://x.ai/news/grok-3

[38] Wikipedia contributors. "Pearson correlation coefficient. In Wikipedia, The Free Encyclopedia." 2025 https://en.wikipedia.org/w/index.php?title=Pearson_correlation_coefficient&oldid=1297070537

A Appendix: Prompts and LLM Outputs

Prompt #1: Tic-Tac-Toe

Prompt:
You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked with developing a complete solution for a command-line Tic Tac Toe game using Python. The game should allow two players to compete by taking turns to place their marks on a 3×3 board. The board must be displayed after every move, and the program should validate each player's input by ensuring that the provided row and column values are valid integers within the specified range and that the chosen cell is not already occupied. The game logic should include the detection of winning conditions—three of the same marks in a row, column, or diagonal—as well as the identification of a draw when the board is filled without any winner. Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone; they are run as follows: `exec(main_code.py, ns)`; `exec(test_code.py, ns)`. Each, the tests and script classes must have its own `if __name__ == "__main__": main()`.

A.1 Reference Code

```
def print_board(board):
    for row in board:
        print("␣|␣".join(row))
        print("-" * 9)

def check_winner(board, player):
    for i in range(3):
        # Check rows and columns
        if all(board[i][j] == player for j in range(3)) or all(
            board[j][i] == player for j in range(3)):
             return True
    # Check diagonals
    if all(board[i][i] == player for i in range(3)) or all(board[
        i][2 - i] == player for i in range(3)):
        return True
    return False

def is_full(board):
    return all(cell != "␣" for row in board for cell in row)
# A function that validates user input
def get_valid_input(prompt):
    while True:
        try:
            value = int(input(prompt))
```

```
            if 0 <= value < 3:  # Check if the value is within the valid range
                return value
            else:
                print("Invalid␣input:␣Enter␣a␣number␣between␣0␣and␣2.")
        except ValueError:
            print("Invalid␣input:␣Please␣enter␣an␣integer.")

def main():
    board = [["␣" for _ in range(3)] for _ in range(3)]
    player = "X"

    while True:
        print_board(board)
        print(f"Player␣{player}'s␣turn:")

        # Get validated inputs
        row = get_valid_input("Enter␣the␣row␣(0,␣1,␣2):␣")
        col = get_valid_input("Enter␣the␣column␣(0,␣1,␣2):␣")

        if board[row][col] == "␣":
            board[row][col] = player

            if check_winner(board, player):
                print_board(board)
                print(f"Player␣{player}␣wins!")
                break

            if is_full(board):
                print_board(board)
                print("It's␣a␣draw!")
                break

            player = "O" if player == "X" else "X"
        else:
            print("Invalid␣move:␣That␣spot␣is␣already␣taken.␣Try␣again.")

if __name__ == "__main__":
    main()
```

Listing 1: Reference Code — GPT4o

GPT4o Response

Generated Code

```
## tic_tac_toe.py - Main Game Logic

def print_board(board):
    print("\nCurrent␣Board:")
```

```
    for row in board:
        print("␣|␣".join(row))
        print("-" * 9)

def check_winner(board, player):
    # Rows, columns and diagonals
    for i in range(3):
        if all(board[i][j] == player for j in range(3)): return
           True
        if all(board[j][i] == player for j in range(3)): return
           True
    if all(board[i][i] == player for i in range(3)): return True
    if all(board[i][2 - i] == player for i in range(3)): return
       True
    return False

def is_full(board):
    return all(cell != "␣" for row in board for cell in row)

def get_valid_input(prompt, board):
    while True:
        try:
            value = input(prompt)
            row, col = map(int, value.split())
            if 0 <= row < 3 and 0 <= col < 3:
                if board[row][col] == "␣":
                    return row, col
                else:
                    print("Cell␣is␣already␣occupied.")
            else:
                print("Input␣out␣of␣range.␣Use␣values␣between␣0␣
                   and␣2.")
        except ValueError:
            print("Invalid␣input.␣Please␣enter␣two␣integers␣
               separated␣by␣space.")

def main():
    board = [["␣" for _ in range(3)] for _ in range(3)]
    current_player = "X"

    while True:
        print_board(board)
        row, col = get_valid_input(f"Player␣{current_player},␣
           enter␣your␣move␣(row␣and␣column):␣", board)
        board[row][col] = current_player

        if check_winner(board, current_player):
            print_board(board)
            print(f"Player␣{current_player}␣wins!")
            break
        elif is_full(board):
```

```
                print_board(board)
                print("It's␣a␣draw!")
                break
            else:
                current_player = "O" if current_player == "X" else "X
                   "

if __name__ == "__main__":
    main()
```

Listing 2: Generated Code — GPT4o

Test Code

```
# test_tic_tac_toe.py - Unit Tests

import unittest
from io import StringIO
import sys

class TestTicTacToe(unittest.TestCase):

    def test_check_winner_rows(self):
        board = [
            ['X', 'X', 'X'],
            ['O', '␣', 'O'],
            ['␣', '␣', '␣']
        ]
        self.assertTrue(check_winner(board, 'X'))
        self.assertFalse(check_winner(board, 'O'))

    def test_check_winner_columns(self):
        board = [
            ['O', 'X', '␣'],
            ['O', 'X', '␣'],
            ['O', '␣', 'X']
        ]
        self.assertTrue(check_winner(board, 'O'))
        self.assertFalse(check_winner(board, 'X'))

    def test_check_winner_diagonals(self):
        board1 = [
            ['X', 'O', '␣'],
            ['O', 'X', '␣'],
            ['␣', '␣', 'X']
        ]
        board2 = [
            ['␣', '␣', 'O'],
            ['␣', 'O', '␣'],
            ['O', '␣', '␣']
        ]
        self.assertTrue(check_winner(board1, 'X'))
```

```
        self.assertTrue(check_winner(board2, ’O’))

    def test_is_full_true(self):
        board = [
            [’X’, ’O’, ’X’],
            [’O’, ’X’, ’O’],
            [’O’, ’X’, ’O’]
        ]
        self.assertTrue(is_full(board))

    def test_is_full_false(self):
        board = [
            [’X’, ’O’, ’X’],
            [’O’, ’␣’, ’O’],
            [’O’, ’X’, ’O’]
        ]
        self.assertFalse(is_full(board))

    def test_print_board_output(self):
        board = [
            [’X’, ’O’, ’X’],
            [’O’, ’X’, ’O’],
            [’O’, ’X’, ’O’]
        ]
        expected_output = (
            "\nCurrent␣Board:\n"
            "X␣|␣O␣|␣X\n"
            "---------\n"
            "O␣|␣X␣|␣O\n"
            "---------\n"
            "O␣|␣X␣|␣O\n"
            "---------\n"
        )

        captured_output = StringIO()
        sys.stdout = captured_output
        print_board(board)
        sys.stdout = sys.__stdout__

        output = captured_output.getvalue()
        self.assertIn("Current␣Board", output)
        self.assertIn("X␣|␣O␣|␣X", output)

if __name__ == "__main__":
    unittest.main()
```

Listing 3: Test Code — GPT4o

GPT o4-mini-high Response

Generated Code

```
"""
Command-line Tic Tac Toe Game

This script implements a two-player Tic Tac Toe game on a 3x3 board.
Players take turns entering their moves. The board is displayed after every
move, and input is validated to ensure valid numbers (1 3 ) are entered,
and that the chosen cell is not already occupied. The game checks for a win
(after a move, three of the same mark in a row, column or diagonal) or a draw
(if the board fills without a winner).
"""

class TicTacToe:
    def __init__(self):
        # Initialize a 3x3 board filled with spaces and set the starting player.
        self.board = [['␣' for _ in range(3)] for _ in range(3)]
        self.current_player = 'X'
        self.winner = None

    def display_board(self):
        # Display the current board state.
        for i, row in enumerate(self.board):
            print("␣|␣".join(row))
            if i < 2:
                print("-" * 9)

    def is_valid_move(self, row, col):
        # A move is valid if the row and col are in range and the cell is empty.
        if row < 0 or row > 2 or col < 0 or col > 2:
            return False
        return self.board[row][col] == '␣'

    def make_move(self, row, col):
        # Place the current player's mark on the board if the move is valid.
        if not self.is_valid_move(row, col):
            return False
        self.board[row][col] = self.current_player
        return True

    def check_winner(self):
        # Check rows for a win.
        for row in self.board:
```

```
            if row[0] != '␣' and row[0] == row[1] == row[2]:
                self.winner = row[0]
                return self.winner

        # Check columns for a win.
        for col in range(3):
            if self.board[0][col] != '␣' and \
               self.board[0][col] == self.board[1][col] == self.
                  board[2][col]:
                self.winner = self.board[0][col]
                return self.winner

        # Check the two diagonals for a win.
        if self.board[0][0] != '␣' and \
           self.board[0][0] == self.board[1][1] == self.board
              [2][2]:
            self.winner = self.board[0][0]
            return self.winner

        if self.board[0][2] != '␣' and \
           self.board[0][2] == self.board[1][1] == self.board
              [2][0]:
            self.winner = self.board[0][2]
            return self.winner

        return None

    def check_draw(self):
        # A draw occurs when there are no empty cells and no
           winner.
        if any('␣' in row for row in self.board) or self.winner
           is not None:
            return False
        return True

    def switch_player(self):
        # Toggle between players 'X' and 'O'.
        self.current_player = 'O' if self.current_player == 'X'
           else 'X'

    def play(self):
        # Main loop for the interactive game.
        while True:
            self.display_board()
            move_made = False
            while not move_made:
                user_input = input(
                    f"Player␣{self.current_player},␣enter␣your␣
                       move␣(row␣and␣column,␣1-3,␣separated␣by␣
                       space):␣"
                )
```

```
                try:
                    row_str, col_str = user_input.split()
                    row = int(row_str) - 1  # Convert to 0-
                       indexed.
                    col = int(col_str) - 1
                except ValueError:
                    print("Invalid␣input.␣Please␣enter␣two␣
                       numbers␣separated␣by␣a␣space.")
                    continue

                if row not in [0, 1, 2] or col not in [0, 1, 2]:
                    print("Row␣and␣column␣must␣be␣between␣1␣and␣
                       3.")
                    continue

                if not self.is_valid_move(row, col):
                    print("That␣cell␣is␣already␣occupied.␣Try␣a␣
                       different␣move.")
                    continue

                move_made = self.make_move(row, col)

            if self.check_winner():
                self.display_board()
                print(f"Player␣{self.winner}␣wins!")
                break

            if self.check_draw():
                self.display_board()
                print("The␣game␣is␣a␣draw!")
                break

            self.switch_player()

def main():
    game = TicTacToe()
    game.play()

if __name__ == "__main__":
    main()
```

Listing 4: Generated Code — GPTo4-mini-high

Test Code

```
"""
Unit Tests for Tic Tac Toe Game

This script contains a comprehensive suite of unit tests to
   verify that the Tic Tac Toe
game logic works correctly. It tests the functionality for valid
   moves, win detection
```

```
(in rows, columns, and diagonals), draw detection, and player
   switching.
"""

import unittest

class TestTicTacToe(unittest.TestCase):

    def test_valid_move(self):
        game = TicTacToe()
        # Initially, (0,0) should be a valid move.
        self.assertTrue(game.is_valid_move(0, 0))
        # Making a move at (0,0) should succeed.
        self.assertTrue(game.make_move(0, 0))
        # Now the same cell should not be valid.
        self.assertFalse(game.is_valid_move(0, 0))
        # Trying to move again at (0,0) should fail.
        self.assertFalse(game.make_move(0, 0))

    def test_row_win(self):
        game = TicTacToe()
        game.board = [
            ['X', 'X', 'X'],
            ['O', ' ', ' '],
            ['O', ' ', ' ']
        ]
        winner = game.check_winner()
        self.assertEqual(winner, 'X')

    def test_column_win(self):
        game = TicTacToe()
        game.board = [
            ['O', 'X', ' '],
            ['O', 'X', ' '],
            [' ', 'X', ' ']
        ]
        winner = game.check_winner()
        self.assertEqual(winner, 'X')

    def test_diagonal_win(self):
        game = TicTacToe()
        # Test main diagonal win.
        game.board = [
            ['O', 'X', 'X'],
            [' ', 'O', ' '],
            [' ', ' ', 'O']
        ]
        winner = game.check_winner()
        self.assertEqual(winner, 'O')

        # Test anti-diagonal win.
```

```
        game2 = TicTacToe()
        game2.board = [
            ['X', 'O', 'O'],
            ['␣', 'O', '␣'],
            ['O', '␣', 'X']
        ]
        winner2 = game2.check_winner()
        self.assertEqual(winner2, 'O')

    def test_draw(self):
        game = TicTacToe()
        game.board = [
            ['X', 'O', 'X'],
            ['O', 'X', 'O'],
            ['O', 'X', 'O']
        ]
        # Ensure that no winner has been set.
        game.winner = None
        self.assertTrue(game.check_draw())

        # If a winner is set, it should not be considered a draw.
        game.winner = 'X'
        self.assertFalse(game.check_draw())

    def test_switch_player(self):
        game = TicTacToe()
        self.assertEqual(game.current_player, 'X')
        game.switch_player()
        self.assertEqual(game.current_player, 'O')
        game.switch_player()
        self.assertEqual(game.current_player, 'X')

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 5: Test Code — GPTo4-mini-high

GEMINI 2.5 Response

Generated Code

```
import sys
import unittest
from io import StringIO

# --- Main Game Code ---

class TicTacToeGame:
    """
    Represents and manages a game of Tic Tac Toe.
    """
    def __init__(self):
        """Initializes the game board, current player, and game
           status."""
        self.board = [[’␣’ for _ in range(3)] for _ in range(3)]
        self.current_player = ’X’
        self.game_over = False
        self.winner = None

    def display_board(self):
        """Prints the current state of the Tic Tac Toe board to
           the console."""
        print("\n␣␣0␣1␣2")
        for i, row in enumerate(self.board):
            print(f"{i}␣{’|’.join(row)}")
            if i < 2:
                print("␣␣-----")
        print() # Extra newline for spacing

    def is_valid_move(self, row, col):
        """
        Checks if a move is valid.

        Args:
            row (int): The row index (0-2).
            col (int): The column index (0-2).

        Returns:
            bool: True if the move is valid (within bounds and
               cell is empty), False otherwise.
        """
        if not (0 <= row <= 2 and 0 <= col <= 2):
            print("Error:␣Row␣and␣column␣must␣be␣between␣0␣and␣2.
               ")
            return False
        if self.board[row][col] != ’␣’:
            print("Error:␣Cell␣is␣already␣occupied.␣Choose␣
               another␣cell.")
            return False
        return True
```

```python
    def make_move(self, row, col):
        """
        Places the current player's mark on the board if the move is valid.

        Args:
            row (int): The row index (0-2).
            col (int): The column index (0-2).

        Returns:
            bool: True if the move was successfully made, False otherwise.
        """
        if self.is_valid_move(row, col):
            self.board[row][col] = self.current_player
            return True
        return False

    def check_win(self):
        """
        Checks if the current player has won the game.

        Returns:
            bool: True if the current player has won, False otherwise.
        """
        player = self.current_player
        # Check rows
        for row in self.board:
            if all(cell == player for cell in row):
                return True
        # Check columns
        for col in range(3):
            if all(self.board[row][col] == player for row in range(3)):
                return True
        # Check diagonals
        if all(self.board[i][i] == player for i in range(3)):
            return True
        if all(self.board[i][2 - i] == player for i in range(3)):
            return True
        return False

    def check_draw(self):
        """
        Checks if the game is a draw (board is full, no winner).

        Returns:
            bool: True if the game is a draw, False otherwise.
        """
```

```
        # A draw occurs if the board is full AND no one has won
        # (The win condition is checked before the draw condition
            in the game loop)
        for row in self.board:
            if any(cell == ’␣’ for cell in row):
                return False  # Board is not full
        # If we reach here, the board is full. Since check_win is
            called first,
        # if check_win was False and the board is full, it’s a
           draw.
        return True

    def switch_player(self):
        """Switches the current player between ’X’ and ’O’."""
        self.current_player = ’O’ if self.current_player == ’X’
           else ’X’

    def get_player_input(self):
        """
        Prompts the current player for their move and validates
           the input.

        Returns:
            tuple[int, int] or None: A tuple (row, col) if input
               is valid, None otherwise.
        """
        while True:
            try:
                prompt = (f"Player␣{self.current_player},␣enter␣
                   your␣move␣"
                          f"(row␣and␣column,␣e.g.,␣’1␣2’):␣")
                move_input = input(prompt).split()

                if len(move_input) != 2:
                    print("Error:␣Please␣enter␣exactly␣two␣
                       numbers␣(row␣and␣column)␣separated␣by␣a␣
                       space.")
                    continue

                row_str, col_str = move_input
                row = int(row_str)
                col = int(col_str)
                return row, col # Validation of range and
                   occupancy happens in make_move

            except ValueError:
                print("Error:␣Invalid␣input.␣Please␣enter␣numbers
                   ␣for␣row␣and␣column.")
            except Exception as e:
                print(f"An␣unexpected␣error␣occurred:␣{e}")
```

```
    def play_game(self):
        """Runs the main game loop."""
        print("Welcome␣to␣Tic␣Tac␣Toe!")
        self.display_board()

        while not self.game_over:
            # Get valid player input
            move_made = False
            while not move_made:
                coords = self.get_player_input()
                if coords:
                    row, col = coords
                    if self.make_move(row, col):
                        move_made = True
                    # If make_move fails, is_valid_move already
                       printed the error
                # If get_player_input had an issue, it already
                   printed the error

            self.display_board()

            # Check for win
            if self.check_win():
                self.winner = self.current_player
                self.game_over = True
                print(f"Player␣{self.winner}␣wins!")
            # Check for draw (only if no one won)
            elif self.check_draw():
                self.game_over = True
                print("It's␣a␣draw!")
            # If game is not over, switch player
            else:
                self.switch_player()

        print("Game␣Over.")


def main():
    """Entry point for the Tic Tac Toe game script."""
    game = TicTacToeGame()
    game.play_game()

if __name__ == "__main__":
    # This block runs when the script is executed directly
    main()
```

Listing 6: Generated Code — GEMINI 2.5

Test Code

```
class TestTicTacToeGame(unittest.TestCase):
```

```
    """Unit tests for the TicTacToeGame class."""

    def setUp(self):
        """Set up a new game instance before each test."""
        self.game = TicTacToeGame()
        # Suppress print statements during tests by redirecting stdout
        self.held_stdout = sys.stdout
        sys.stdout = StringIO()

    def tearDown(self):
        """Restore stdout after each test."""
        sys.stdout = self.held_stdout

    def test_initial_state(self):
        """Test the initial state of the game board and player."""
        self.assertEqual(self.game.board, [['␣', '␣', '␣'], ['␣', '␣', '␣'], ['␣', '␣', '␣']])
        self.assertEqual(self.game.current_player, 'X')
        self.assertFalse(self.game.game_over)
        self.assertIsNone(self.game.winner)

    def test_switch_player(self):
        """Test switching the current player."""
        self.assertEqual(self.game.current_player, 'X')
        self.game.switch_player()
        self.assertEqual(self.game.current_player, 'O')
        self.game.switch_player()
        self.assertEqual(self.game.current_player, 'X')

    def test_is_valid_move_valid(self):
        """Test valid moves on an empty board."""
        self.assertTrue(self.game.is_valid_move(0, 0))
        self.assertTrue(self.game.is_valid_move(1, 1))
        self.assertTrue(self.game.is_valid_move(2, 2))

    def test_is_valid_move_out_of_bounds(self):
        """Test moves outside the board boundaries."""
        self.assertFalse(self.game.is_valid_move(-1, 0))
        self.assertFalse(self.game.is_valid_move(0, 3))
        self.assertFalse(self.game.is_valid_move(3, 3))

    def test_is_valid_move_occupied(self):
        """Test moves on an already occupied cell."""
        self.game.board[1][1] = 'X'
        self.assertFalse(self.game.is_valid_move(1, 1))
        self.assertTrue(self.game.is_valid_move(0, 0)) # Check other cell still valid

    def test_make_move_valid(self):
```

```
        """Test making a valid move."""
        self.assertTrue(self.game.make_move(0, 0))
        self.assertEqual(self.game.board[0][0], 'X')
        # Player should not switch automatically in make_move
        self.assertEqual(self.game.current_player, 'X')

    def test_make_move_invalid(self):
        """Test making an invalid move (occupied cell)."""
        self.game.make_move(1, 1) # X moves at 1,1
        self.game.switch_player() # Switch to O
        self.assertFalse(self.game.make_move(1, 1)) # O tries to move at 1,1
        self.assertEqual(self.game.board[1][1], 'X') # Cell should still be X

    def test_check_win_row(self):
        """Test winning condition in a row."""
        self.game.board = [['X', 'X', 'X'], ['␣', 'O', '␣'], ['O', '␣', '␣']]
        self.game.current_player = 'X'
        self.assertTrue(self.game.check_win())
        self.game.current_player = 'O' # O should not win
        self.assertFalse(self.game.check_win())

    def test_check_win_column(self):
        """Test winning condition in a column."""
        self.game.board = [['O', 'X', '␣'], ['O', 'X', '␣'], ['␣', 'X', 'O']]
        self.game.current_player = 'X'
        self.assertTrue(self.game.check_win())
        self.game.current_player = 'O' # O should not win
        self.assertFalse(self.game.check_win())

    def test_check_win_diagonal1(self):
        """Test winning condition in the main diagonal."""
        self.game.board = [['O', 'X', 'X'], ['␣', 'O', '␣'], ['X', '␣', 'O']]
        self.game.current_player = 'O'
        self.assertTrue(self.game.check_win())
        self.game.current_player = 'X' # X should not win
        self.assertFalse(self.game.check_win())

    def test_check_win_diagonal2(self):
        """Test winning condition in the anti-diagonal."""
        self.game.board = [['O', '␣', 'X'], ['␣', 'X', 'O'], ['X', '␣', '␣']]
        self.game.current_player = 'X'
        self.assertTrue(self.game.check_win())
        self.game.current_player = 'O' # O should not win
        self.assertFalse(self.game.check_win())
```

```
    def test_check_win_no_win(self):
        """Test a board state with no winner yet."""
        self.game.board = [['X', 'O', 'X'], ['O', 'O', '␣'], ['X', '␣', 'O']]
        self.game.current_player = 'X'
        self.assertFalse(self.game.check_win())
        self.game.current_player = 'O'
        self.assertFalse(self.game.check_win())

    def test_check_draw_true(self):
        """Test a board state that results in a draw."""
        self.game.board = [['X', 'O', 'X'], ['X', 'O', 'O'], ['O', 'X', 'X']]
        # Ensure no one actually won (though check_draw doesn't strictly need this)
        self.game.current_player = 'X'; self.assertFalse(self.game.check_win())
        self.game.current_player = 'O'; self.assertFalse(self.game.check_win())
        # Now test draw
        self.assertTrue(self.game.check_draw())

    def test_check_draw_false_not_full(self):
        """Test draw condition on a non-full board."""
        self.game.board = [['X', 'O', 'X'], ['␣', 'O', 'O'], ['O', 'X', 'X']]
        self.assertFalse(self.game.check_draw())

    def test_check_draw_false_winner(self):
        """Test draw condition on a full board where someone has won."""
        # Although the game loop checks win *before* draw, the draw function
        # itself should return true if the board is full, regardless of win state.
        # The game logic handles the precedence.
        self.game.board = [['X', 'X', 'X'], ['O', 'O', 'X'], ['X', 'O', 'O']]
        self.assertTrue(self.game.check_draw()) # Board is full


def run_tests():
    """Runs the unit tests."""
    # Need to use TestLoader to run tests within the same script conditionally
    loader = unittest.TestLoader()
    suite = loader.loadTestsFromTestCase(TestTicTacToeGame)
    runner = unittest.TextTestRunner()
    print("\n---␣Running␣Unit␣Tests␣---")
    runner.run(suite)
    print("---␣Unit␣Tests␣Finished␣---\n")
```

```
# Example of how to run tests if needed, separate from the main game execution
# Typically, you'd run tests using `python -m unittest your_script_name.py`
# But to keep it self-contained as requested:
if __name__ == "__main__":
    # The main game runs first because of the earlier if __name__ == "__main__":
    # If you want to *only* run tests, you could comment out the main() call above
    # or add command-line argument handling.
    # For simplicity here, we'll just print a message that tests *can* be run.
    # To actually run them from here, uncomment the next line:
    # run_tests()
    print("To␣run␣unit␣tests,␣uncomment␣'run_tests()'␣in␣the␣script's␣second")
    print("'if␣__name__␣==␣\"__main__\":'␣block␣or␣run:")
    print(f"python␣-m␣unittest␣{__file__}")
```

Listing 7: Test Code — GEMINI 2.5

GROK 3 Response

Generated Code

```
# Main Tic Tac Toe Game Implementation

class TicTacToe:
    def __init__(self):
        self.board = [[’␣’ for _ in range(3)] for _ in range(3)]
        self.current_player = ’X’

    def display_board(self):
        print(’\n’)
        for i, row in enumerate(self.board):
            print(’|’.join(row))
            if i < 2:
                print(’-’ * 5)
        print(’\n’)

    def make_move(self, row, col):
        if not self.is_valid_move(row, col):
            return False
        self.board[row][col] = self.current_player
        self.current_player = ’O’ if self.current_player == ’X’
           else ’X’
        return True

    def is_valid_move(self, row, col):
        if not (0 <= row <= 2 and 0 <= col <= 2):
            return False
        return self.board[row][col] == ’␣’

    def check_winner(self):
        # Check rows
        for row in self.board:
            if row[0] == row[1] == row[2] != ’␣’:
                return row[0]
        # Check columns
        for col in range(3):
            if self.board[0][col] == self.board[1][col] == self.
               board[2][col] != ’␣’:
                return self.board[0][col]
        # Check diagonals
        if self.board[0][0] == self.board[1][1] == self.board
           [2][2] != ’␣’:
            return self.board[0][0]
        if self.board[0][2] == self.board[1][1] == self.board
           [2][0] != ’␣’:
            return self.board[0][2]
        return None

    def is_board_full(self):
```

```
        return all(cell != '␣' for row in self.board for cell in
           row)

def get_player_input():
    try:
        row = int(input("Enter␣row␣(0-2):␣"))
        col = int(input("Enter␣column␣(0-2):␣"))
        return row, col
    except ValueError:
        return None, None

def main():
    game = TicTacToe()
    print("Welcome␣to␣Tic␣Tac␣Toe!")

    while True:
        game.display_board()
        print(f"Player␣{game.current_player}'s␣turn")

        row, col = get_player_input()
        if row is None or not game.make_move(row, col):
            print("Invalid␣move!␣Try␣again.")
            continue

        winner = game.check_winner()
        if winner:
            game.display_board()
            print(f"Player␣{winner}␣wins!")
            break

        if game.is_board_full():
            game.display_board()
            print("It's␣a␣draw!")
            break

if __name__ == "__main__":
    main()
```

Listing 8: Generated Code — GROK 3

Test Code

```
import unittest
class TestTicTacToe(unittest.TestCase):
    def setUp(self):
        self.game = TicTacToe()

    def test_initial_board_empty(self):
        expected = [['␣' for _ in range(3)] for _ in range(3)]
        self.assertEqual(self.game.board, expected)

    def test_initial_player(self):
```

```
        self.assertEqual(self.game.current_player, ’X’)

    def test_valid_move(self):
        result = self.game.make_move(0, 0)
        self.assertTrue(result)
        self.assertEqual(self.game.board[0][0], ’X’)
        self.assertEqual(self.game.current_player, ’O’)

    def test_invalid_move_out_of_bounds(self):
        result = self.game.make_move(3, 0)
        self.assertFalse(result)
        result = self.game.make_move(-1, 0)
        self.assertFalse(result)

    def test_invalid_move_occupied(self):
        self.game.make_move(0, 0)
        result = self.game.make_move(0, 0)
        self.assertFalse(result)

    def test_player_switch(self):
        self.game.make_move(0, 0)
        self.assertEqual(self.game.current_player, ’O’)
        self.game.make_move(1, 1)
        self.assertEqual(self.game.current_player, ’X’)

    def test_row_win(self):
        self.game.board = [[’X’, ’X’, ’X’],
                         [’␣’, ’␣’, ’␣’],
                         [’␣’, ’␣’, ’␣’]]
        self.assertEqual(self.game.check_winner(), ’X’)

    def test_column_win(self):
        self.game.board = [[’O’, ’␣’, ’␣’],
                         [’O’, ’␣’, ’␣’],
                         [’O’, ’␣’, ’␣’]]
        self.assertEqual(self.game.check_winner(), ’O’)

    def test_diagonal_win(self):
        self.game.board = [[’X’, ’␣’, ’␣’],
                         [’␣’, ’X’, ’␣’],
                         [’␣’, ’␣’, ’X’]]
        self.assertEqual(self.game.check_winner(), ’X’)

    def test_reverse_diagonal_win(self):
        self.game.board = [[’␣’, ’␣’, ’O’],
                         [’␣’, ’O’, ’␣’],
                         [’O’, ’␣’, ’␣’]]
        self.assertEqual(self.game.check_winner(), ’O’)

    def test_no_winner(self):
        self.game.board = [[’X’, ’O’, ’X’],
```

```
                         ['O', 'X', 'O'],
                         ['O', 'X', 'O']]
        self.assertIsNone(self.game.check_winner())

    def test_board_full(self):
        self.game.board = [['X', 'O', 'X'],
                         ['O', 'X', 'O'],
                         ['O', 'X', 'O']]
        self.assertTrue(self.game.is_board_full())

    def test_board_not_full(self):
        self.game.board = [['X', 'O', '␣'],
                         ['O', 'X', 'O'],
                         ['O', 'X', 'O']]
        self.assertFalse(self.game.is_board_full())

def main_test():
    unittest.main()

if __name__ == "__main__":
    main_test()
```

Listing 9: Test Code — GROK 3

Prompt #2: Binary Search

Prompt:
You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked with developing a recursive binary search algorithm. This script must receive as input the array, the value x, the start point and the end point. Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone; they are run as follows: `exec(main_code.py, ns)`; `exec(test_code.py, ns)`. Each, the tests and script classes must have its own `if __name__ == "__main__": main()`.

Reference Code

```python
# Binary Search in python


def binarySearch(array, x, low, high):

    if high >= low:

        mid = low + (high - low)//2

        # If found at mid, then return it
        if x == array[mid]:
            return mid

        # Search the right half
        elif x > array[mid]:
            return binarySearch(array, x, mid + 1, high)

        # Search the left half
        else:
            return binarySearch(array, x, low, mid - 1)

    else:
        return -1


array = [3, 4, 5, 6, 7, 8, 9]
x = 4

result = binarySearch(array, x, 0, len(array)-1)

if result != -1:
    print("Element␣is␣present␣at␣index␣" + str(result))
```

```
else:
    print("Not␣found")
```

Listing 10: Reference Code

GPT4o Response

Generated Code

```
# main_code.py
def recursive_binary_search(arr, x, start, end):
    """
    Performs a recursive binary search to find the index of x in arr[start:end+1].
    Returns the index of x if found, otherwise -1.
    """
    if start > end:
        return -1

    mid = start + (end - start) // 2

    if arr[mid] == x:
        return mid
    elif arr[mid] > x:
        return recursive_binary_search(arr, x, start, mid - 1)
    else:
        return recursive_binary_search(arr, x, mid + 1, end)


def main():
    # Example usage
    arr = [1, 3, 5, 7, 9, 11]
    x = 5
    index = recursive_binary_search(arr, x, 0, len(arr) - 1)
    print(f"Index␣of␣{x}␣is:␣{index}")


if __name__ == "__main__":
    main()
```

Listing 11: Generated Code — GPT4o

Test Code

```
# test_code.py
import unittest


class TestRecursiveBinarySearch(unittest.TestCase):
    def test_element_found(self):
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(recursive_binary_search(arr, 3, 0, len(arr) - 1), 2)
```

```
    def test_element_not_found(self):
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(recursive_binary_search(arr, 6, 0, len(
           arr) - 1), -1)

    def test_single_element_found(self):
        arr = [10]
        self.assertEqual(recursive_binary_search(arr, 10, 0, 0),
           0)

    def test_single_element_not_found(self):
        arr = [10]
        self.assertEqual(recursive_binary_search(arr, 5, 0, 0),
           -1)

    def test_empty_array(self):
        arr = []
        self.assertEqual(recursive_binary_search(arr, 1, 0, -1),
           -1)

    def test_first_element(self):
        arr = [2, 4, 6, 8, 10]
        self.assertEqual(recursive_binary_search(arr, 2, 0, len(
           arr) - 1), 0)

    def test_last_element(self):
        arr = [2, 4, 6, 8, 10]
        self.assertEqual(recursive_binary_search(arr, 10, 0, len(
           arr) - 1), 4)

    def test_duplicates(self):
        arr = [1, 2, 2, 2, 3]
        result = recursive_binary_search(arr, 2, 0, len(arr) - 1)
        self.assertIn(result, [1, 2, 3])  # Accept any valid
           index


def main():
    unittest.main()


if __name__ == "__main__":
    main()
```

Listing 12: Test Code — GPT4o

GPT o4-mini-high Response

Generated Code

```
# ===== main_code.py =====

#!/usr/bin/env python3

def binary_search(arr, x, low, high):
    """
    Perform recursive binary search on sorted list arr.
    Returns the index of x if present, otherwise -1.
    """
    if low > high:
        return -1
    mid = (low + high) // 2
    if arr[mid] == x:
        return mid
    elif arr[mid] < x:
        return binary_search(arr, x, mid + 1, high)
    else:
        return binary_search(arr, x, low, mid - 1)

def main():
    import sys
    if len(sys.argv) != 5:
        print("Usage:␣python␣main_code.py␣<comma-separated-array>
           ␣<x>␣<low>␣<high>")
        sys.exit(1)
    # Parse inputs
    arr = list(map(int, sys.argv[1].split(',')))
    x = int(sys.argv[2])
    low = int(sys.argv[3])
    high = int(sys.argv[4])
    # Execute search and print result
    result = binary_search(arr, x, low, high)
    print(result)

if __name__ == "__main__":
    main()
```

Listing 13: Generated Code — GPTo4-mini-high

Test Code

```
#!/usr/bin/env python3

import unittest

class TestBinarySearch(unittest.TestCase):
    def test_found_middle(self):
        self.assertEqual(binary_search([1, 2, 3, 4, 5], 3, 0, 4),
            2)
```

```
    def test_found_first(self):
        self.assertEqual(binary_search([10, 20, 30], 10, 0, 2),
           0)

    def test_found_last(self):
        self.assertEqual(binary_search([10, 20, 30], 30, 0, 2),
           2)

    def test_not_found(self):
        self.assertEqual(binary_search([1, 2, 3, 4, 5], 6, 0, 4),
            -1)

    def test_empty_array(self):
        # For an empty array, low=0, high=-1 by convention
        self.assertEqual(binary_search([], 1, 0, -1), -1)

    def test_low_greater_than_high(self):
        self.assertEqual(binary_search([1, 2, 3], 2, 2, 1), -1)

    def test_duplicates(self):
        arr = [1, 2, 2, 2, 3]
        idx = binary_search(arr, 2, 0, 4)
        # Since there are multiple ’2’s, any of indices 1,2,3 is
           acceptable
        self.assertIn(idx, [1, 2, 3])

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 14: Test Code — GPTo4-mini-high

GEMINI 2.5 Response

Generated Code

```
# Binary search function (recursive implementation)
def binary_search(arr, x, low, high):
    """
    Performs a binary search on a sorted array.

    Args:
        arr: A sorted list of elements.
        x: The element to search for.
        low: The starting index of the portion of the array to
           search.
        high: The ending index of the portion of the array to
           search.

    Returns:
        The index of x in arr if found, otherwise -1.
    """
    # Check base case: if the search range is invalid
    if high < low:
        return -1  # Element not found

    # Calculate the middle index
    # Using (low + high) // 2 can lead to overflow if low and
       high are very large.
    # A safer way is low + (high - low) // 2
    mid = low + (high - low) // 2

    # Check if x is present at mid
    if arr[mid] == x:
        return mid
    # If x is smaller than mid, then it can only be present in
       the left subarray
    elif arr[mid] > x:
        return binary_search(arr, x, low, mid - 1)
    # Else the element can only be present in the right subarray
    else:
        return binary_search(arr, x, mid + 1, high)

def main():
    # Example usage of binary_search:

    arr = [1, 3, 5, 7, 9, 11]
    x = 5
    index = binary_search(arr, x, 0, len(arr) - 1)

if __name__ == "__main__":
    main()
```

Listing 15: Generated Code — GEMINI 2.5

Test Code

```
# --- Unit Tests (as provided by you) ---
import unittest

class TestBinarySearch(unittest.TestCase):
    def test_element_present_middle(self):
        """Test when element is in the middle of the array"""
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(binary_search(arr, 3, 0, len(arr) - 1),
           2)

    def test_element_present_start(self):
        """Test when element is at the start of the array"""
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(binary_search(arr, 1, 0, len(arr) - 1),
           0)

    def test_element_present_end(self):
        """Test when element is at the end of the array"""
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(binary_search(arr, 5, 0, len(arr) - 1),
           4)

    def test_element_not_present(self):
        """Test when element is not in the array"""
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(binary_search(arr, 6, 0, len(arr) - 1),
           -1)

    def test_empty_array(self):
        """Test with empty array"""
        arr = []
        # For an empty array, len(arr) - 1 is -1. So high will be
            -1, low will be 0.
        # The condition high < low (i.e., -1 < 0) will be true,
           returning -1.
        self.assertEqual(binary_search(arr, 1, 0, len(arr) - 1),
           -1)

    def test_single_element_found(self):
        """Test with single element array where element is
           present"""
        arr = [1]
        self.assertEqual(binary_search(arr, 1, 0, len(arr) - 1),
           0)

    def test_single_element_not_found(self):
        """Test with single element array where element is not
           present"""
        arr = [1]
        self.assertEqual(binary_search(arr, 2, 0, len(arr) - 1),
           -1)
```

```
    def test_invalid_range(self):
        """Test with invalid range (start > end)"""
        arr = [1, 2, 3, 4, 5]
        # Here low = 4, high = 2. The condition high < low (2 <
           4) is true.
        self.assertEqual(binary_search(arr, 3, 4, 2), -1)

    def test_large_array(self):
        """Test with larger array"""
        arr = list(range(0, 1000, 2))  # Even numbers 0 to 998
        self.assertEqual(binary_search(arr, 500, 0, len(arr) - 1)
           , 250)
        self.assertEqual(binary_search(arr, 501, 0, len(arr) - 1)
           , -1)

def main_test():
    # This will run the tests defined in TestBinarySearch
    unittest.main(argv=[’first-arg-is-ignored’], exit=False)

if __name__ == "__main__":
    # Run the unit tests
    print("\nRunning␣unit␣tests:")
    main_test()
```

Listing 16: Test Code — GEMINI 2.5

GROK 3 Response

Generated Code

```python
# Main Binary Search Implementation
def binary_search(arr, x, start, end):
    """
    Perform a recursive binary search to find the index of x in
       arr.
    Args:
        arr: Sorted list of numbers
        x: Value to search for
        start: Starting index of the search range
        end: Ending index of the search range
    Returns:
        Index of x if found, -1 otherwise
    """
    # Base case: if start > end, element not found
    if start > end:
        return -1

    # Calculate middle index
    mid = (start + end) // 2

    # If element is found at mid, return the index
    if arr[mid] == x:
        return mid

    # If element is smaller than mid, search left half
    elif arr[mid] > x:
        return binary_search(arr, x, start, mid - 1)

    # If element is larger than mid, search right half
    else:
        return binary_search(arr, x, mid + 1, end)

def main():
    # Example usage of binary search
    arr = [2, 3, 4, 10, 40, 50, 60, 70]
    x = 10
    result = binary_search(arr, x, 0, len(arr) - 1)
    print(f"Element␣{x}␣is␣{’present␣at␣index␣’␣+␣str(result)␣if␣
       result␣!=␣-1␣else␣’not␣present’}")

if __name__ == "__main__":
    main()
```

Listing 17: Generated Code — GROK 3

Test Code

```python
# Unit Tests
import unittest
```

```
class TestBinarySearch(unittest.TestCase):
    def test_element_present_middle(self):
        """Test when element is in the middle of the array"""
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(binary_search(arr, 3, 0, len(arr) - 1),
           2)

    def test_element_present_start(self):
        """Test when element is at the start of the array"""
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(binary_search(arr, 1, 0, len(arr) - 1),
           0)

    def test_element_present_end(self):
        """Test when element is at the end of the array"""
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(binary_search(arr, 5, 0, len(arr) - 1),
           4)

    def test_element_not_present(self):
        """Test when element is not in the array"""
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(binary_search(arr, 6, 0, len(arr) - 1),
           -1)

    def test_empty_array(self):
        """Test with empty array"""
        arr = []
        self.assertEqual(binary_search(arr, 1, 0, len(arr) - 1),
           -1)

    def test_single_element_found(self):
        """Test with single element array where element is
           present"""
        arr = [1]
        self.assertEqual(binary_search(arr, 1, 0, len(arr) - 1),
           0)

    def test_single_element_not_found(self):
        """Test with single element array where element is not
           present"""
        arr = [1]
        self.assertEqual(binary_search(arr, 2, 0, len(arr) - 1),
           -1)

    def test_invalid_range(self):
        """Test with invalid range (start > end)"""
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(binary_search(arr, 3, 4, 2), -1)
```

```
    def test_large_array(self):
        """Test with larger array"""
        arr = list(range(0, 1000, 2))  # Even numbers 0 to 998
        self.assertEqual(binary_search(arr, 500, 0, len(arr) - 1)
            , 250)
        self.assertEqual(binary_search(arr, 501, 0, len(arr) - 1)
            , -1)

def main_test():
    unittest.main(argv=[''], exit=False)

if __name__ == "__main__":
    main_test()
```

Listing 18: Test Code — GROK 3

Prompt #3: Binary Tree Search

Prompt:
You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked with developing a binary tree search algorithm. This search algorithm must receive as input the root node and the searched number (let's call it `key`). Let's use this to execute the program: `root = Node(50)`, `root.left = Node(30)`, `root.right = Node(70)`, `root.left.left = Node(20)`, `root.left.right = Node(40)`, `root.right.left = Node(60)`, `root.right.right = Node(80)`.
`print("Found" if search(root, 19) else "Not Found")`
`print("Found" if search(root, 80) else "Not Found")`.
Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone; they are run as follows: `exec(main_code.py, ns)`; `exec(test_code.py, ns)`. Each, the tests and script classes must have its own `if __name__ == "__main__": main()`.

Reference Code

```python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# function to search a key in a BST
def search(root, key):

    # Base Cases: root is null or key
    # is present at root
    if root is None or root.key == key:
        return root

    # Key is greater than root's key
    if root.key < key:
        return search(root.right, key)

    # Key is smaller than root's key
    return search(root.left, key)


 # Creating a hard coded tree for keeping
 # the length of the code small. We need
 # to make sure that BST properties are
 # maintained if we try some other cases.
root = Node(50)
```

```
root.left = Node(30)
root.right = Node(70)
root.left.left = Node(20)
root.left.right = Node(40)
root.right.left = Node(60)
root.right.right = Node(80)

# Searching for keys in the BST
print("Found" if search(root, 19) else "Not␣Found")
print("Found" if search(root, 80) else "Not␣Found")
```

Listing 19: Reference Code — GPT4o

GPT4o Response

Generated Code

```
# main_code.py

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def search(root, key):
    """
    Searches for a key in a Binary Search Tree (BST).
    Returns True if found, False otherwise.
    """
    if root is None:
        return False
    if root.key == key:
        return True
    elif key < root.key:
        return search(root.left, key)
    else:
        return search(root.right, key)

def main():
    # Constructing the binary search tree
    root = Node(50)
    root.left = Node(30)
    root.right = Node(70)
    root.left.left = Node(20)
    root.left.right = Node(40)
    root.right.left = Node(60)
    root.right.right = Node(80)

    # Example searches
    print("Found" if search(root, 19) else "Not␣Found")
    print("Found" if search(root, 80) else "Not␣Found")
```

```
if __name__ == "__main__":
    main()
```

Listing 20: Generated Code — GPT4o

Test Code

```
# test_code.py

import unittest

# Import main functions and classes

class TestBinarySearch(unittest.TestCase):

    def setUp(self):
        # Setup a common tree for all tests
        self.root = Node(50)
        self.root.left = Node(30)
        self.root.right = Node(70)
        self.root.left.left = Node(20)
        self.root.left.right = Node(40)
        self.root.right.left = Node(60)
        self.root.right.right = Node(80)

    def test_key_present_root(self):
        self.assertTrue(search(self.root, 50))

    def test_key_present_leaf(self):
        self.assertTrue(search(self.root, 20))
        self.assertTrue(search(self.root, 80))

    def test_key_present_middle(self):
        self.assertTrue(search(self.root, 30))
        self.assertTrue(search(self.root, 70))

    def test_key_absent_lower_than_all(self):
        self.assertFalse(search(self.root, 10))

    def test_key_absent_between_nodes(self):
        self.assertFalse(search(self.root, 35))
        self.assertFalse(search(self.root, 65))

    def test_key_absent_higher_than_all(self):
        self.assertFalse(search(self.root, 100))

    def test_empty_tree(self):
        self.assertFalse(search(None, 50))

def main():
    unittest.main()
```

```
if __name__ == "__main__":
    main()
```

Listing 21: Test Code — GPT4o

GPT o4-mini-high Response

Generated Code

```
# === main_code.py ===

class Node:
    def __init__(self, value):
        """
        Initialize a binary tree node.
        :param value: The value stored at this node.
        """
        self.value = value
        self.left = None
        self.right = None

def search(root, key):
    """
    Search for a key in a binary search tree.
    :param root: Node or None     the root of the BST.
    :param key: int     the value to search for.
    :return: bool     True if key is found, False otherwise.
    """
    if root is None:
        return False
    if key == root.value:
        return True
    elif key < root.value:
        return search(root.left, key)
    else:
        return search(root.right, key)

def main():
    """
    Build a sample BST and search for two keys, printing results.
    """
    # Construct the tree:
    #          50
    #        /    \
    #      30      70
    #     /  \    /  \
    #   20   40  60   80
    root = Node(50)
    root.left = Node(30)
    root.right = Node(70)
    root.left.left = Node(20)
    root.left.right = Node(40)
    root.right.left = Node(60)
    root.right.right = Node(80)

    # Search for keys and print outcome
    print("Found" if search(root, 19) else "Not␣Found")
    print("Found" if search(root, 80) else "Not␣Found")
```

```
if __name__ == "__main__":
    main()
```

Listing 22: Generated Code — GPTo4-mini-high

Test Code

```
# === test_code.py ===

import unittest

class TestBinaryTreeSearch(unittest.TestCase):
    def setUp(self):
        """
        Create a sample BST for testing:
                 50
                /  \
              30    70
             /  \   / \
           20   40 60  80
        """
        self.root = Node(50)
        self.root.left = Node(30)
        self.root.right = Node(70)
        self.root.left.left = Node(20)
        self.root.left.right = Node(40)
        self.root.right.left = Node(60)
        self.root.right.right = Node(80)

    def test_search_existing_root(self):
        self.assertTrue(search(self.root, 50))

    def test_search_existing_leaf_left(self):
        self.assertTrue(search(self.root, 20))

    def test_search_existing_leaf_right(self):
        self.assertTrue(search(self.root, 80))

    def test_search_existing_internal(self):
        self.assertTrue(search(self.root, 60))

    def test_search_nonexistent_less(self):
        self.assertFalse(search(self.root, 10))

    def test_search_nonexistent_between(self):
        self.assertFalse(search(self.root, 25))

    def test_search_nonexistent_greater(self):
        self.assertFalse(search(self.root, 90))

    def test_search_empty_tree(self):
```

```
        self.assertFalse(search(None, 50))

    def test_search_single_node_found(self):
        single = Node(42)
        self.assertTrue(search(single, 42))

    def test_search_single_node_not_found(self):
        single = Node(42)
        self.assertFalse(search(single, 100))

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 23: Test Code — GPTo4-mini-high

GEMINI 2.5 Response

Generated Code

```
# Main Code
class Node:
    """
    Represents a node in a binary tree.
    """
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

def search(root, key):
    """
    Searches for a key in a Binary Search Tree (BST).

    Args:
        root: The root node of the BST.
        key: The value to search for.

    Returns:
        True if the key is found in the BST, False otherwise.
    """
    if root is None:
        return False
    if root.val == key:
        return True
    if root.val < key:
        return search(root.right, key)
    return search(root.left, key)

def main():
    """
    Main function to demonstrate the BST search.
    """
    root = Node(50)
    root.left = Node(30)
    root.right = Node(70)
    root.left.left = Node(20)
    root.left.right = Node(40)
    root.right.left = Node(60)
    root.right.right = Node(80)

    # Searching for keys in the BST
    print("Found" if search(root, 19) else "Not␣Found")
    print("Found" if search(root, 80) else "Not␣Found")
    print("Found" if search(root, 50) else "Not␣Found") # Test
       root
    print("Found" if search(root, 20) else "Not␣Found") # Test
       leaf
```

```
    print("Found" if search(root, 60) else "Not␣Found") # Test
       internal node
    print("Found" if search(root, 100) else "Not␣Found") # Test
       not found (greater)
    print("Found" if search(root, 10) else "Not␣Found")  # Test
       not found (smaller)
    print("Found" if search(None, 10) else "Not␣Found") # Test
       empty tree

if __name__ == "__main__":
    main()
```

Listing 24: Generated Code — GEMINI 2.5

Test Code

```
# Unit Tests
import unittest

# If the Node class is not already defined in the scope where
   tests are run,
# uncomment the following lines or ensure it's imported/defined.
# class Node:
#     def __init__(self, key):
#         self.left = None
#         self.right = None
#         self.val = key
#
# def search(root, key):
#     if root is None:
#         return False
#     if root.val == key:
#         return True
#     if root.val < key:
#         return search(root.right, key)
#     return search(root.left, key)

class TestBinaryTreeSearch(unittest.TestCase):
    """
    Unit tests for the binary tree search algorithm.
    """
    def setUp(self):
        """
        Set up a sample BST for testing.
        """
        self.root = Node(50)
        self.root.left = Node(30)
        self.root.right = Node(70)
        self.root.left.left = Node(20)
        self.root.left.right = Node(40)
        self.root.right.left = Node(60)
        self.root.right.right = Node(80)
```

```
    def test_key_present(self):
        """Test searching for keys that are present in the BST.
           """
        self.assertTrue(search(self.root, 50), "Should␣find␣root␣
           node␣(50)")
        self.assertTrue(search(self.root, 30), "Should␣find␣left␣
           child␣(30)")
        self.assertTrue(search(self.root, 70), "Should␣find␣right
           ␣child␣(70)")
        self.assertTrue(search(self.root, 20), "Should␣find␣left-
           left␣grandchild␣(20)")
        self.assertTrue(search(self.root, 40), "Should␣find␣left-
           right␣grandchild␣(40)")
        self.assertTrue(search(self.root, 60), "Should␣find␣right
           -left␣grandchild␣(60)")
        self.assertTrue(search(self.root, 80), "Should␣find␣right
           -right␣grandchild␣(80)")

    def test_key_not_present(self):
        """Test searching for keys that are not present in the
           BST."""
        self.assertFalse(search(self.root, 10), "Should␣not␣find␣
           10␣(smaller␣than␣min)")
        self.assertFalse(search(self.root, 90), "Should␣not␣find␣
           90␣(larger␣than␣max)")
        self.assertFalse(search(self.root, 35), "Should␣not␣find␣
           35␣(intermediate␣missing)")
        self.assertFalse(search(self.root, 55), "Should␣not␣find␣
           55␣(intermediate␣missing)")
        self.assertFalse(search(self.root, 75), "Should␣not␣find␣
           75␣(intermediate␣missing)")
        self.assertFalse(search(self.root, 19), "Should␣not␣find␣
           19␣(as␣per␣example)")


    def test_empty_tree(self):
        """Test searching in an empty tree."""
        self.assertFalse(search(None, 50), "Should␣not␣find␣any␣
           key␣in␣an␣empty␣tree")
        self.assertFalse(search(None, 0), "Should␣not␣find␣0␣in␣
           an␣empty␣tree")

    def test_single_node_tree(self):
        """Test searching in a tree with a single node."""
        single_node_root = Node(100)
        self.assertTrue(search(single_node_root, 100), "Should␣
           find␣the␣key␣in␣a␣single␣node␣tree")
        self.assertFalse(search(single_node_root, 50), "Should␣
           not␣find␣a␣different␣key␣in␣a␣single␣node␣tree")
```

```
    def test_left_skewed_tree(self):
        """Test searching in a left-skewed tree."""
        root_skewed = Node(50)
        root_skewed.left = Node(40)
        root_skewed.left.left = Node(30)
        root_skewed.left.left.left = Node(20)
        self.assertTrue(search(root_skewed, 20), "Should find 20 
           in left-skewed tree")
        self.assertTrue(search(root_skewed, 50), "Should find 50 
           in left-skewed tree")
        self.assertFalse(search(root_skewed, 60), "Should not 
           find 60 in left-skewed tree")
        self.assertFalse(search(root_skewed, 10), "Should not 
           find 10 in left-skewed tree")

    def test_right_skewed_tree(self):
        """Test searching in a right-skewed tree."""
        root_skewed = Node(50)
        root_skewed.right = Node(60)
        root_skewed.right.right = Node(70)
        root_skewed.right.right.right = Node(80)
        self.assertTrue(search(root_skewed, 80), "Should find 80 
           in right-skewed tree")
        self.assertTrue(search(root_skewed, 50), "Should find 50 
           in right-skewed tree")
        self.assertFalse(search(root_skewed, 40), "Should not 
           find 40 in right-skewed tree")
        self.assertFalse(search(root_skewed, 90), "Should not 
           find 90 in right-skewed tree")

def main_tests():
    """
    Main function to run the unit tests.
    """
    unittest.main(argv=['first-arg-is-ignored'], exit=False)

if __name__ == "__main__":
    # This part is primarily for the test script execution.
    # When the evaluator runs `exec(test_code.py, ns)`,
    # it will look for a `main` function in that context.
    # We rename it to main_tests to avoid conflict if both
       scripts are
    # somehow run in a way that their __main__ blocks get
       triggered together.
    # However, given the `exec` pattern, it's better to have a
       distinct main for tests.
    main_tests()
```

Listing 25: Test Code — GEMINI 2.5

GROK 3 Response

Generated Code

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def search(root, key):
    # Base cases: root is None or we found the key
    if root is None or root.value == key:
        return root is not None and root.value == key

    # If key is less than root, search left subtree
    if key < root.value:
        return search(root.left, key)

    # If key is greater than root, search right subtree
    return search(root.right, key)

def main():
    # Create the binary search tree
    root = Node(50)
    root.left = Node(30)
    root.right = Node(70)
    root.left.left = Node(20)
    root.left.right = Node(40)
    root.right.left = Node(60)
    root.right.right = Node(80)

    # Test the search function
    print("Found" if search(root, 19) else "Not␣Found")  # Should
        print "Not Found"
    print("Found" if search(root, 80) else "Not␣Found")  # Should
        print "Found"

if __name__ == "__main__":
    main()
```

Listing 26: Generated Code — GROK 3

Test Code

```
import unittest

class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

```
def search(root, key):
    # Base cases: root is None or we found the key
    if root is None or root.value == key:
        return root is not None and root.value == key

    # If key is less than root, search left subtree
    if key < root.value:
        return search(root.left, key)

    # If key is greater than root, search right subtree
    return search(root.right, key)

class TestBinarySearchTree(unittest.TestCase):
    def setUp(self):
        # Create a sample BST for testing
        self.root = Node(50)
        self.root.left = Node(30)
        self.root.right = Node(70)
        self.root.left.left = Node(20)
        self.root.left.right = Node(40)
        self.root.right.left = Node(60)
        self.root.right.right = Node(80)

    def test_search_existing_key(self):
        """Test searching for keys that exist in the tree"""
        self.assertTrue(search(self.root, 50))  # Root node
        self.assertTrue(search(self.root, 20))  # Leaf node
        self.assertTrue(search(self.root, 70))  # Internal node
        self.assertTrue(search(self.root, 80))  # Leaf node

    def test_search_non_existing_key(self):
        """Test searching for keys that don't exist in the tree
           """
        self.assertFalse(search(self.root, 19))  # Less than
           smallest
        self.assertFalse(search(self.root, 90))  # Greater than
           largest
        self.assertFalse(search(self.root, 35))  # In between
           existing nodes

    def test_search_empty_tree(self):
        """Test searching in an empty tree"""
        self.assertFalse(search(None, 50))

    def test_search_single_node_tree(self):
        """Test searching in a tree with only one node"""
        root = Node(100)
        self.assertTrue(search(root, 100))
        self.assertFalse(search(root, 101))

    def test_search_left_subtree(self):
```

```
        """Test searching in left subtree specifically"""
        self.assertTrue(search(self.root, 30))
        self.assertTrue(search(self.root, 20))
        self.assertTrue(search(self.root, 40))
        self.assertFalse(search(self.root, 25))

    def test_search_right_subtree(self):
        """Test searching in right subtree specifically"""
        self.assertTrue(search(self.root, 70))
        self.assertTrue(search(self.root, 60))
        self.assertTrue(search(self.root, 80))
        self.assertFalse(search(self.root, 75))

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 27: Test Code — GROK 3

Prompt #4: Calculator

Prompt:
You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked with developing a simple calculator using Python. The calculator should allow the following operations: Add, Subtract, Multiply, and Divide. Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone; they are run as follows: `exec(main_code.py, ns)`; `exec(test_code.py, ns)`. Each, the tests and script classes must have its own `if __name__ == "__main__": main()`.

Reference Code

```
# Define functions for each operation
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    # Prevent division by zero
    if y == 0:
        return "Error:␣Division␣by␣zero␣is␣not␣allowed"
    return x / y

# Display the options to the user
def display_menu():
    print("\nSelect␣operation:")
    print("1.␣Add")
    print("2.␣Subtract")
    print("3.␣Multiply")
    print("4.␣Divide")

# Main program loop
while True:
    display_menu()

    # Take input from the user
    choice = input("Enter␣choice␣(1/2/3/4):␣")

    # Check if the choice is valid
```

```
    if choice in (’1’, ’2’, ’3’, ’4’):
        try:
            num1 = float(input("Enter␣first␣number:␣"))
            num2 = float(input("Enter␣second␣number:␣"))
        except ValueError:
            print("Invalid␣input.␣Please␣enter␣numeric␣values.")
            continue

        # Perform the chosen operation
        if choice == ’1’:
            print(f"{num1}␣+␣{num2}␣=␣{add(num1,␣num2)}")
        elif choice == ’2’:
            print(f"{num1}␣-␣{num2}␣=␣{subtract(num1,␣num2)}")
        elif choice == ’3’:
            print(f"{num1}␣*␣{num2}␣=␣{multiply(num1,␣num2)}")
        elif choice == ’4’:
            print(f"{num1}␣/␣{num2}␣=␣{divide(num1,␣num2)}")

        # Check if the user wants another calculation
        next_calculation = input("Do␣you␣want␣to␣perform␣another␣
           calculation?␣(yes/no):␣").lower()
        if next_calculation != ’yes’:
            print("Exiting␣the␣calculator.")
            break
    else:
        print("Invalid␣input.␣Please␣select␣a␣valid␣operation.")
```

Listing 28: Reference Code — GPT4o

GPT4o Response

Generated Code

```
# ==========================
# Calculator Implementation
# ==========================

class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot␣divide␣by␣zero.")
        return a / b

```

```
def main():
    calc = Calculator()

    print("Simple␣Calculator")
    print("Choose␣operation:␣add,␣subtract,␣multiply,␣divide")
    operation = input("Operation:␣").strip().lower()

    try:
        a = float(input("Enter␣first␣number:␣"))
        b = float(input("Enter␣second␣number:␣"))
    except ValueError:
        print("Invalid␣number␣input.")
        return

    try:
        if operation == "add":
            result = calc.add(a, b)
        elif operation == "subtract":
            result = calc.subtract(a, b)
        elif operation == "multiply":
            result = calc.multiply(a, b)
        elif operation == "divide":
            result = calc.divide(a, b)
        else:
            print("Invalid␣operation␣selected.")
            return

        print(f"Result:␣{result}")
    except ValueError as e:
        print(f"Error:␣{e}")


if __name__ == "__main__":
    main()
```

Listing 29: Generated Code — GPT4o

Test Code

```
# ==========================
# Unit Tests
# ==========================

import unittest

class TestCalculator(unittest.TestCase):
    def setUp(self):
        self.calc = Calculator()

    def test_add(self):
        self.assertEqual(self.calc.add(2, 3), 5)
```

```
        self.assertEqual(self.calc.add(-1, -1), -2)
        self.assertEqual(self.calc.add(0, 0), 0)

    def test_subtract(self):
        self.assertEqual(self.calc.subtract(10, 5), 5)
        self.assertEqual(self.calc.subtract(0, 0), 0)
        self.assertEqual(self.calc.subtract(-3, -7), 4)

    def test_multiply(self):
        self.assertEqual(self.calc.multiply(4, 5), 20)
        self.assertEqual(self.calc.multiply(-2, 3), -6)
        self.assertEqual(self.calc.multiply(0, 100), 0)

    def test_divide(self):
        self.assertEqual(self.calc.divide(10, 2), 5)
        self.assertAlmostEqual(self.calc.divide(5, 3),
           1.6666666666666667)
        with self.assertRaises(ValueError):
            self.calc.divide(5, 0)


def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 30: Test Code — GPT4o

GPT o4-mini-high Response

Generated Code

```python
# calculator.py

class Calculator:
    """Simple calculator supporting add, subtract, multiply, and divide."""

    def add(self, a, b):
        """Return the sum of a and b."""
        return a + b

    def subtract(self, a, b):
        """Return the difference a - b."""
        return a - b

    def multiply(self, a, b):
        """Return the product of a and b."""
        return a * b

    def divide(self, a, b):
        """Return a divided by b; raises ValueError on division by zero."""
        if b == 0:
            raise ValueError("Cannot␣divide␣by␣zero.")
        return a / b


def main():
    """Demo of Calculator functionality."""
    calc = Calculator()
    print("2␣+␣3␣=", calc.add(2, 3))
    print("5␣-␣2␣=", calc.subtract(5, 2))
    print("3␣*␣4␣=", calc.multiply(3, 4))
    print("10␣/␣2␣=", calc.divide(10, 2))

if __name__ == "__main__":
    main()
```

Listing 31: Generated Code — GPTo4-mini-high

Test Code

```python
import unittest

class TestCalculator(unittest.TestCase):
    """Unit tests for Calculator."""

    def setUp(self):
        # Assumes Calculator is already defined in the execd namespace
```

```python
        self.calc = Calculator()

    def test_add(self):
        self.assertEqual(self.calc.add(2, 3), 5)
        self.assertEqual(self.calc.add(-1, -1), -2)
        self.assertAlmostEqual(self.calc.add(2.5, 0.5), 3.0)

    def test_subtract(self):
        self.assertEqual(self.calc.subtract(5, 2), 3)
        self.assertEqual(self.calc.subtract(0, 5), -5)
        self.assertAlmostEqual(self.calc.subtract(2.5, 1.0), 1.5)

    def test_multiply(self):
        self.assertEqual(self.calc.multiply(3, 4), 12)
        self.assertEqual(self.calc.multiply(-2, 3), -6)
        self.assertAlmostEqual(self.calc.multiply(2.5, 2), 5.0)

    def test_divide(self):
        self.assertEqual(self.calc.divide(10, 2), 5)
        self.assertAlmostEqual(self.calc.divide(5, 2), 2.5)
        self.assertAlmostEqual(self.calc.divide(-9, 3), -3.0)

    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            self.calc.divide(5, 0)


def main():
    unittest.main(verbosity=2)

if __name__ == "__main__":
    main()
```

Listing 32: Test Code — GPTo4-mini-high

GEMINI 2.5 Response

Generated Code

```
# main_code.py
# ------------
# This section contains the main calculator implementation.

import operator

class Calculator:
    """
    A simple calculator class that performs basic arithmetic
        operations.
    """

    def add(self, x: float, y: float) -> float:
        """Adds two numbers."""
        return operator.add(x, y)

    def subtract(self, x: float, y: float) -> float:
        """Subtracts the second number from the first."""
        return operator.sub(x, y)

    def multiply(self, x: float, y: float) -> float:
        """Multiplies two numbers."""
        return operator.mul(x, y)

    def divide(self, x: float, y: float) -> float:
        """
        Divides the first number by the second.
        Raises ValueError if the divisor is zero.
        """
        if y == 0:
            raise ValueError("Cannot␣divide␣by␣zero")
        return operator.truediv(x, y)

def main():
    """Main function for the calculator script (demonstration).
        """
    print("Calculator␣Module␣Loaded.")
    # Example usage (optional demonstration)
    # calc = Calculator()
    # print(f"5 + 3 = {calc.add(5, 3)}")
    # print(f"5 - 3 = {calc.subtract(5, 3)}")
    # print(f"5 * 3 = {calc.multiply(5, 3)}")
    # try:
    #     print(f"5 / 3 = {calc.divide(5, 3)}")
    #     print(f"5 / 0 = {calc.divide(5, 0)}") # This will raise
        ValueError
    # except ValueError as e:
    #     print(e)
```

```
# Main execution block for the calculator script
if __name__ == "__main__":
    main()
```

Listing 33: Generated Code — GEMINI 2.5

Test Code

```
# ==========================
# Unit Tests
# ==========================

# ==================================================
#                  UNIT TEST SECTION
# ==================================================
# test_code.py
# ------------
# This section contains the unit tests for the Calculator class.

import unittest
import sys
import io # Required to capture print output if main() prints
   anything

# Note: The Calculator class is implicitly available here because
# the execution environment runs both code blocks in the same
   namespace.
# No explicit import like 'from __main__ import Calculator' is
   needed
# when run via `exec(main_code.py, ns); exec(test_code.py, ns)`.

class TestCalculator(unittest.TestCase):
    """Unit tests for the Calculator class."""

    def setUp(self):
        """Set up a Calculator instance before each test."""
        # Since Calculator is defined in the same script
           execution context,
        # we can instantiate it directly.
        self.calculator = Calculator()

    def test_add(self):
        """Test the add method."""
        self.assertEqual(self.calculator.add(5, 3), 8)
        self.assertEqual(self.calculator.add(-1, 1), 0)
        self.assertEqual(self.calculator.add(-1, -1), -2)
        self.assertEqual(self.calculator.add(0, 0), 0)
        self.assertAlmostEqual(self.calculator.add(0.1, 0.2),
           0.3)

    def test_subtract(self):
        """Test the subtract method."""
```

```
        self.assertEqual(self.calculator.subtract(10, 4), 6)
        self.assertEqual(self.calculator.subtract(-1, 1), -2)
        self.assertEqual(self.calculator.subtract(-1, -1), 0)
        self.assertEqual(self.calculator.subtract(0, 0), 0)
        self.assertEqual(self.calculator.subtract(5, 10), -5)
        self.assertAlmostEqual(self.calculator.subtract(0.3, 0.1)
           , 0.2)

    def test_multiply(self):
        """Test the multiply method."""
        self.assertEqual(self.calculator.multiply(3, 7), 21)
        self.assertEqual(self.calculator.multiply(-1, 1), -1)
        self.assertEqual(self.calculator.multiply(-1, -1), 1)
        self.assertEqual(self.calculator.multiply(0, 100), 0)
        self.assertEqual(self.calculator.multiply(100, 0), 0)
        self.assertAlmostEqual(self.calculator.multiply(0.5, 0.5)
           , 0.25)

    def test_divide(self):
        """Test the divide method."""
        self.assertEqual(self.calculator.divide(10, 2), 5)
        self.assertEqual(self.calculator.divide(-10, 2), -5)
        self.assertEqual(self.calculator.divide(-10, -2), 5)
        self.assertEqual(self.calculator.divide(0, 5), 0)
        self.assertAlmostEqual(self.calculator.divide(1, 3),
           0.3333333333333333)
        self.assertAlmostEqual(self.calculator.divide(5, 2), 2.5)

    def test_divide_by_zero(self):
        """Test that division by zero raises a ValueError."""
        with self.assertRaises(ValueError) as cm:
            self.calculator.divide(10, 0)
        self.assertEqual(str(cm.exception), "Cannot␣divide␣by␣
           zero")

        with self.assertRaises(ValueError) as cm:
            self.calculator.divide(0, 0)
        self.assertEqual(str(cm.exception), "Cannot␣divide␣by␣
           zero")

        with self.assertRaises(ValueError) as cm:
            self.calculator.divide(-5, 0)
        self.assertEqual(str(cm.exception), "Cannot␣divide␣by␣
           zero")

def test_main():
    """Main function for running the unit tests."""
    print("\nRunning␣Unit␣Tests...")
    # Redirect stdout to avoid clutter from unittest's default
       runner output
```

```
    # if we just want a clean pass/fail indication or custom
       summary.
    # However, for standard execution, unittest.main() handles
       output well.
    # Using buffer=True suppresses print output during test runs
       unless a test fails.
    suite = unittest.TestSuite()
    suite.addTest(unittest.makeSuite(TestCalculator))
    runner = unittest.TextTestRunner(verbosity=2, buffer=True)
    result = runner.run(suite)
    if result.wasSuccessful():
        print("All␣tests␣passed␣successfully.")
    else:
        print("Some␣tests␣failed.")
        # Exit with a non-zero code if tests fail, useful for CI/
           CD
        sys.exit(1)


# Main execution block for the unit tests
if __name__ == "__main__":
    # When this entire script is run directly, both main()
       functions
    # (calculator's and test's) might be called depending on how
       it's run.
    # However, the instructions specify separate execution
       contexts simulation
    # via exec(). In that scenario, only the second __main__
       block
    # (the test block's main) would effectively run after the
       first exec call
    # defines the Calculator class.

    # To ensure only tests run if the *entire file* is executed
       as the main script:
    # Option 1: Rely on the exec() separation as described.
    # Option 2: Add logic to differentiate, e.g., using command-
       line args,
    #           but that complicates the self-contained
       requirement.
    # Option 3: Structure so the test main function is the
       primary entry point
    #           when the script is run as a whole.

    # Let's assume the exec() method described is used. In that
       case,
    # the first `if __name__ == "__main__": main()` will run
       during the first exec
    # and the second `if __name__ == "__main__": test_main()`
       will run during the second exec.
```

```
    # If running the *entire file* directly (e.g., `python
       your_script.py`),
    # the behavior depends on Python's execution flow. Typically,
        the first
    # `if __name__ == "__main__"` block executes, then the
       definitions below it
    # are processed, and finally the *second* `if __name__ == "
       __main__"` block
    # also executes because `__name__` is still "__main__".
    # To prevent the calculator's main() from running again when
       the test main()
    # is the intended entry point for testing, we might rename
       the test main
    # or add conditional logic.

    # Given the `exec` context, we'll keep both distinct main
       blocks.
    # The second block below is intended to be run by the test
       execution step.
    test_main()
```

Listing 34: Test Code — GEMINI 2.5

GROK 3 Response

Generated Code

```
# --- Main Calculator Script ---

class Calculator:
    def add(self, a: float, b: float) -> float:
        """Add two numbers."""
        return a + b

    def subtract(self, a: float, b: float) -> float:
        """Subtract b from a."""
        return a - b

    def multiply(self, a: float, b: float) -> float:
        """Multiply two numbers."""
        return a * b

    def divide(self, a: float, b: float) -> float:
        """Divide a by b. Raises ValueError if b is zero."""
        if b == 0:
            raise ValueError("Division␣by␣zero␣is␣not␣allowed")
        return a / b

def main():
    """Main function to demonstrate calculator usage."""
    calc = Calculator()
    print(f"Add:␣{calc.add(10,␣5)}")
    print(f"Subtract:␣{calc.subtract(10,␣5)}")
    print(f"Multiply:␣{calc.multiply(10,␣5)}")
    print(f"Divide:␣{calc.divide(10,␣5)}")

if __name__ == "__main__":
    main()
```

Listing 35: Generated Code — GROK 3

Test Code

```
import unittest

class TestCalculator(unittest.TestCase):
    def setUp(self):
        """Set up a Calculator instance for each test."""
        self.calc = Calculator()

    def test_add_positive_numbers(self):
        """Test adding two positive numbers."""
        self.assertEqual(self.calc.add(3, 4), 7)

    def test_add_negative_numbers(self):
        """Test adding two negative numbers."""
```

```
        self.assertEqual(self.calc.add(-2, -3), -5)

    def test_add_zero(self):
        """Test adding a number and zero."""
        self.assertEqual(self.calc.add(5, 0), 5)

    def test_subtract_positive_numbers(self):
        """Test subtracting two positive numbers."""
        self.assertEqual(self.calc.subtract(7, 4), 3)

    def test_subtract_negative_result(self):
        """Test subtraction yielding a negative result."""
        self.assertEqual(self.calc.subtract(2, 5), -3)

    def test_subtract_zero(self):
        """Test subtracting zero from a number."""
        self.assertEqual(self.calc.subtract(5, 0), 5)

    def test_multiply_positive_numbers(self):
        """Test multiplying two positive numbers."""
        self.assertEqual(self.calc.multiply(3, 4), 12)

    def test_multiply_by_zero(self):
        """Test multiplying a number by zero."""
        self.assertEqual(self.calc.multiply(5, 0), 0)

    def test_multiply_negative_numbers(self):
        """Test multiplying two negative numbers."""
        self.assertEqual(self.calc.multiply(-2, -3), 6)

    def test_divide_positive_numbers(self):
        """Test dividing two positive numbers."""
        self.assertEqual(self.calc.divide(8, 2), 4)

    def test_divide_by_negative(self):
        """Test dividing by a negative number."""
        self.assertEqual(self.calc.divide(6, -2), -3)

    def test_divide_zero_by_number(self):
        """Test dividing zero by a non-zero number."""
        self.assertEqual(self.calc.divide(0, 5), 0)

    def test_divide_by_zero(self):
        """Test division by zero raises ValueError."""
        with self.assertRaises(ValueError):
            self.calc.divide(10, 0)

    def test_divide_floating_point(self):
        """Test division resulting in a floating-point number."""
        self.assertAlmostEqual(self.calc.divide(5, 2), 2.5)
```

```
def main():
    """Main function to run the unit tests."""
    unittest.main(argv=[''], exit=False)

if __name__ == "__main__":
    main()
```

Listing 36: Test Code — GROK 3

Prompt #5: DFS Search

Prompt:
You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked with developing a Depth First Search (DFS) from a given source of an undirected graph. Let's use this to execute the program:# Driver code V = 5 adj = [[0] * V for _ in range(V)] # Adjacency matrix edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)] # Define the edges of the graph for s, t in edges: add_edge(adj, s, t) res = DFS(adj) # Perform DFS print(" ".join(map(str, res))) Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone; they are run as follows: `exec(main_code.py, ns)`; `exec(test_code.py, ns)`. Each, the tests and script classes must have its own `if __name__ == "__main__": main()`.

Reference Code

```python
def dfsRec(adj, visited, s, res):
    visited[s] = True
    res.append(s)

    # Recursively visit all adjacent vertices that are not
       visited yet
    for i in range(len(adj)):
        if adj[s][i] == 1 and not visited[i]:
            dfsRec(adj, visited, i, res)


def DFS(adj):
    visited = [False] * len(adj)
    res = []
    dfsRec(adj, visited, 0, res)  # Start DFS from vertex 0
    return res


def add_edge(adj, s, t):
    adj[s][t] = 1
    adj[t][s] = 1  # Since it's an undirected graph


# Driver code
V = 5
adj = [[0] * V for _ in range(V)]  # Adjacency matrix

# Define the edges of the graph
edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]
```

```
# Populate the adjacency matrix with edges
for s, t in edges:
    add_edge(adj, s, t)

res = DFS(adj)  # Perform DFS
print("␣".join(map(str, res)))
```

Listing 37: Reference Code — GPT4o

GPT4o Response

Generated Code

```
# main_code.py

def add_edge(adj, u, v):
    adj[u][v] = 1
    adj[v][u] = 1  # Since the graph is undirected

def DFSUtil(v, visited, adj, result):
    visited[v] = True
    result.append(v)
    for i in range(len(adj)):
        if adj[v][i] == 1 and not visited[i]:
            DFSUtil(i, visited, adj, result)

def DFS(adj, start=0):
    V = len(adj)
    visited = [False] * V
    result = []
    DFSUtil(start, visited, adj, result)
    return result

def main():
    V = 5
    adj = [[0] * V for _ in range(V)]  # Adjacency matrix
    edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]

    for s, t in edges:
        add_edge(adj, s, t)

    res = DFS(adj)
    print("␣".join(map(str, res)))

if __name__ == "__main__":
    main()
```

Listing 38: Generated Code — GPT4o

Test Code

```
# test_code.py
```

```python
import unittest

class TestDFS(unittest.TestCase):
    def test_dfs_traversal_from_zero(self):
        V = 5
        adj = [[0] * V for _ in range(V)]
        edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]
        for s, t in edges:
            add_edge(adj, s, t)

        expected_nodes = set([0, 1, 2, 3, 4])
        result = DFS(adj, start=0)
        self.assertEqual(set(result), expected_nodes)
        self.assertTrue(result[0] == 0)

    def test_dfs_traversal_from_two(self):
        V = 5
        adj = [[0] * V for _ in range(V)]
        edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]
        for s, t in edges:
            add_edge(adj, s, t)

        result = DFS(adj, start=2)
        self.assertIn(2, result)
        self.assertEqual(len(result), 5)

    def test_dfs_disconnected_graph(self):
        V = 6
        adj = [[0] * V for _ in range(V)]
        edges = [(0, 1), (0, 2), (3, 4)]
        for s, t in edges:
            add_edge(adj, s, t)

        result = DFS(adj, start=0)
        self.assertEqual(set(result), set([0, 1, 2]))

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 39: Test Code — GPT4o

GPT o4-mini-high Response

Generated Code

```
# main_code.py

def add_edge(adj, u, v):
    """
    Add an undirected edge between u and v in the adjacency
       matrix adj.
    """
    adj[u][v] = 1
    adj[v][u] = 1

def DFS(adj, start=0):
    """
    Perform a depth-first search on an undirected graph
       represented by
    adjacency matrix ‘adj‘, starting from vertex ‘start‘.
    Returns the list of vertices in the order they are visited.
    """
    V = len(adj)
    visited = [False] * V
    order = []

    def dfs(v):
        visited[v] = True
        order.append(v)
        for u, connected in enumerate(adj[v]):
            if connected and not visited[u]:
                dfs(u)

    dfs(start)
    return order

def main():
    # Example driver code
    V = 5
    adj = [[0] * V for _ in range(V)]
    edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]
    for s, t in edges:
        add_edge(adj, s, t)
    res = DFS(adj)
    print("␣".join(map(str, res)))

if __name__ == "__main__":
    main()
```

Listing 40: Generated Code — GPTo4-mini-high

Test Code

```
# test_code.py
```

```
import unittest

class TestDFS(unittest.TestCase):
    def test_add_edge(self):
        adj = [[0]*3 for _ in range(3)]
        add_edge(adj, 0, 2)
        self.assertEqual(adj[0][2], 1)
        self.assertEqual(adj[2][0], 1)

    def test_dfs_simple(self):
        V = 5
        adj = [[0]*V for _ in range(V)]
        edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]
        for s, t in edges:
            add_edge(adj, s, t)
        result = DFS(adj)
        # Starting at 0, DFS should visit 0->1->2->3->4
        self.assertEqual(result, [0, 1, 2, 3, 4])

    def test_dfs_single_node(self):
        adj = [[0]]
        result = DFS(adj)
        self.assertEqual(result, [0])

    def test_dfs_disconnected(self):
        V = 4
        adj = [[0]*V for _ in range(V)]
        # No edges at all
        result = DFS(adj)
        # Only the start node (0) is reachable
        self.assertEqual(result, [0])

    def test_dfs_cycle(self):
        # Graph: 0-1-2-0 forms a cycle
        V = 3
        adj = [[0]*V for _ in range(V)]
        edges = [(0,1),(1,2),(2,0)]
        for s,t in edges:
            add_edge(adj, s, t)
        result = DFS(adj)
        # Expect to traverse each exactly once: 0->1->2
        self.assertEqual(result, [0,1,2])

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 41: Test Code — GPTo4-mini-high

GEMINI 2.5 Response

Generated Code

```
# main_code.py

def add_edge(adj, s, t):
    """
    Adds an undirected edge between vertex s and vertex t
    to the adjacency matrix adj.
    """
    adj[s][t] = 1
    adj[t][s] = 1

def _dfs_util(u, adj, visited, dfs_order_list):
    """
    Recursive helper function for DFS.

    Args:
        u (int): The current vertex to visit.
        adj (list[list[int]]): The adjacency matrix of the graph.
        visited (list[bool]): A list keeping track of visited
           vertices.
        dfs_order_list (list[int]): A list to store the DFS
           traversal order.
    """
    visited[u] = True
    dfs_order_list.append(u)

    # Explore all adjacent vertices
    for v in range(len(adj[u])): # Iterate through potential
       neighbors
        if adj[u][v] == 1 and not visited[v]:
            _dfs_util(v, adj, visited, dfs_order_list)

def DFS(adj, source_node=0):
    """
    Performs Depth First Search on the graph represented by the
       adjacency matrix adj,
    starting from the given source_node.

    If no source_node is provided, it defaults to 0.

    Args:
        adj (list[list[int]]): The adjacency matrix of the graph.
        source_node (int, optional): The starting node for DFS.
           Defaults to 0.

    Returns:
        list[int]: A list of vertices in DFS traversal order,
           starting from source_node.
                   Returns an empty list if the graph is empty (
                      has 0 vertices).
```

```
    Raises:
        ValueError: If source_node is out of bounds for the given
            graph.
    """
    V = len(adj)

    if V == 0:
        return []  # Handle empty graph (no vertices)

    if not (0 <= source_node < V):
        raise ValueError("Source␣node␣is␣out␣of␣bounds.")

    visited = [False] * V
    dfs_order_list = []

    _dfs_util(source_node, adj, visited, dfs_order_list)

    return dfs_order_list

def main():
    """
    Driver code to demonstrate DFS functionality.
    """
    # Driver code as provided in the problem description
    V = 5
    adj = [[0] * V for _ in range(V)]

    # Define the edges of the graph
    edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]

    # Populate the adjacency matrix with edges
    for s, t in edges:
        add_edge(adj, s, t)

    # Perform DFS. Since the original driver was DFS(adj),
    # we assume it implies starting from a default source, e.g.,
       0.
    # Our DFS function uses source_node=0 by default.
    res = DFS(adj)
    print("␣".join(map(str, res)))

if __name__ == "__main__":
    main()
```

Listing 42: Generated Code — GEMINI 2.5

Test Code

```
# test_code.py
import unittest
```

```
# The functions add_edge, _dfs_util, and DFS are expected to be
   in the
# global namespace when this script is executed by the evaluator
# after main_code.py.

class TestDFS(unittest.TestCase):

    def test_add_edge(self):
        V = 3
        adj = [[0] * V for _ in range(V)]
        add_edge(adj, 0, 1)
        self.assertEqual(adj[0][1], 1, "Edge␣0-1␣not␣created␣(adj
           [0][1])")
        self.assertEqual(adj[1][0], 1, "Edge␣0-1␣not␣created␣(adj
           [1][0])")
        self.assertEqual(adj[0][2], 0, "Edge␣0-2␣should␣not␣exist
           ␣yet")

        add_edge(adj, 0, 2)
        self.assertEqual(adj[0][2], 1, "Edge␣0-2␣not␣created␣(adj
           [0][2])")
        self.assertEqual(adj[2][0], 1, "Edge␣0-2␣not␣created␣(adj
           [2][0])")

    def test_dfs_empty_graph(self):
        adj_empty = []
        # DFS(adj_empty) will call DFS(adj_empty, 0)
        # The DFS function handles V=0 by returning [] before
           checking source_node.
        res_empty = DFS(adj_empty)
        self.assertEqual(res_empty, [], "DFS␣on␣empty␣graph␣
           should␣return␣[]")

    def test_dfs_single_node_graph(self):
        adj_single_node = [[0]]
        res_s0 = DFS(adj_single_node, 0)
        self.assertEqual(res_s0, [0], "DFS␣on␣single␣node␣graph␣
           from␣source␣0")

    def test_dfs_from_problem_description(self):
        V = 5
        adj = [[0] * V for _ in range(V)]
        edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]
        for s, t in edges:
            add_edge(adj, s, t)

        # DFS(adj) will use default source_node=0
        res = DFS(adj)
        # Based on iteration order (0 to V-1 for neighbors):
        # 0 -> 1 (0's first unvisited neighbor)
        # 1 -> 2 (1's first unvisited neighbor, 0 is visited)
```

```
        # 2 -> 3 (2's first unvisited neighbor, 0,1 are visited)
        # 3 -> (no unvisited neighbors)
        # 2 -> 4 (2's next unvisited neighbor)
        # 4 -> (no unvisited neighbors)
        self.assertEqual(res, [0, 1, 2, 3, 4], "DFS␣from␣problem␣
           description␣(source␣0)")

    def test_dfs_example_graph_different_source(self):
        V = 5
        adj = [[0] * V for _ in range(V)]
        edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]
        for s, t in edges:
            add_edge(adj, s, t)

        res_source_3 = DFS(adj, 3)
        # Expected from source 3 (iterating neighbors 0 to V-1):
        # 3 -> 2
        # 2 -> 0
        # 0 -> 1
        # 1 -> (no unvisited connected)
        # 0 -> (no unvisited connected)
        # 2 -> 4
        # 4 -> (no unvisited connected)
        self.assertEqual(res_source_3, [3, 2, 0, 1, 4], "DFS␣from
           ␣source␣3")

    def test_dfs_disconnected_graph_source_in_component1(self):
        V = 5
        adj = [[0] * V for _ in range(V)]
        # Component 1: 0-1, 1-2
        add_edge(adj, 0, 1)
        add_edge(adj, 1, 2)
        # Component 2: 3-4
        add_edge(adj, 3, 4)

        res_source_0 = DFS(adj, 0)
        # Expected: 0 -> 1 -> 2
        self.assertEqual(sorted(res_source_0), sorted([0, 1, 2]),
            "DFS␣from␣source␣0␣on␣disconnected␣graph␣(component␣1)
           ")
        self.assertEqual(len(res_source_0), 3, "DFS␣from␣source␣0
           ␣should␣only␣visit␣its␣component")
        # Specific order for this implementation:
        self.assertEqual(res_source_0, [0, 1, 2])


    def test_dfs_disconnected_graph_source_in_component2(self):
        V = 5
        adj = [[0] * V for _ in range(V)]
        add_edge(adj, 0, 1)
        add_edge(adj, 1, 2)
```

```
        add_edge(adj, 3, 4)

        res_source_3 = DFS(adj, 3)
        # Expected: 3 -> 4
        self.assertEqual(sorted(res_source_3), sorted([3, 4]), "
           DFS␣from␣source␣3␣on␣disconnected␣graph␣(component␣2)")
        self.assertEqual(len(res_source_3), 2, "DFS␣from␣source␣3
           ␣should␣only␣visit␣its␣component")
        # Specific order for this implementation:
        self.assertEqual(res_source_3, [3, 4])

    def test_dfs_linear_graph(self):
        V = 4
        adj = [[0] * V for _ in range(V)]
        edges = [(0,1), (1,2), (2,3)]
        for s, t in edges: add_edge(adj, s, t)

        res_source_0 = DFS(adj, 0)
        self.assertEqual(res_source_0, [0, 1, 2, 3], "DFS␣on␣
           linear␣graph␣from␣one␣end")

        res_source_3 = DFS(adj, 3)
        self.assertEqual(res_source_3, [3, 2, 1, 0], "DFS␣on␣
           linear␣graph␣from␣other␣end")

    def test_dfs_cycle_graph(self):
        V = 4
        adj = [[0] * V for _ in range(V)]
        edges = [(0,1), (1,2), (2,3), (3,0)] # Cycle: 0-1-2-3-0
        for s, t in edges: add_edge(adj, s, t)

        res_source_0 = DFS(adj, 0)
        # Expected: 0 -> 1 -> 2 -> 3
        self.assertEqual(res_source_0, [0, 1, 2, 3], "DFS␣on␣
           cycle␣graph")

    def test_dfs_source_out_of_bounds(self):
        V = 3
        adj = [[0] * V for _ in range(V)]
        add_edge(adj, 0, 1)

        with self.assertRaisesRegex(ValueError, "Source␣node␣is␣
           out␣of␣bounds."):
            DFS(adj, 3) # Source 3 is out of bounds for V=3 (
               indices 0,1,2)
        with self.assertRaisesRegex(ValueError, "Source␣node␣is␣
           out␣of␣bounds."):
            DFS(adj, -1) # Source -1 is out of bounds

    def test_dfs_graph_with_no_edges_multiple_nodes(self):
        V = 3
```

```
        adj = [[0] * V for _ in range(V)] # Nodes 0, 1, 2, no
           edges

        res_source_0 = DFS(adj, 0)
        self.assertEqual(res_source_0, [0], "DFS␣from␣source␣0␣in
           ␣no-edge␣graph")

        res_source_1 = DFS(adj, 1)
        self.assertEqual(res_source_1, [1], "DFS␣from␣source␣1␣in
           ␣no-edge␣graph")

def main():
    # This allows running tests when the script is executed
    # The argv and exit arguments are to play well with automated
        test runners / environments
    unittest.main(argv=['first-arg-is-ignored'], exit=False)

if __name__ == "__main__":
    main()
```

Listing 43: Test Code — GEMINI 2.5

GROK 3 Response

Generated Code

```
def add_edge(adj, s, t):
    """Add an edge to the adjacency matrix for an undirected
       graph."""
    adj[s][t] = 1
    adj[t][s] = 1

def dfs_util(adj, v, visited, result):
    """Utility function for DFS traversal."""
    visited[v] = True
    result.append(v)

    # Visit all adjacent vertices
    for i in range(len(adj)):
        if adj[v][i] == 1 and not visited[i]:
            dfs_util(adj, i, visited, result)

def DFS(adj):
    """
    Performs Depth-First Search on an undirected graph
       represented by an adjacency matrix.
    Returns a list of vertices in DFS traversal order.
    """
    V = len(adj)
    visited = [False] * V
    result = []

    # Handle all vertices in case graph is disconnected
    for v in range(V):
        if not visited[v]:
            dfs_util(adj, v, visited, result)

    return result

def main():
    # Example usage
    V = 5
    adj = [[0] * V for _ in range(V)]
    edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]

    # Populate the adjacency matrix
    for s, t in edges:
        add_edge(adj, s, t)

    # Perform DFS
    res = DFS(adj)
    print("␣".join(map(str, res)))

if __name__ == "__main__":
    main()
```

Listing 44: Generated Code — GROK 3

Test Code

```
import unittest

def add_edge(adj, s, t):
    """Add an edge to the adjacency matrix for an undirected
       graph."""
    adj[s][t] = 1
    adj[t][s] = 1

def dfs_util(adj, v, visited, result):
    """Utility function for DFS traversal."""
    visited[v] = True
    result.append(v)

    for i in range(len(adj)):
        if adj[v][i] == 1 and not visited[i]:
            dfs_util(adj, i, visited, result)

def DFS(adj):
    """
    Performs Depth-First Search on an undirected graph
       represented by an adjacency matrix.
    Returns a list of vertices in DFS traversal order.
    """
    V = len(adj)
    visited = [False] * V
    result = []

    for v in range(V):
        if not visited[v]:
            dfs_util(adj, v, visited, result)

    return result

class TestDFS(unittest.TestCase):
    def test_empty_graph(self):
        """Test DFS on an empty graph."""
        adj = [[]]
        self.assertEqual(DFS(adj), [])

    def test_single_vertex(self):
        """Test DFS on a graph with one vertex and no edges."""
        adj = [[0]]
        self.assertEqual(DFS(adj), [0])

    def test_linear_graph(self):
        """Test DFS on a linear graph (0-1-2)."""
        V = 3
```

```
        adj = [[0] * V for _ in range(V)]
        edges = [(0, 1), (1, 2)]
        for s, t in edges:
            add_edge(adj, s, t)
        result = DFS(adj)
        # Possible valid DFS orders: [0,1,2] or [2,1,0]
        self.assertTrue(result == [0, 1, 2] or result == [2, 1,
            0])

    def test_disconnected_graph(self):
        """Test DFS on a disconnected graph."""
        V = 4
        adj = [[0] * V for _ in range(V)]
        edges = [(0, 1), (2, 3)]
        for s, t in edges:
            add_edge(adj, s, t)
        result = DFS(adj)
        # Check if all vertices are visited and in valid order
        self.assertEqual(len(result), 4)
        self.assertTrue(set(result) == {0, 1, 2, 3})

    def test_complete_graph(self):
        """Test DFS on a complete graph with 3 vertices."""
        V = 3
        adj = [[0] * V for _ in range(V)]
        edges = [(0, 1), (0, 2), (1, 2)]
        for s, t in edges:
            add_edge(adj, s, t)
        result = DFS(adj)
        # Check if all vertices are visited
        self.assertEqual(len(result), 3)
        self.assertTrue(set(result) == {0, 1, 2})

    def test_cycle_graph(self):
        """Test DFS on a cycle graph (0-1-2-0)."""
        V = 3
        adj = [[0] * V for _ in range(V)]
        edges = [(0, 1), (1, 2), (2, 0)]
        for s, t in edges:
            add_edge(adj, s, t)
        result = DFS(adj)
        self.assertEqual(len(result), 3)
        self.assertTrue(set(result) == {0, 1, 2})

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 45: Test Code — GROK 3

Prompt #6: Divide and Conquer

Prompt:
You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked with developing a merge sort algorithm using a divide-and-conquer strategy. This algorithm must receive as input one array. Let's use this to execute the program: arr = [12, 11, 13, 5, 6, 7] print(”Given array is”) print(*arr) arr = mergeSort(arr) print(”\nSorted array is ”) print(*arr). Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone; they are run as follows: `exec(main_code.py, ns)`; `exec(test_code.py, ns)`. Each—the tests and script—classes must have its own `if __name__ == "__main__": main()`.

Reference Code

```python
# Function to merge two sorted arrays
def merge(arr1, arr2):
    i = 0
    j = 0
    # Array to store the merged sorted array
    result = []
    while(i < len(arr1) and j < len(arr2)):
        # If arr1[i] is smaller than arr2[j], push arr1[i] into the result and move to the next element in arr1
        if arr2[j] > arr1[i]:
            result.append(arr1[i])
            i += 1
        # If arr2[j] is smaller than arr1[i], push arr2[j] into the result and move to the next element in arr2
        else:
            result.append(arr2[j])
            j += 1

    # Push the remaining elements of arr1, if any
    while(i < len(arr1)):
        result.append(arr1[i])
        i += 1
        # Push the remaining elements of arr2, if any
    while(j < len(arr2)):
        result.append(arr2[j])
        j += 1

    return result
```

```
# Function to sort arr using merge sort
def mergeSort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    # Sort the left half
    left = mergeSort(arr[:mid])
    # Sort the right half
    right = mergeSort(arr[mid:])

    return merge(left, right)


# Driver Code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given␣array␣is")
    print(*arr)
    arr = mergeSort(arr)
    print("\nSorted␣array␣is␣")
    print(*arr)
```

Listing 46: Reference Code — GPT4o

GPT4o Response

Generated Code

```
# main_code.py

def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        mergeSort(L)
        mergeSort(R)

        i = j = k = 0

        # Merge the two halves
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
```

```
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

    return arr

def main():
    arr = [12, 11, 13, 5, 6, 7]
    print("Given␣array␣is")
    print(*arr)
    arr = mergeSort(arr)
    print("\nSorted␣array␣is")
    print(*arr)

if __name__ == "__main__":
    main()
```

Listing 47: Generated Code — GPT4o

Test Code

```
# test_code.py

import unittest

class TestMergeSort(unittest.TestCase):

    def test_empty_array(self):
        self.assertEqual(mergeSort([]), [])

    def test_single_element(self):
        self.assertEqual(mergeSort([1]), [1])

    def test_sorted_array(self):
        self.assertEqual(mergeSort([1, 2, 3]), [1, 2, 3])

    def test_unsorted_array(self):
        self.assertEqual(mergeSort([3, 2, 1]), [1, 2, 3])

    def test_duplicates(self):
        self.assertEqual(mergeSort([4, 5, 1, 2, 2, 4]), [1, 2, 2,
            4, 4, 5])

    def test_negative_numbers(self):
        self.assertEqual(mergeSort([-1, -3, -2, 0, 2]), [-3, -2,
           -1, 0, 2])
```

```python

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 48: Test Code — GPT4o

GPT o4-mini-high Response

Generated Code

```
# main_code.py

def mergeSort(arr):
    """
    Perform merge sort on the input list and return a new sorted
       list.
    """
    # Lists of length 0 or 1 are already sorted
    if len(arr) <= 1:
        return arr.copy()

    mid = len(arr) // 2
    left = mergeSort(arr[:mid])
    right = mergeSort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    """
    Merge two sorted lists into one sorted list, preserving
       stability.
    """
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        # Use <= to ensure stability (left elements come first
           when equal)
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # Append any leftovers
    result.extend(left[i:])
    result.extend(right[j:])
    return result

def main():
    arr = [12, 11, 13, 5, 6, 7]
    print("Given␣array␣is")
    print(*arr)
    arr = mergeSort(arr)
    print("\nSorted␣array␣is␣")
    print(*arr)

if __name__ == "__main__":
    main()
```

Listing 49: Generated Code — GPTo4-mini-high

Test Code

```python
# test_code.py

import unittest

class TestMergeSort(unittest.TestCase):
    def test_empty_list(self):
        self.assertEqual(mergeSort([]), [])

    def test_single_element(self):
        self.assertEqual(mergeSort([42]), [42])

    def test_sorted_list(self):
        self.assertEqual(mergeSort([1, 2, 3, 4]), [1, 2, 3, 4])

    def test_reverse_sorted(self):
        self.assertEqual(mergeSort([4, 3, 2, 1]), [1, 2, 3, 4])

    def test_duplicates(self):
        self.assertEqual(mergeSort([5, 1, 5, 3, 1]), [1, 1, 3, 5,
            5])

    def test_negative_and_zero(self):
        self.assertEqual(mergeSort([-2, 0, 5, -1, 3]), [-2, -1,
           0, 3, 5])

    def test_floats(self):
        self.assertEqual(mergeSort([3.2, 1.5, 2.7]), [1.5, 2.7,
           3.2])

    def test_strings(self):
        self.assertEqual(
            mergeSort(["banana", "apple", "cherry"]),
            ["apple", "banana", "cherry"]
        )

    def test_random_list(self):
        import random
        data = random.sample(range(-100, 100), 20)
        expected = sorted(data)
        self.assertEqual(mergeSort(data), expected)

    def test_stability(self):
        # Objects with equal keys should remain in original order
        class Obj:
            def __init__(self, key, name):
                self.key = key
                self.name = name
            def __lt__(self, other):
                return self.key < other.key
            def __eq__(self, other):
```

```
                return self.key == other.key and self.name ==
                   other.name
            def __repr__(self):
                return f"Obj({self.key},{self.name})"

        objs = [Obj(1, ’a’), Obj(3, ’b’), Obj(1, ’c’), Obj(2, ’d’
           )]
        sorted_objs = mergeSort(objs)
        # Check keys are sorted
        self.assertEqual([o.key for o in sorted_objs], [1, 1, 2,
           3])
        # Check stability for key == 1: ’a’ comes before ’c’
        self.assertEqual(
            [o.name for o in sorted_objs if o.key == 1],
            [’a’, ’c’]
        )

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 50: Test Code — GPTo4-mini-high

GEMINI 2.5 Response

Generated Code

```
# Main Code: Merge Sort Algorithm
def mergeSort(arr):
    """
    Sorts an array using the merge sort algorithm (divide and
       conquer).

    Args:
        arr: The list of elements to be sorted.

    Returns:
        A new list containing the sorted elements.
        Returns the original list if it has 0 or 1 element.
    """
    if len(arr) > 1:
        mid = len(arr) // 2  # Finding the mid of the array
        L = arr[:mid]       # Dividing the array elements into 2
           halves
        R = arr[mid:]

        mergeSort(L)        # Sorting the first half
        mergeSort(R)        # Sorting the second half

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
    return arr

def main():
    """
    Main function to demonstrate merge sort.
```

```
    """
    arr = [12, 11, 13, 5, 6, 7]
    print("Given␣array␣is")
    print(*arr)
    sorted_arr = mergeSort(list(arr)) # Pass a copy to avoid
       modifying original during test
    print("\nSorted␣array␣is␣")
    print(*sorted_arr)

    arr_empty = []
    print("\nGiven␣array␣is")
    print(*arr_empty)
    sorted_arr_empty = mergeSort(list(arr_empty))
    print("\nSorted␣array␣is␣")
    print(*sorted_arr_empty)

    arr_single = [42]
    print("\nGiven␣array␣is")
    print(*arr_single)
    sorted_arr_single = mergeSort(list(arr_single))
    print("\nSorted␣array␣is␣")
    print(*sorted_arr_single)

if __name__ == "__main__":
    main()
```

Listing 51: Generated Code — GEMINI 2.5

Test Code

```
# --- Unit Tests ---
import unittest

class TestMergeSort(unittest.TestCase):
    """
    Unit tests for the mergeSort function.
    """

    def test_empty_array(self):
        """Test sorting an empty array."""
        arr = []
        self.assertEqual(mergeSort(list(arr)), [])

    def test_single_element_array(self):
        """Test sorting an array with a single element."""
        arr = [1]
        self.assertEqual(mergeSort(list(arr)), [1])
        arr = [42]
        self.assertEqual(mergeSort(list(arr)), [42])

    def test_already_sorted_array(self):
        """Test sorting an already sorted array."""
```

```
        arr = [1, 2, 3, 4, 5]
        self.assertEqual(mergeSort(list(arr)), [1, 2, 3, 4, 5])

    def test_reverse_sorted_array(self):
        """Test sorting a reverse sorted array."""
        arr = [5, 4, 3, 2, 1]
        self.assertEqual(mergeSort(list(arr)), [1, 2, 3, 4, 5])

    def test_array_with_duplicates(self):
        """Test sorting an array with duplicate elements."""
        arr = [12, 11, 13, 5, 6, 5, 7, 11]
        self.assertEqual(mergeSort(list(arr)), [5, 5, 6, 7, 11,
           11, 12, 13])

    def test_provided_example_array(self):
        """Test sorting the example array given in the problem
           description."""
        arr = [12, 11, 13, 5, 6, 7]
        self.assertEqual(mergeSort(list(arr)), [5, 6, 7, 11, 12,
           13])

    def test_array_with_negative_numbers(self):
        """Test sorting an array containing negative numbers."""
        arr = [-5, 10, -2, 0, 3, -8]
        self.assertEqual(mergeSort(list(arr)), [-8, -5, -2, 0, 3,
            10])

    def test_array_with_mixed_positive_and_negative_duplicates(
       self):
        """Test sorting an array with mixed positive/negative
           numbers and duplicates."""
        arr = [3, -1, 4, 1, -5, 9, -1, 2, -5, 6]
        self.assertEqual(mergeSort(list(arr)), [-5, -5, -1, -1,
           1, 2, 3, 4, 6, 9])

def main_tests():
    """
    Main function to run unit tests.
    """
    unittest.main(argv=[’first-arg-is-ignored’], exit=False)

if __name__ == "__main__": # This will be for running tests if
   this script is run directly
    # Note: The problem states the evaluator runs main_code.py
       and then test_code.py separately.
    # So, the __main__ block in the test section is primarily for
        standalone testing of the tests.
    # To ensure the evaluator’s ns (namespace) mechanism works as
        expected,
    # we’ll conditionally call main_tests().
```

```
    # If this script were named 'test_code.py' and run after 'main_code.py',
    # the 'mergeSort' function would already be in the namespace.

    # For the evaluator:
    # Assuming the evaluator executes the main code first, 'mergeSort' will be in the namespace.
    # Then, when this test code is executed, TestMergeSort can find 'mergeSort'.
    # The `main()` from the main code block should not be re-run by the test script.
    # The `main_tests()` will run the unit tests.

    # If this entire file is run as a single script (e.g. `python this_file.py`):
    # 1. The first `if __name__ == "__main__": main()` will execute, running the demo.
    # 2. The second `if __name__ == "__main__": main_tests()` will execute, running the tests.
    # This setup ensures both `main()` (demo) and `main_tests()` (unit tests) run when the script is executed directly,
    # while also being compatible with the described external evaluator.
    #
    # To simulate the evaluator, one might comment out the first `if __name__ == "__main__": main()`
    # when "running" the test part, but the prompt implies `exec(main_code.py, ns); exec(test_code.py, ns)`
    # so both `main` functions will be defined, but only the one in the *current* `__name__ == "__main__"` context
    # would typically run if not for the explicit separate exec calls.
    # The evaluator will effectively do:
    # ns = {}
    # exec(content_of_main_code_section, ns) # This runs the main_code's main() if __name__ is "__main__"
    # exec(content_of_test_code_section, ns) # This runs the test_code's main_tests() if __name__ is "__main__"

    # For clarity in the provided single script:
    # The problem asks for a single output. When run, the main code's main() will execute
    # and then the test code's main_tests() will execute.
    # The evaluator will handle the separation.
    print("\n---␣Running␣Unit␣Tests␣---")
    main_tests()
```

Listing 52: Test Code — GEMINI 2.5

GROK 3 Response

Generated Code

```
# Main Merge Sort Implementation
def merge(arr, left, mid, right):
    left_half = arr[left:mid + 1]
    right_half = arr[mid + 1:right + 1]

    i = j = 0
    k = left

    while i < len(left_half) and j < len(right_half):
        if left_half[i] <= right_half[j]:
            arr[k] = left_half[i]
            i += 1
        else:
            arr[k] = right_half[j]
            j += 1
        k += 1

    while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1

    while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1

def mergeSort(arr, left=None, right=None):
    if left is None and right is None:
        left = 0
        right = len(arr) - 1

    if left < right:
        mid = (left + right) // 2
        mergeSort(arr, left, mid)
        mergeSort(arr, mid + 1, right)
        merge(arr, left, mid, right)
    return arr

# Main function for the merge sort implementation
def main():
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is")
    print(*arr)
    arr = mergeSort(arr)
    print("\nSorted array is")
    print(*arr)

if __name__ == "__main__":
```

```
    main()
```

Listing 53: Generated Code — GROK 3

Test Code

```
# Unit Tests
import unittest

class TestMergeSort(unittest.TestCase):
    def test_empty_array(self):
        """Test merge sort with an empty array"""
        self.assertEqual(mergeSort([]), [])

    def test_single_element(self):
        """Test merge sort with a single element"""
        self.assertEqual(mergeSort([1]), [1])

    def test_sorted_array(self):
        """Test merge sort with an already sorted array"""
        self.assertEqual(mergeSort([1, 2, 3, 4, 5]), [1, 2, 3, 4,
            5])

    def test_reverse_sorted_array(self):
        """Test merge sort with a reverse sorted array"""
        self.assertEqual(mergeSort([5, 4, 3, 2, 1]), [1, 2, 3, 4,
            5])

    def test_duplicate_elements(self):
        """Test merge sort with duplicate elements"""
        self.assertEqual(mergeSort([3, 1, 4, 1, 5, 9, 2, 6, 5]),
           [1, 1, 2, 3, 4, 5, 5, 6, 9])

    def test_negative_numbers(self):
        """Test merge sort with negative numbers"""
        self.assertEqual(mergeSort([-3, 1, -5, 4, -2]), [-5, -3,
           -2, 1, 4])

    def test_example_case(self):
        """Test merge sort with the example case provided"""
        self.assertEqual(mergeSort([12, 11, 13, 5, 6, 7]), [5, 6,
            7, 11, 12, 13])

    def test_large_numbers(self):
        """Test merge sort with large numbers"""
        self.assertEqual(mergeSort([1000000, 2, 500000, 1]), [1,
           2, 500000, 1000000])

# Main function for running tests
def main_test():
    unittest.main(argv=[''], exit=False)
```

```
if __name__ == "__main__":
    main_test()
```

Listing 54: Test Code — GROK 3

Prompt #7: Flappy Bird Game

Prompt:
You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked with developing a simple Flappy Bird game using Python. The Flappy Bird game should include a UI; you don't need to generate an assets file, just use characters instead of images.Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone; they are run as follows: `exec(main_code.py, ns)`; `exec(test_code.py, ns)`. Each, the tests and script classes must have its own guard: `if __name__ == "__main__": main()`.

Reference Code

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Mar 23 14:17:24 2019

@author: Mehul
"""



import math
import os
from collections import deque
from random import randint

import pygame
from pygame.locals import *

FPS = 60
ANI_SPEED = 0.18  # pixels per millisecond
W_WIDTH = 284 * 2  # BG image size: 284x512 px; tiled twice
W_HEIGHT = 512


class Bird(pygame.sprite.Sprite):

    WIDTH = 32  #   bird image width
    HEIGHT = 32  #   bird image height
    DOWN_SPEED = 0.18  #   pix per ms  -y
    UP_SPEED = 0.3  #   pix per ms  +y
    UP_DURATION = 150  #   time for which bird go up

    def __init__(self, x, y, ms_to_up, images):
```

```
        super(Bird, self).__init__()
        self.x, self.y = x, y
        self.ms_to_up = ms_to_up
        self._img_wingup, self._img_wingdown = images
        self._mask_wingup = pygame.mask.from_surface(self.
           _img_wingup)
        self._mask_wingdown = pygame.mask.from_surface(self.
           _img_wingdown)

    def update(self, delta_frames=1):

        if self.ms_to_up > 0:
            frac_climb_done = 1 - self.ms_to_up / Bird.
               UP_DURATION
            self.y -= (
                Bird.UP_SPEED
                * frames_to_msec(delta_frames)
                * (1 - math.cos(frac_climb_done * math.pi))
            )
            self.ms_to_up -= frames_to_msec(delta_frames)
        else:
            self.y += Bird.DOWN_SPEED * frames_to_msec(
               delta_frames)

    @property
    def image(self):
        # to animate bird
        if pygame.time.get_ticks() % 500 >= 250:
            return self._img_wingup
        else:
            return self._img_wingdown

    @property
    def mask(self):
        # collision detection
        if pygame.time.get_ticks() % 500 >= 250:
            return self._mask_wingup
        else:
            return self._mask_wingdown

    @property
    def rect(self):
        # return birds params
        return Rect(self.x, self.y, Bird.WIDTH, Bird.HEIGHT)


class PipePair(pygame.sprite.Sprite):

    WIDTH = 80  #    width of pipe
    PIECE_HEIGHT = 32
```

```
    ADD_INTERVAL = 3000

    def __init__(self, pipe_end_img, pipe_body_img):

        self.x = float(W_WIDTH - 1)
        self.score_counted = False

        self.image = pygame.Surface((PipePair.WIDTH, W_HEIGHT),
           SRCALPHA)
        self.image.convert()  # speeds up blitting
        self.image.fill((0, 0, 0, 0))
        total_pipe_body_pieces = int(
            (
                W_HEIGHT
                - 3 * Bird.HEIGHT  # fill window from top to
                   bottom
                - 3 * PipePair.PIECE_HEIGHT  # make room for bird
                    to fit through
            )
            / PipePair.PIECE_HEIGHT  # 2 end pieces + 1 body
               piece  # to get number of pipe pieces
        )
        self.bottom_pieces = randint(1, total_pipe_body_pieces)
        self.top_pieces = total_pipe_body_pieces - self.
           bottom_pieces

        # bottom pipe
        for i in range(1, self.bottom_pieces + 1):
            piece_pos = (0, W_HEIGHT - i * PipePair.PIECE_HEIGHT)
            self.image.blit(pipe_body_img, piece_pos)
        bottom_pipe_end_y = W_HEIGHT - self.bottom_height_px
        bottom_end_piece_pos = (0, bottom_pipe_end_y - PipePair.
           PIECE_HEIGHT)
        self.image.blit(pipe_end_img, bottom_end_piece_pos)

        # top pipe
        for i in range(self.top_pieces):
            self.image.blit(pipe_body_img, (0, i * PipePair.
               PIECE_HEIGHT))
        top_pipe_end_y = self.top_height_px
        self.image.blit(pipe_end_img, (0, top_pipe_end_y))

        # compensate for added end pieces
        self.top_pieces += 1
        self.bottom_pieces += 1

        # for collision detection
        self.mask = pygame.mask.from_surface(self.image)

    @property
    def top_height_px(self):
```

```
        # returns top pipe's height in pix
        return self.top_pieces * PipePair.PIECE_HEIGHT

    @property
    def bottom_height_px(self):

        return self.bottom_pieces * PipePair.PIECE_HEIGHT

    @property
    def visible(self):
        # pipe is on screen or not
        return -PipePair.WIDTH < self.x < W_WIDTH

    @property
    def rect(self):
        # Get the Rect which contains this Pipe.
        return Rect(self.x, 0, PipePair.WIDTH, PipePair.
           PIECE_HEIGHT)

    def update(self, delta_frames=1):

        self.x -= ANI_SPEED * frames_to_msec(delta_frames)

    def collides_with(self, bird):

        return pygame.sprite.collide_mask(self, bird)


def load_images():
    def load_image(img_file_name):

        file_name = os.path.join(".", "images", img_file_name)
        img = pygame.image.load(file_name)
        img.convert()
        return img

    return {
        "background": load_image("background.png"),
        "pipe-end": load_image("pipe_end.png"),
        "pipe-body": load_image("pipe_body.png"),
        # images for animating the flapping bird -- animated GIFs
            are
        # not supported in pygame
        "bird-wingup": load_image("bird_wing_up.png"),
        "bird-wingdown": load_image("bird_wing_down.png"),
    }


def frames_to_msec(frames, fps=FPS):

    return 1000.0 * frames / fps
```

```
def msec_to_frames(milliseconds, fps=FPS):

    return fps * milliseconds / 1000.0


"""
def gameover(display, score):
    font = pygame.font.SysFont(None,55)
    text = font.render("Game Over! Score: {}".format(score),True
       ,(255,0,0))
    display.blit(text, [150,250])"""


def main():

    pygame.init()

    display_surface = pygame.display.set_mode((W_WIDTH, W_HEIGHT)
       )
    pygame.display.set_caption("Flappy␣Bird␣by␣PMN")

    clock = pygame.time.Clock()
    score_font = pygame.font.SysFont(None, 32, bold=True)  #
       default font
    images = load_images()

    # the bird stays in the same x position, so bird.x is a
       constant
    # center bird on screen
    bird = Bird(
        50,
        int(W_HEIGHT / 2 - Bird.HEIGHT / 2),
        2,
        (images["bird-wingup"], images["bird-wingdown"]),
    )

    pipes = deque()

    frame_clock = 0  # this counter is only incremented if the
       game isn't paused
    score = 0
    done = paused = False
    while not done:
        clock.tick(FPS)

        # Handle this 'manually'.  If we used pygame.time.
           set_timer(),
        # pipe addition would be messed up when paused.
```

```
        if not (paused or frame_clock % msec_to_frames(PipePair.
           ADD_INTERVAL)):
            pp = PipePair(images["pipe-end"], images["pipe-body"
               ])
            pipes.append(pp)

        for e in pygame.event.get():
            if e.type == QUIT or (e.type == KEYUP and e.key ==
               K_ESCAPE):
                done = True
                break
            elif e.type == KEYUP and e.key in (K_PAUSE, K_p):
                paused = not paused
            elif e.type == MOUSEBUTTONUP or (
                e.type == KEYUP and e.key in (K_UP, K_RETURN,
                   K_SPACE)
            ):
                bird.ms_to_up = Bird.UP_DURATION

        if paused:
            continue  # don't draw anything

        # check for collisions
        pipe_collision = any(p.collides_with(bird) for p in pipes
           )
        if pipe_collision or 0 >= bird.y or bird.y >= W_HEIGHT -
           Bird.HEIGHT:
            done = True

        for x in (0, W_WIDTH / 2):
            display_surface.blit(images["background"], (x, 0))

        while pipes and not pipes[0].visible:
            pipes.popleft()

        for p in pipes:
            p.update()
            display_surface.blit(p.image, p.rect)

        bird.update()
        display_surface.blit(bird.image, bird.rect)

        # update and display score
        for p in pipes:
            if p.x + PipePair.WIDTH < bird.x and not p.
               score_counted:
                score += 1
                p.score_counted = True

        score_surface = score_font.render(str(score), True, (255,
            255, 255))
```

```
        score_x = W_WIDTH / 2 - score_surface.get_width() / 2
        display_surface.blit(score_surface, (score_x, PipePair.
           PIECE_HEIGHT))

        pygame.display.flip()
        frame_clock += 1
    # gameover(display_surface, score)

    print("Game␣over!␣Score:␣%i" % score)

    pygame.quit()


if __name__ == "__main__":
    # If this module had been imported, __name__ would be '
       flappybird'.
    # It was executed (e.g. by double-clicking the file), so call
        main.
    main()
```

Listing 55: Reference Code — GPT4o

GPT4o Response

Generated Code

```
# main_code.py

import curses
import random
import time

class FlappyBirdGame:
    def __init__(self, stdscr, height=20, width=60):
        self.stdscr = stdscr
        self.height = height
        self.width = width
        self.bird_y = height // 2
        self.bird_x = 5
        self.velocity = 0
        self.gravity = 1
        self.lift = -3
        self.pipes = []
        self.pipe_gap = 5
        self.pipe_freq = 30
        self.frame = 0
        self.score = 0
        self.game_over = False

    def create_pipe(self):
        gap_y = random.randint(3, self.height - self.pipe_gap -
           3)
```

```
        pipe = {
            'x': self.width - 1,
            'gap_y': gap_y
        }
        self.pipes.append(pipe)

    def update_pipes(self):
        new_pipes = []
        for pipe in self.pipes:
            pipe['x'] -= 1
            if pipe['x'] + 1 >= 0:
                new_pipes.append(pipe)
            if pipe['x'] == self.bird_x:
                self.score += 1
        self.pipes = new_pipes

    def draw(self):
        self.stdscr.clear()
        for y in range(self.height):
            for x in range(self.width):
                self.stdscr.addch(y, x, '␣')
        self.stdscr.addch(self.bird_y, self.bird_x, 'O')
        for pipe in self.pipes:
            for y in range(self.height):
                if y < pipe['gap_y'] or y > pipe['gap_y'] + self.
                   pipe_gap:
                    if 0 <= pipe['x'] < self.width:
                        self.stdscr.addch(y, pipe['x'], '|')
        self.stdscr.addstr(0, 0, f"Score:␣{self.score}")
        self.stdscr.refresh()

    def apply_physics(self):
        self.velocity += self.gravity
        self.bird_y += self.velocity
        if self.bird_y < 0:
            self.bird_y = 0
        elif self.bird_y >= self.height:
            self.game_over = True

    def check_collision(self):
        for pipe in self.pipes:
            if pipe['x'] == self.bird_x:
                if not (pipe['gap_y'] <= self.bird_y <= pipe['
                   gap_y'] + self.pipe_gap):
                    self.game_over = True

    def handle_input(self):
        key = self.stdscr.getch()
        if key == ord('␣'):
            self.velocity = self.lift
```

```
    def loop(self):
        self.stdscr.nodelay(True)
        while not self.game_over:
            self.handle_input()
            self.apply_physics()
            if self.frame % self.pipe_freq == 0:
                self.create_pipe()
            self.update_pipes()
            self.check_collision()
            self.draw()
            self.frame += 1
            time.sleep(0.1)
        self.stdscr.nodelay(False)
        self.stdscr.addstr(self.height // 2, self.width // 2 - 5,
            "GAME␣OVER")
        self.stdscr.refresh()
        self.stdscr.getch()

def main():
    curses.wrapper(lambda stdscr: FlappyBirdGame(stdscr).loop())

if __name__ == "__main__":
    main()
```

Listing 56: Generated Code — GPT4o

Test Code

```
# test_code.py

import unittest
from unittest.mock import Mock

class TestFlappyBirdGame(unittest.TestCase):

    def setUp(self):
        self.mock_screen = Mock()
        self.game = FlappyBirdGame(self.mock_screen)

    def test_initial_bird_position(self):
        self.assertEqual(self.game.bird_y, self.game.height // 2)
        self.assertEqual(self.game.bird_x, 5)

    def test_create_pipe(self):
        self.assertEqual(len(self.game.pipes), 0)
        self.game.create_pipe()
        self.assertEqual(len(self.game.pipes), 1)
        self.assertTrue(3 <= self.game.pipes[0]['gap_y'] <= self.
           game.height - self.game.pipe_gap - 3)

    def test_pipe_update_and_score(self):
```

```
        self.game.pipes = [{'x': self.game.bird_x + 1, 'gap_y':
           5}]
        self.game.update_pipes()
        self.assertEqual(len(self.game.pipes), 1)
        self.game.update_pipes()
        self.assertEqual(self.game.score, 1)

    def test_collision_detection(self):
        self.game.bird_y = 0
        self.game.pipes = [{'x': self.game.bird_x, 'gap_y': 5}]
        self.game.check_collision()
        self.assertTrue(self.game.game_over)

    def test_bird_falls_due_to_gravity(self):
        self.game.velocity = 0
        initial_y = self.game.bird_y
        self.game.apply_physics()
        self.assertGreaterEqual(self.game.bird_y, initial_y)

    def test_handle_input_jump(self):
        self.game.stdscr.getch = Mock(return_value=ord('␣'))
        self.game.handle_input()
        self.assertEqual(self.game.velocity, self.game.lift)

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 57: Test Code — GPT4o

GPT o4-mini-high Response

Generated Code

```
# main_code.py

import tkinter as tk
import random

class Bird:
    GRAVITY = 0.5
    FLAP_STRENGTH = -8

    def __init__(self, canvas, x, y):
        self.canvas = canvas
        self.x = x
        self.y = y
        self.velocity = 0
        # Bird is a 20 20  square
        self.id = canvas.create_rectangle(x - 10, y - 10, x + 10,
            y + 10, fill="yellow")

    def flap(self):
        self.velocity = Bird.FLAP_STRENGTH

    def update(self):
        # apply gravity
        self.velocity += Bird.GRAVITY
        # update position
        dy = self.velocity
        self.y += dy
        self.canvas.move(self.id, 0, dy)

    def get_position(self):
        # returns [x1, y1, x2, y2]
        return self.canvas.coords(self.id)


class Pipe:
    WIDTH = 50

    def __init__(self, canvas, x, gap_y, gap_size, canvas_height)
       :
        self.canvas = canvas
        self.x = x
        self.gap_y = gap_y
        self.gap_size = gap_size
        self.canvas_height = canvas_height
        half_gap = gap_size // 2
        # top pipe from y=0 down to gap_y - half_gap
        self.top = canvas.create_rectangle(x, 0, x + Pipe.WIDTH,
            gap_y - half_gap, fill="green")
        # bottom pipe from gap_y + half_gap down to bottom
```

```
        self.bottom = canvas.create_rectangle(x, gap_y + half_gap
           , x + Pipe.WIDTH, canvas_height, fill="green")

    def update(self):
        # move pipe left
        self.x -= 3
        self.canvas.move(self.top, -3, 0)
        self.canvas.move(self.bottom, -3, 0)

    def get_positions(self):
        return self.canvas.coords(self.top), self.canvas.coords(
           self.bottom)


class Game:
    WIDTH = 400
    HEIGHT = 600

    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Flappy␣Bird␣(Text␣Edition)")
        self.canvas = tk.Canvas(self.root, width=Game.WIDTH,
           height=Game.HEIGHT, bg="skyblue")
        self.canvas.pack()
        self.bird = Bird(self.canvas, 100, Game.HEIGHT // 2)
        self.pipes = []
        self.score = 0
        self.running = True

        # spacebar to flap
        self.root.bind("<space>", lambda e: self.bird.flap())

    def start(self):
        # begin spawning pipes and updating
        self.spawn_pipe()
        self.update()
        self.root.mainloop()

    def spawn_pipe(self):
        gap_size = 150
        gap_y = random.randint(100, Game.HEIGHT - 100)
        pipe = Pipe(self.canvas, Game.WIDTH, gap_y, gap_size,
           Game.HEIGHT)
        self.pipes.append(pipe)
        # schedule next pipe
        self.root.after(2000, self.spawn_pipe)

    def update(self):
        if not self.running:
            return
```

```
        # update bird
        self.bird.update()

        # update pipes
        for pipe in list(self.pipes):
            pipe.update()
            # remove off-screen pipes
            if pipe.x + Pipe.WIDTH < 0:
                self.canvas.delete(pipe.top)
                self.canvas.delete(pipe.bottom)
                self.pipes.remove(pipe)
                self.score += 1

        # check collisions
        if self.check_collision():
            self.end_game()
            return

        # redraw score
        self.canvas.delete("score")
        self.canvas.create_text(
            Game.WIDTH // 2, 50,
            text=f"Score:␣{self.score}",
            font=("Arial", 24),
            tag="score"
        )

        # loop
        self.root.after(30, self.update)

    def check_collision(self):
        x1, y1, x2, y2 = self.bird.get_position()
        # hit floor or ceiling
        if y1 <= 0 or y2 >= Game.HEIGHT:
            return True

        for pipe in self.pipes:
            top_coords, bottom_coords = pipe.get_positions()
            if self.overlap([x1, y1, x2, y2], top_coords) or \
               self.overlap([x1, y1, x2, y2], bottom_coords):
                return True
        return False

    @staticmethod
    def overlap(a, b):
        ax1, ay1, ax2, ay2 = a
        bx1, by1, bx2, by2 = b
        return not (ax2 < bx1 or ax1 > bx2 or ay2 < by1 or ay1 >
           by2)

    def end_game(self):
```

```
        self.running = False
        self.canvas.create_text(
            Game.WIDTH // 2, Game.HEIGHT // 2 - 20,
            text="Game␣Over", font=("Arial", 32), fill="red"
        )
        self.canvas.create_text(
            Game.WIDTH // 2, Game.HEIGHT // 2 + 20,
            text=f"Final␣Score:␣{self.score}", font=("Arial", 24)
               , tag="score"
        )


def main():
    game = Game()
    game.start()


if __name__ == "__main__":
    main()
```

Listing 58: Generated Code — GPTo4-mini-high

Test Code

```
# test_code.py

import unittest

# A minimal stand-in for tkinter.Canvas to test logic without a
   real UI
class DummyCanvas:
    def __init__(self):
        self._items = {}
        self._next_id = 1

    def create_rectangle(self, x1, y1, x2, y2, fill=None):
        _id = self._next_id
        self._next_id += 1
        self._items[_id] = [x1, y1, x2, y2]
        return _id

    def move(self, _id, dx, dy):
        coords = self._items[_id]
        self._items[_id] = [coords[0] + dx, coords[1] + dy,
           coords[2] + dx, coords[3] + dy]

    def coords(self, _id):
        return self._items[_id]

    def create_text(self, *args, **kwargs):
        # no-op for tests
        return None
```

```
    def delete(self, _id):
        if _id in self._items:
            del self._items[_id]


# Bring in classes from the main script namespace
# (When using exec(main_code.py, ns); exec(test_code.py, ns),
#  Bird, Pipe, and Game are already defined in ns.)

class TestBird(unittest.TestCase):
    def setUp(self):
        self.canvas = DummyCanvas()
        from __main__ import Bird
        self.bird = Bird(self.canvas, 50, 75)

    def test_flap_sets_velocity(self):
        self.bird.velocity = 5
        self.bird.flap()
        self.assertEqual(self.bird.velocity, Bird.FLAP_STRENGTH)

    def test_update_applies_gravity(self):
        initial_velocity = self.bird.velocity
        self.bird.update()
        self.assertAlmostEqual(self.bird.velocity,
           initial_velocity + Bird.GRAVITY)

    def test_update_moves_canvas_item(self):
        self.bird.velocity = 0
        self.bird.update()
        coords = self.canvas.coords(self.bird.id)
        # bird started at (50,75) with size 20, so top-left was
           (40,65)
        expected_dy = Bird.GRAVITY
        self.assertEqual(coords, [40, 65 + expected_dy, 60, 85 +
           expected_dy])


class TestPipe(unittest.TestCase):
    def setUp(self):
        self.canvas = DummyCanvas()
        from __main__ import Pipe, Game
        # use known values
        self.pipe = Pipe(self.canvas, x=200, gap_y=300, gap_size
           =150, canvas_height=Game.HEIGHT)

    def test_initial_pipe_positions(self):
        top_coords, bottom_coords = self.pipe.get_positions()
        half_gap = 150 // 2
        self.assertEqual(top_coords, [200, 0, 200 + Pipe.WIDTH,
           300 - half_gap])
```

```
        self.assertEqual(bottom_coords, [200, 300 + half_gap, 200
            + Pipe.WIDTH, Game.HEIGHT])

    def test_pipe_moves_left(self):
        old_x = self.pipe.x
        self.pipe.update()
        self.assertEqual(self.pipe.x, old_x - 3)
        top_coords, _ = self.pipe.get_positions()
        self.assertEqual(top_coords[0], 200 - 3)


class TestGameOverlap(unittest.TestCase):
    def setUp(self):
        from __main__ import Game
        self.game = Game()

    def test_overlap_true(self):
        a = [0, 0, 10, 10]
        b = [5, 5, 15, 15]
        self.assertTrue(self.game.overlap(a, b))

    def test_overlap_false(self):
        a = [0, 0, 10, 10]
        b = [11, 11, 20, 20]
        self.assertFalse(self.game.overlap(a, b))


def main():
    unittest.main(verbosity=2)


if __name__ == "__main__":
    main()
```

Listing 59: Test Code — GPTo4-mini-high

GEMINI 2.5 Response

Generated Code

```
# -*- coding: utf-8 -*-
import curses
import random
import time
import unittest
from unittest.mock import MagicMock

# --- Main Game Code ---

# Constants
BIRD_CHAR = 'O'
PIPE_CHAR = '#'
GAP_SIZE = 5  # Vertical space between pipes
PIPE_WIDTH = 5
MIN_PIPE_HEIGHT = 2
GRAVITY = 1
FLAP_STRENGTH = -3 # Negative because y=0 is top
PIPE_SPEED = 1 # How many columns pipes move per frame
PIPE_SPAWN_RATE = 40 # How often (in game loops) a new pipe spawns

class FlappyBirdGame:
    """
    Encapsulates the logic and rendering for a text-based Flappy Bird game
    using the curses library.
    """
    def __init__(self, stdscr):
        """
        Initializes the game state.

        Args:
            stdscr: The curses screen object.
        """
        self.stdscr = stdscr
        self.height, self.width = stdscr.getmaxyx()

        # Game state
        self.bird_y = self.height // 2
        self.bird_x = self.width // 4
        self.bird_vel = 0
        self.pipes = []  # List of tuples: (x, gap_y, scored)
        self.score = 0
        self.frames = 0
        self.game_over = False
        self.paused = False

        # Configure curses
        curses.curs_set(0)  # Hide cursor
```

```
        self.stdscr.nodelay(1)  # Non-blocking input
        self.stdscr.timeout(100) # Refresh rate (milliseconds) -
            controls game speed

    def _flap(self):
        """Applies upward velocity to the bird."""
        if not self.game_over and not self.paused:
            self.bird_vel = FLAP_STRENGTH

    def _update_bird(self):
        """Updates bird position based on velocity and gravity.
            """
        if self.paused:
            return

        # Apply gravity
        self.bird_vel += GRAVITY

        # Update position
        self.bird_y += self.bird_vel

        # Collision with ceiling or floor
        if self.bird_y <= 0 or self.bird_y >= self.height - 1:
            self.game_over = True

    def _spawn_pipe(self):
        """Generates a new pipe pair."""
        gap_y = random.randint(MIN_PIPE_HEIGHT, self.height -
            MIN_PIPE_HEIGHT - GAP_SIZE)
        self.pipes.append([self.width, gap_y, False]) # x, gap_y,
            scored

    def _update_pipes(self):
        """Moves pipes, removes old ones, spawns new ones, and
            checks scoring."""
        if self.paused:
            return

        # Spawn new pipes
        if self.frames % PIPE_SPAWN_RATE == 0:
            self._spawn_pipe()

        new_pipes = []
        scored_this_frame = False
        for pipe in self.pipes:
            pipe[0] -= PIPE_SPEED  # Move pipe left

            # Check if bird passed this pipe
            if not pipe[2] and pipe[0] + PIPE_WIDTH < self.bird_x
                :
                pipe[2] = True  # Mark as scored
```

```
                self.score += 1
                scored_this_frame = True

            # Keep pipe if it's still on screen
            if pipe[0] + PIPE_WIDTH > 0:
                new_pipes.append(pipe)

        self.pipes = new_pipes
        # Optional: Make a sound or visual cue on score
        # if scored_this_frame:
        #     curses.beep()

    def _check_collision(self):
        """Checks if the bird has collided with any pipes."""
        if self.paused:
            return False

        for pipe_x, gap_y, _ in self.pipes:
            # Check if bird's x aligns with the pipe's x range
            if self.bird_x >= pipe_x and self.bird_x < pipe_x +
               PIPE_WIDTH:
                # Check if bird is within the gap
                if not (self.bird_y > gap_y and self.bird_y <
                   gap_y + GAP_SIZE):
                    self.game_over = True
                    return True # Collision occurred
        return False # No collision

    def _draw(self):
        """Draws the current game state to the screen."""
        self.stdscr.clear()

        # Draw Bird
        try:
            if 0 <= self.bird_y < self.height and 0 <= self.
               bird_x < self.width:
                 self.stdscr.addstr(int(self.bird_y), int(self.
                    bird_x), BIRD_CHAR)
        except curses.error:
            # Handle cases where bird position might be slightly
               out of bounds
            # before game over is triggered, prevents crash.
            self.game_over = True # Force game over if draw fails


        # Draw Pipes
        for pipe_x, gap_y, _ in self.pipes:
            for y in range(self.height):
                # Draw top part of pipe
                if y <= gap_y:
                    for x_offset in range(PIPE_WIDTH):
```

```
                         current_x = pipe_x + x_offset
                         if 0 <= y < self.height and 0 <=
                            current_x < self.width:
                             try:
                                 self.stdscr.addstr(y, current_x,
                                     PIPE_CHAR)
                             except curses.error:
                                 pass # Ignore drawing errors at
                                    screen edges
                # Draw bottom part of pipe
                elif y >= gap_y + GAP_SIZE:
                    for x_offset in range(PIPE_WIDTH):
                        current_x = pipe_x + x_offset
                        if 0 <= y < self.height and 0 <=
                           current_x < self.width:
                           try:
                               self.stdscr.addstr(y, current_x,
                                  PIPE_CHAR)
                           except curses.error:
                               pass # Ignore drawing errors at
                                  screen edges

        # Draw Score
        score_text = f"Score:␣{self.score}"
        self.stdscr.addstr(0, 1, score_text)

        # Draw Game Over / Paused message
        if self.game_over:
            msg = "GAME␣OVER!␣Score:␣{}␣(Press␣’r’␣to␣restart,␣’q
               ’␣to␣quit)".format(self.score)
            msg_x = self.width // 2 - len(msg) // 2
            msg_y = self.height // 2
            self.stdscr.addstr(msg_y, msg_x, msg)
        elif self.paused:
            msg = "PAUSED␣(Press␣’p’␣to␣resume,␣’q’␣to␣quit)"
            msg_x = self.width // 2 - len(msg) // 2
            msg_y = self.height // 2
            self.stdscr.addstr(msg_y, msg_x, msg)
        else:
             # Draw instructions while playing
             instructions = "’Space’␣to␣flap,␣’p’␣to␣pause,␣’q’␣
                to␣quit"
             self.stdscr.addstr(self.height - 1, 1, instructions)


        self.stdscr.refresh()

    def _handle_input(self):
        """Handles user input."""
        key = self.stdscr.getch() # Get character, non-blocking
```

```
        if key == ord('q') or key == ord('Q'):
            return False  # Signal to quit the game loop
        elif key == ord('␣'):
            if not self.game_over:
                 self._flap()
        elif key == ord('p') or key == ord('P'):
             if not self.game_over:
                 self.paused = not self.paused
        elif key == ord('r') or key == ord('R'):
            if self.game_over:
                self._reset_game() # Restart the game

        return True # Signal to continue

    def _reset_game(self):
        """Resets the game state for a new game."""
        self.bird_y = self.height // 2
        self.bird_vel = 0
        self.pipes = []
        self.score = 0
        self.frames = 0
        self.game_over = False
        self.paused = False
        # Ensure pipes start spawning again soon
        self._spawn_pipe()


    def run(self):
        """Main game loop."""
        self._spawn_pipe() # Start with one pipe
        while True:
            if not self._handle_input(): # Check for quit command
                break

            if not self.game_over and not self.paused:
                self._update_bird()
                self._update_pipes()
                self._check_collision() # Sets game_over if
                   collision occurs
                self.frames += 1

            self.draw()

            # Removed explicit time.sleep(), relying on stdscr.
               timeout()
            # time.sleep(0.1) # Control game speed - replaced by
               timeout

def main_game(stdscr):
    """
    Wrapper function called by curses.wrapper to run the game.
```

```
    Handles curses initialization and cleanup.
    """
    game = FlappyBirdGame(stdscr)
    game.run()

# Main execution block for the game
def main():
    """Entry point for the game script."""
    try:
        curses.wrapper(main_game)
    except curses.error as e:
        print(f"Curses␣error:␣{e}")
        print("Your␣terminal␣might␣not␣support␣curses,␣or␣the␣
           window␣is␣too␣small.")
    except KeyboardInterrupt:
        print("Game␣interrupted␣by␣user.")
    finally:
        # Just in case wrapper didn't clean up fully (e.g., on
           error)
        try:
            curses.endwin()
        except:
            pass
        print("Game␣exited.")

if __name__ == "__main__":
    main()
```

Listing 60: Generated Code — GEMINI 2.5

Test Code

```
import curses
import random
import time
import unittest
from unittest.mock import MagicMock
class MockStdScr:
    """
    A mock object for the curses screen (stdscr) to facilitate
       testing
    without a real terminal.
    """
    def __init__(self, height=24, width=80):
        self.height = height
        self.width = width
        self.cursor_visible = 1
        self._nodelay = False
        self._timeout = -1
        self.key_queue = []

    def getmaxyx(self):
```

```
        return self.height, self.width

    def nodelay(self, flag):
        self._nodelay = flag

    def timeout(self, delay):
         self._timeout = delay # Store timeout, but don’t simulate timing here

    def curs_set(self, visibility):
        self.cursor_visible = visibility

    def clear(self):
        pass # No-op for testing logic

    def addstr(self, y, x, text, attr=None):
         # Basic check for bounds, could be more robust if needed
        if not (0 <= y < self.height and 0 <= x < self.width):
            raise curses.error("Mock␣curses␣error:␣Write␣out␣of␣bounds")
         # No actual drawing needed for most logic tests

    def refresh(self):
        pass # No-op

    def getch(self):
        if self.key_queue:
            return self.key_queue.pop(0)
        return -1 # No key pressed

    # Helper for tests to simulate key presses
    def add_key(self, key_ord):
        self.key_queue.append(key_ord)


class TestFlappyBirdGame(unittest.TestCase):
    """Unit tests for the FlappyBirdGame class."""

    def setUp(self):
        """Set up a game instance with a mock screen before each test."""
        self.mock_scr = MockStdScr(height=30, width=80)
        # Pass the mock screen to the game instance
        self.game = FlappyBirdGame(self.mock_scr)
        # Reset necessary game state components for isolation if needed
        self.game.bird_y = self.mock_scr.height // 2
        self.game.bird_vel = 0
        self.game.pipes = []
        self.game.score = 0
        self.game.frames = 0
```

```
        self.game.game_over = False
        self.game.paused = False
        self.game.bird_x = self.mock_scr.width // 4 # Set
           consistent bird_x


    def test_initial_state(self):
        """Test the initial state of the game variables."""
        self.assertEqual(self.game.bird_y, self.mock_scr.height
           // 2)
        self.assertEqual(self.game.bird_vel, 0)
        self.assertEqual(self.game.score, 0)
        self.assertFalse(self.game.game_over)
        self.assertFalse(self.game.paused)
        self.assertEqual(len(self.game.pipes), 0)

    def test_flap(self):
        """Test the bird flap mechanism."""
        initial_vel = self.game.bird_vel
        self.game._flap()
        # Flap strength is negative (upward)
        self.assertEqual(self.game.bird_vel, FLAP_STRENGTH)
        self.assertLess(self.game.bird_vel, initial_vel)

    def test_bird_fall_gravity(self):
        """Test gravity affecting the bird's velocity and
           position."""
        initial_y = self.game.bird_y
        initial_vel = self.game.bird_vel
        self.game._update_bird() # Simulate one frame of falling
        self.assertEqual(self.game.bird_vel, initial_vel +
           GRAVITY)
        self.assertEqual(self.game.bird_y, initial_y + self.game.
           bird_vel)

    def test_bird_flap_then_fall(self):
        """Test flapping followed by gravity."""
        self.game._flap()
        flap_vel = self.game.bird_vel
        initial_y = self.game.bird_y
        self.game._update_bird() # Apply gravity after flap
        self.assertEqual(self.game.bird_vel, flap_vel + GRAVITY)
        self.assertEqual(self.game.bird_y, initial_y + self.game.
           bird_vel)

    def test_pipe_spawn(self):
        """Test pipe generation."""
        self.assertEqual(len(self.game.pipes), 0)
        self.game._spawn_pipe()
        self.assertEqual(len(self.game.pipes), 1)
        pipe_x, gap_y, scored = self.game.pipes[0]
```

```
        self.assertEqual(pipe_x, self.mock_scr.width) # Should
           spawn at right edge
        self.assertTrue(MIN_PIPE_HEIGHT <= gap_y <= self.mock_scr
           .height - MIN_PIPE_HEIGHT - GAP_SIZE)
        self.assertFalse(scored)

    def test_pipe_movement(self):
        """Test that pipes move leftwards."""
        self.game._spawn_pipe()
        initial_pipe_x = self.game.pipes[0][0]
        self.game._update_pipes() # Simulate one frame
        self.assertEqual(len(self.game.pipes), 1) # Should still
           have the pipe
        self.assertEqual(self.game.pipes[0][0], initial_pipe_x -
           PIPE_SPEED)

    def test_pipe_removal(self):
        """Test that pipes are removed when they go off-screen.
           """
        # Add a pipe already far left
        self.game.pipes.append([-PIPE_WIDTH, self.mock_scr.height
            // 2, False])
        self.game._update_pipes() # Move it further left
        self.assertEqual(len(self.game.pipes), 0) # Should be
           removed

    def test_collision_floor(self):
        """Test collision with the floor."""
        self.game.bird_y = self.mock_scr.height - 1 # Place bird
           right above floor
        self.game.bird_vel = 1 # Make it move down
        self.assertFalse(self.game.game_over)
        self.game._update_bird() # Should hit the floor boundary
        self.assertTrue(self.game.game_over)

    def test_collision_ceiling(self):
        """Test collision with the ceiling."""
        self.game.bird_y = 1 # Place bird right below ceiling
        self.game.bird_vel = -1 # Make it move up
        self.assertFalse(self.game.game_over)
        self.game._update_bird() # Should hit the ceiling
           boundary
        self.assertTrue(self.game.game_over)
        # Test exact ceiling collision as well
        self.setUp() # Reset
        self.game.bird_y = 0
        self.assertFalse(self.game.game_over)
        self.game._update_bird() # Should trigger game over
           immediately if starting at 0
        self.assertTrue(self.game.game_over)
```

```
    def test_collision_pipe_top(self):
        """Test collision with the top part of a pipe."""
        pipe_x = self.game.bird_x
        gap_y = self.game.bird_y + 1 # Gap starts just below the bird
        self.game.pipes.append([pipe_x, gap_y, False])
        self.assertFalse(self.game.game_over)
        collision = self.game._check_collision()
        self.assertTrue(collision)
        self.assertTrue(self.game.game_over)

    def test_collision_pipe_bottom(self):
        """Test collision with the bottom part of a pipe."""
        pipe_x = self.game.bird_x
        # Place gap such that bird_y is below the gap start + gap size
        gap_y = self.game.bird_y - GAP_SIZE - 1
        self.game.pipes.append([pipe_x, gap_y, False])
        self.assertFalse(self.game.game_over)
        collision = self.game._check_collision()
        self.assertTrue(collision)
        self.assertTrue(self.game.game_over)

    def test_no_collision_in_gap(self):
        """Test that no collision occurs when the bird is in the pipe gap."""
        pipe_x = self.game.bird_x
        # Center the gap around the bird
        gap_y = self.game.bird_y - GAP_SIZE // 2
        self.game.pipes.append([pipe_x, gap_y, False])
        self.assertFalse(self.game.game_over)
        collision = self.game._check_collision()
        self.assertFalse(collision)
        self.assertFalse(self.game.game_over)

    def test_scoring(self):
        """Test score increment when passing a pipe."""
        # Place a pipe just to the right of the bird
        pipe_x = self.game.bird_x + 1
        gap_y = self.mock_scr.height // 2
        self.game.pipes.append([pipe_x, gap_y, False])
        self.assertEqual(self.game.score, 0)
        self.assertFalse(self.game.pipes[0][2]) # Not scored yet

        # Move pipe so its right edge passes the bird
        # Need to move it PIPE_WIDTH + 1 steps relative to bird's x
        for _ in range(PIPE_WIDTH + 1):
            # Ensure bird doesn't hit floor/ceiling during test
            self.game.bird_y = self.mock_scr.height // 2
```

```
             self.game._update_pipes() # Update pipe positions
                only

        self.assertEqual(self.game.score, 1)
        # Find the pipe (it might have moved index if others
           spawned)
        pipe_found = False
        for p in self.game.pipes:
            if p[0] < self.game.bird_x - PIPE_WIDTH: # Check if
               it's the passed pipe
                self.assertTrue(p[2]) # Should be marked as
                   scored
                pipe_found = True
                break
        # self.assertTrue(pipe_found) # Pipe might be removed if
           test runs long, focus on score


    def test_no_double_scoring(self):
        """Test that a pipe only increments the score once."""
        pipe_x = self.game.bird_x + 1
        gap_y = self.mock_scr.height // 2
        self.game.pipes.append([pipe_x, gap_y, False])
        self.assertEqual(self.game.score, 0)

        # Move pipe past bird
        for _ in range(PIPE_WIDTH + 1):
             self.game.bird_y = self.mock_scr.height // 2 # Keep
                bird stable
             self.game._update_pipes()

        self.assertEqual(self.game.score, 1)

        # Move pipe further left, score should not increase again
        self.game._update_pipes()
        self.game._update_pipes()
        self.assertEqual(self.game.score, 1)

    def test_pause_game(self):
        """Test pausing the game stops updates."""
        self.game.paused = True
        initial_y = self.game.bird_y
        initial_vel = self.game.bird_vel
        self.game._spawn_pipe()
        initial_pipe_x = self.game.pipes[0][0]

        # Simulate game loop iterations while paused
        self.game._update_bird()
        self.game._update_pipes()
        collision = self.game._check_collision()
```

```
        # Bird position/velocity should not change
        self.assertEqual(self.game.bird_y, initial_y)
        self.assertEqual(self.game.bird_vel, initial_vel)
        # Pipe position should not change
        self.assertEqual(self.game.pipes[0][0], initial_pipe_x)
        # No collision should be detected even if bird is in pipe
            path
        self.assertFalse(collision)
        self.assertFalse(self.game.game_over) # Game should not
           end while paused

    def test_reset_game(self):
        """Test resetting the game after game over."""
        # Simulate game over state
        self.game.game_over = True
        self.game.score = 10
        self.game.bird_y = 5
        self.game.bird_vel = 5
        self.game.pipes.append([10, 10, True]) # Add a dummy
           scored pipe

        self.game._reset_game()

        self.assertFalse(self.game.game_over)
        self.assertFalse(self.game.paused)
        self.assertEqual(self.game.score, 0)
        self.assertEqual(self.game.bird_y, self.mock_scr.height
           // 2)
        self.assertEqual(self.game.bird_vel, 0)
        # Pipes should be reset (usually starts with one new pipe
           )
        self.assertEqual(len(self.game.pipes), 1) # _reset calls
           _spawn_pipe
        self.assertEqual(self.game.pipes[0][0], self.mock_scr.
           width) # New pipe at edge
        self.assertFalse(self.game.pipes[0][2]) # New pipe not
           scored


# Main execution block for tests
def main_tests():
    """Entry point for the test script."""
    # If running tests, ensure curses doesn't interfere if
       main_game was run before
    try:
        curses.endwin()
    except:
        pass
    unittest.main(exit=False) # exit=False prevents unittest from
        exiting the script
```

```
if __name__ == "__main__":
    # This structure allows running either the game or the tests,
    # but standard execution will run the game.
    # To run tests, you’d typically execute this script with
       unittest discover
    # or specifically target the test execution part if needed.
    # For the evaluator requirement, we assume separate execution
        contexts.

    # Check if the script is being run specifically for testing (
       e.g., via an argument)
    # or rely on the external evaluator calling the appropriate
       main function.
    # Since the instructions imply separate execution, we
       prioritize the game’s main.
    # If you needed to run tests from this file directly, you
       might use args:
    # import sys
    # if len(sys.argv) > 1 and sys.argv[1] == ’test’:
    #     main_tests()
    # else:
    #     main()

    # Per instructions, assuming external runner calls main() or
       main_tests() contextually.
    # The default behavior when running ‘python your_script.py‘
       will be the game.
    # The evaluator presumably executes the code setting up a
       context where
    # either main() or main_tests() is called as needed.
    # Let’s keep the original game main() call here.
     # print("Running main game...") # Debug print
     main()

    # To ensure the test main exists as requested, define it but
       don’t call by default:
    # def run_tests_main():
    #    main_tests()

    # The evaluator will need to ‘exec‘ the code and then
       potentially call ‘main_tests()‘
    # or ‘main()‘ in the appropriate namespace. The ‘if __name__
       == "__main__":‘
    # block handles the direct execution case.
```

Listing 61: Test Code — GEMINI 2.5

GROK 3 Response

Generated Code

```
# -------------------- Main Game Script --------------------

import curses
import time
import random
from typing import List, Tuple

class FlappyBirdGame:
    def __init__(self, stdscr):
        self.stdscr = stdscr
        self.height, self.width = stdscr.getmaxyx()
        self.bird_y = self.height // 2
        self.bird_x = self.width // 3
        self.bird_velocity = 0
        self.gravity = 0.5
        self.flap_strength = -2
        self.pipes: List[Tuple[int, int]] = []
        self.pipe_gap = 5
        self.pipe_width = 3
        self.score = 0
        self.game_over = False
        self.setup_curses()

    def setup_curses(self):
        curses.curs_set(0)
        self.stdscr.timeout(100)
        curses.start_color()
        curses.init_pair(1, curses.COLOR_GREEN, curses.
           COLOR_BLACK)
        curses.init_pair(2, curses.COLOR_RED, curses.COLOR_BLACK)

    def draw(self):
        self.stdscr.clear()
        # Draw bird
        try:
            self.stdscr.addch(int(self.bird_y), self.bird_x, 'B',
                curses.color_pair(1))
        except curses.error:
            self.game_over = True
        # Draw pipes
        for pipe_x, gap_y in self.pipes:
            for y in range(self.height - 2):
                if y < gap_y or y >= gap_y + self.pipe_gap:
                    for dx in range(self.pipe_width):
                        try:
                            self.stdscr.addch(y, pipe_x + dx, '|'
                               , curses.color_pair(2))
                        except curses.error:
                            pass
```

```
        # Draw ground
        for x in range(self.width):
            try:
                self.stdscr.addch(self.height - 2, x, ’=’)
            except curses.error:
                pass
        # Draw score
        score_str = f"Score:␣{self.score}"
        try:
            self.stdscr.addstr(0, 0, score_str)
        except curses.error:
            pass
        self.stdscr.refresh()

    def update_bird(self):
        self.bird_velocity += self.gravity
        self.bird_y += self.bird_velocity
        if self.bird_y >= self.height - 2 or self.bird_y < 0:
            self.game_over = True

    def generate_pipe(self):
        gap_y = random.randint(self.pipe_gap, self.height - self.
           pipe_gap - 2)
        self.pipes.append((self.width - self.pipe_width, gap_y))

    def update_pipes(self):
        new_pipes = []
        for pipe_x, gap_y in self.pipes:
            new_x = pipe_x - 1
            if new_x >= -self.pipe_width:
                new_pipes.append((new_x, gap_y))
            if new_x + self.pipe_width == self.bird_x:
                self.score += 1
        self.pipes = new_pipes

    def check_collision(self):
        for pipe_x, gap_y in self.pipes:
            if pipe_x <= self.bird_x < pipe_x + self.pipe_width:
                if self.bird_y < gap_y or self.bird_y >= gap_y +
                   self.pipe_gap:
                    self.game_over = True
                    return

    def flap(self):
        self.bird_velocity = self.flap_strength

    def run(self):
        last_pipe_time = time.time()
        while not self.game_over:
            self.draw()
            self.update_bird()
```

```
            self.update_pipes()
            self.check_collision()
            if time.time() - last_pipe_time > 2:
                self.generate_pipe()
                last_pipe_time = time.time()
            try:
                key = self.stdscr.getch()
                if key == ord('␣'):
                    self.flap()
            except curses.error:
                pass
        self.stdscr.clear()
        game_over_str = f"Game␣Over!␣Score:␣{self.score}"
        try:
            self.stdscr.addstr(self.height // 2, self.width // 2
               - len(game_over_str) // 2, game_over_str)
            self.stdscr.refresh()
            time.sleep(2)
        except curses.error:
            pass

def main():
    curses.wrapper(FlappyBirdGame)

if __name__ == "__main__":
    main()
```

Listing 62: Generated Code — GROK 3

Test Code

```
import unittest
from unittest.mock import MagicMock
from typing import List, Tuple

# Note: Since the main script is in the same file, we assume
   FlappyBirdGame is accessible.
# In a real scenario, it would be imported. For the evaluator,
   this works as is.

class TestFlappyBirdGame(unittest.TestCase):
    def setUp(self):
        # Mock the curses stdscr to avoid UI dependencies
        self.mock_stdscr = MagicMock()
        self.mock_stdscr.getmaxyx.return_value = (20, 60)  #
           height=20, width=60
        self.game = FlappyBirdGame(self.mock_stdscr)
        # Override setup_curses to avoid curses-specific calls
        self.game.setup_curses = MagicMock()

    def test_initial_state(self):
        self.assertEqual(self.game.bird_y, 10)  # height // 2
```

```
        self.assertEqual(self.game.bird_x, 20)  # width // 3
        self.assertEqual(self.game.bird_velocity, 0)
        self.assertEqual(self.game.score, 0)
        self.assertFalse(self.game.game_over)
        self.assertEqual(len(self.game.pipes), 0)

    def test_flap(self):
        self.game.flap()
        self.assertEqual(self.game.bird_velocity, -2)

    def test_update_bird_gravity(self):
        initial_y = self.game.bird_y
        initial_velocity = self.game.bird_velocity
        self.game.update_bird()
        self.assertEqual(self.game.bird_velocity,
           initial_velocity + self.game.gravity)
        self.assertEqual(self.game.bird_y, initial_y +
           initial_velocity + self.game.gravity)

    def test_update_bird_ground_collision(self):
        self.game.bird_y = self.game.height - 2
        self.game.update_bird()
        self.assertTrue(self.game.game_over)

    def test_update_bird_ceiling_collision(self):
        self.game.bird_y = -1
        self.game.bird_velocity = -1
        self.game.update_bird()
        self.assertTrue(self.game.game_over)

    def test_generate_pipe(self):
        self.game.generate_pipe()
        self.assertEqual(len(self.game.pipes), 1)
        pipe_x, gap_y = self.game.pipes[0]
        self.assertEqual(pipe_x, self.game.width - self.game.
           pipe_width)
        self.assertTrue(self.game.pipe_gap <= gap_y <= self.game.
           height - self.game.pipe_gap - 2)

    def test_update_pipes_movement(self):
        self.game.pipes = [(30, 10)]
        self.game.update_pipes()
        self.assertEqual(self.game.pipes, [(29, 10)])

    def test_update_pipes_score(self):
        self.game.bird_x = 20
        self.game.pipes = [(23, 10)]  # pipe_x + pipe_width = 23
           + 3 = 26
        self.game.update_pipes()
        self.assertEqual(self.game.score, 0)
```

```
        self.game.pipes = [(20, 10)]  # pipe_x + pipe_width = 20 + 3 = 23
        self.game.update_pipes()
        self.assertEqual(self.game.score, 1)

    def test_update_pipes_remove_offscreen(self):
        self.game.pipes = [(-4, 10)]
        self.game.update_pipes()
        self.assertEqual(self.game.pipes, [])

    def test_check_collision_no_pipes(self):
        self.game.check_collision()
        self.assertFalse(self.game.game_over)

    def test_check_collision_no_hit(self):
        self.game.bird_x = 20
        self.game.bird_y = 10
        self.game.pipes = [(20, 8)]  # gap from y=8 to y=13 (gap=5)
        self.game.check_collision()
        self.assertFalse(self.game.game_over)

    def test_check_collision_hit_upper(self):
        self.game.bird_x = 20
        self.game.bird_y = 7
        self.game.pipes = [(20, 8)]  # gap from y=8 to y=13
        self.game.check_collision()
        self.assertTrue(self.game.game_over)

    def test_check_collision_hit_lower(self):
        self.game.bird_x = 20
        self.game.bird_y = 14
        self.game.pipes = [(20, 8)]  # gap from y=8 to y=13
        self.game.check_collision()
        self.assertTrue(self.game.game_over)

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 63: Test Code — GROK 3

Prompt #8: Next Day

Prompt:
You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked with developing a script that, given a date (day, month and year) through the input console, prints the next day.
Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone; they are run as follows: `exec(main_code.py, ns)`; `exec(test_code.py, ns)`. Each, the tests and script classes must have its own guard: `if __name__ == "__main__": main()`.

Reference Code

```
# Prompt the user to input a year and convert it to an integer,
   assigning it to the variable ’year’
year = int(input("Input␣a␣year:␣"))

# Determine if the input year is a leap year

# Check if the year is divisible by 400
if (year % 400 == 0):
    leap_year = True
# Check if the year is divisible by 100
elif (year % 100 == 0):
    leap_year = False
# Check if the year is divisible by 4
elif (year % 4 == 0):
    leap_year = True
else:
    leap_year = False

# Prompt the user to input a month in the range of 1-12 and
   convert it to an integer, assigning it to the variable ’month’
month = int(input("Input␣a␣month␣[1-12]:␣"))

# Determine the number of days in the specified month

# Check for months with 31 days
if month in (1, 3, 5, 7, 8, 10, 12):
    month_length = 31
# Check for February
elif month == 2:
    # Check for leap year and assign the appropriate number of
      days
    if leap_year:
```

```
        month_length = 29
    else:
        month_length = 28
# For months with 30 days
else:
    month_length = 30

# Prompt the user to input a day in the range of 1-31 and convert
    it to an integer, assigning it to the variable 'day'
day = int(input("Input␣a␣day␣[1-31]:␣"))

# Calculate the next date based on the provided day, month, and
   year

# Check if the day is less than the total number of days in the
   month
if day < month_length:
    day += 1
else:
    day = 1
    # Check if the current month is December
    if month == 12:
        month = 1
        year += 1
    else:
        month += 1

# Display the next date in the format [yyyy-mm-dd] based on the
   updated day, month, and year
print("The␣next␣date␣is␣[yyyy-mm-dd]␣%d-%d-%d." % (year, month,
   day))
```

Listing 64: Reference Code — GPT4o

GPT4o Response

Generated Code

```
from datetime import datetime, timedelta

class DateCalculator:
    @staticmethod
    def next_day(day: int, month: int, year: int) -> str:
        try:
            current_date = datetime(year, month, day)
            next_date = current_date + timedelta(days=1)
            return next_date.strftime("%d-%m-%Y")
        except ValueError as e:
            return f"Invalid␣date:␣{e}"

def main():
    try:
```

```
        day = int(input("Enter␣day␣(1-31):␣"))
        month = int(input("Enter␣month␣(1-12):␣"))
        year = int(input("Enter␣year␣(e.g.␣2025):␣"))
        result = DateCalculator.next_day(day, month, year)
        print("Next␣day␣is:", result)
    except ValueError:
        print("Please␣enter␣valid␣integers␣for␣day,␣month,␣and␣
           year.")

if __name__ == "__main__":
    main()
```

Listing 65: Generated Code — GPT4o

Test Code

```
import unittest
from datetime import datetime, timedelta

# Re-declaring the class here to make the test file standalone
class DateCalculator:
    @staticmethod
    def next_day(day: int, month: int, year: int) -> str:
        try:
            current_date = datetime(year, month, day)
            next_date = current_date + timedelta(days=1)
            return next_date.strftime("%d-%m-%Y")
        except ValueError as e:
            return f"Invalid␣date:␣{e}"

class TestDateCalculator(unittest.TestCase):
    def test_regular_day(self):
        self.assertEqual(DateCalculator.next_day(10, 6, 2025), "
           11-06-2025")

    def test_end_of_month(self):
        self.assertEqual(DateCalculator.next_day(31, 1, 2023), "
           01-02-2023")

    def test_end_of_february_non_leap(self):
        self.assertEqual(DateCalculator.next_day(28, 2, 2023), "
           01-03-2023")

    def test_end_of_february_leap(self):
        self.assertEqual(DateCalculator.next_day(29, 2, 2024), "
           01-03-2024")

    def test_end_of_year(self):
        self.assertEqual(DateCalculator.next_day(31, 12, 2024), "
           01-01-2025")

    def test_invalid_date(self):
```

```
        result = DateCalculator.next_day(31, 2, 2023)
        self.assertTrue("Invalid␣date" in result)

    def test_negative_day(self):
        result = DateCalculator.next_day(-5, 1, 2023)
        self.assertTrue("Invalid␣date" in result)

    def test_zero_month(self):
        result = DateCalculator.next_day(5, 0, 2023)
        self.assertTrue("Invalid␣date" in result)

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 66: Test Code — GPT4o

GPT o4-mini-high Response

Generated Code

```
# ===== main_code.py =====

def is_leap_year(year):
    """
    Returns True if the given year is a leap year in the
       Gregorian calendar.
    """
    return (year % 400 == 0) or (year % 4 == 0 and year % 100 !=
       0)


def days_in_month(month, year):
    """
    Returns the number of days in the given month and year.
    """
    if month == 2:
        return 29 if is_leap_year(year) else 28
    if month in {1, 3, 5, 7, 8, 10, 12}:
        return 31
    if month in {4, 6, 9, 11}:
        return 30
    raise ValueError(f"Invalid␣month:␣{month}")


def next_day(day, month, year):
    """
    Given a date specified by day, month, year, returns a tuple (
       next_day, next_month, next_year).
    """
    # Validate inputs
    if year < 1:
        raise ValueError(f"Invalid␣year:␣{year}")
    if not (1 <= month <= 12):
        raise ValueError(f"Invalid␣month:␣{month}")
    dim = days_in_month(month, year)
    if not (1 <= day <= dim):
        raise ValueError(f"Invalid␣day:␣{day}␣for␣month␣{month},␣
           year␣{year}")

    # Compute next day
    if day < dim:
        return (day + 1, month, year)
    else:
        # End of month
        if month < 12:
            return (1, month + 1, year)
        else:
            # End of year
            return (1, 1, year + 1)
```

```
def main():
    """
    Reads three integers (day, month, year) from standard input (
        separated by spaces),
    computes the next day, and prints it in the format "day month
        year".
    """
    try:
        day, month, year = map(int, input().split())
    except Exception:
        print("Please␣provide␣three␣integers:␣day␣month␣year",
           file=sys.stderr)
        return

    try:
        nd, nm, ny = next_day(day, month, year)
        print(f"{nd}␣{nm}␣{ny}")
    except ValueError as e:
        print(f"Error:␣{e}", file=sys.stderr)


if __name__ == "__main__":
    main()
```

Listing 67: Generated Code — GPTo4-mini-high

Test Code

```
# ===== test_code.py =====

import unittest
import sys

class TestDateUtilities(unittest.TestCase):
    def test_leap_years(self):
        # Known leap years
        self.assertTrue(is_leap_year(2000))
        self.assertTrue(is_leap_year(2024))
        # Known non-leap years
        self.assertFalse(is_leap_year(1900))
        self.assertFalse(is_leap_year(2023))

    def test_days_in_month(self):
        # February in leap and non-leap years
        self.assertEqual(days_in_month(2, 2020), 29)
        self.assertEqual(days_in_month(2, 2019), 28)
        # 30-day months
        for m in [4, 6, 9, 11]:
            self.assertEqual(days_in_month(m, 2025), 30)
        # 31-day months
```

```
        for m in [1, 3, 5, 7, 8, 10, 12]:
            self.assertEqual(days_in_month(m, 2025), 31)
        # Invalid month
        with self.assertRaises(ValueError):
            days_in_month(13, 2025)

    def test_next_day_normal(self):
        self.assertEqual(next_day(14, 5, 2025), (15, 5, 2025))
        self.assertEqual(next_day(1, 1, 2025), (2, 1, 2025))

    def test_next_day_end_of_month(self):
        self.assertEqual(next_day(30, 4, 2025), (1, 5, 2025))
        self.assertEqual(next_day(31, 1, 2025), (1, 2, 2025))

    def test_next_day_leap_february(self):
        # Feb 28 on leap year
        self.assertEqual(next_day(28, 2, 2024), (29, 2, 2024))
        # Feb 29 on leap year
        self.assertEqual(next_day(29, 2, 2024), (1, 3, 2024))
        # Feb 28 on non-leap year
        self.assertEqual(next_day(28, 2, 2023), (1, 3, 2023))

    def test_next_day_end_of_year(self):
        self.assertEqual(next_day(31, 12, 2025), (1, 1, 2026))

    def test_invalid_inputs(self):
        # Invalid days
        with self.assertRaises(ValueError):
            next_day(0, 1, 2025)
        with self.assertRaises(ValueError):
            next_day(32, 1, 2025)
        # Invalid month
        with self.assertRaises(ValueError):
            next_day(10, 13, 2025)
        # Invalid year
        with self.assertRaises(ValueError):
            next_day(10, 10, 0)


def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 68: Test Code — GPTo4-mini-high

GEMINI 2.5 Response

Generated Code

```
import datetime

def get_next_day(year, month, day):
    """
    Calculates the next day given a specific date.

    Args:
        year (int): The year.
        month (int): The month.
        day (int): The day.

    Returns:
        tuple: A tuple containing the year, month, and day of the
            next day.
    """
    try:
        current_date = datetime.date(year, month, day)
        next_day = current_date + datetime.timedelta(days=1)
        return next_day.year, next_day.month, next_day.day
    except ValueError:
        return None

def main():
    """
    Main function to get user input and print the next day.
    """
    try:
        day = int(input("Enter␣the␣day␣(1-31):␣"))
        month = int(input("Enter␣the␣month␣(1-12):␣"))
        year = int(input("Enter␣the␣year:␣"))

        result = get_next_day(year, month, day)

        if result:
            next_year, next_month, next_day = result
            print(f"The␣next␣day␣is:␣{next_day:02d}/{next_month
               :02d}/{next_year}")
        else:
            print("Invalid␣date␣provided.")
    except ValueError:
        print("Invalid␣input.␣Please␣enter␣valid␣integer␣values␣
           for␣day,␣month,␣and␣year.")

if __name__ == "__main__":
    main()
```

Listing 69: Generated Code — GEMINI 2.5

Test Code

```python
import unittest
from io import StringIO
import sys

# Assuming the functions get_next_day and main are defined in the same file
# or imported. For this self-contained script, they are accessible.

class TestNextDay(unittest.TestCase):

    def test_standard_day(self):
        """Test a standard day in the middle of the month."""
        self.assertEqual(get_next_day(2024, 6, 10), (2024, 6, 11))

    def test_end_of_month_31_days(self):
        """Test the transition at the end of a 31-day month."""
        self.assertEqual(get_next_day(2024, 1, 31), (2024, 2, 1))

    def test_end_of_month_30_days(self):
        """Test the transition at the end of a 30-day month."""
        self.assertEqual(get_next_day(2024, 4, 30), (2024, 5, 1))

    def test_end_of_february_leap_year(self):
        """Test the end of February in a leap year."""
        self.assertEqual(get_next_day(2024, 2, 28), (2024, 2, 29))
        self.assertEqual(get_next_day(2024, 2, 29), (2024, 3, 1))

    def test_end_of_february_common_year(self):
        """Test the end of February in a common year."""
        self.assertEqual(get_next_day(2023, 2, 28), (2023, 3, 1))

    def test_end_of_year(self):
        """Test the transition from December 31st to January 1st."""
        self.assertEqual(get_next_day(2023, 12, 31), (2024, 1, 1))

    def test_invalid_date(self):
        """Test with an invalid date (e.g., September 31st)."""
        self.assertIsNone(get_next_day(2024, 9, 31))

    def test_invalid_month(self):
        """Test with an invalid month."""
        self.assertIsNone(get_next_day(2024, 13, 1))

    def test_main_function_valid_input(self):
        """Test the main function with valid user input."""
        user_input = "10\n6\n2024\n"
```

```
        expected_output = "The␣next␣day␣is:␣11/06/2024\n"

        original_stdin = sys.stdin
        original_stdout = sys.stdout
        sys.stdin = StringIO(user_input)
        sys.stdout = captured_output = StringIO()

        main()

        sys.stdin = original_stdin
        sys.stdout = original_stdout

        self.assertEqual(captured_output.getvalue(),
           expected_output)

    def test_main_function_invalid_date(self):
        """Test the main function with an invalid date input."""
        user_input = "31\n9\n2024\n"
        expected_output = "Invalid␣date␣provided.\n"

        original_stdin = sys.stdin
        original_stdout = sys.stdout
        sys.stdin = StringIO(user_input)
        sys.stdout = captured_output = StringIO()

        main()

        sys.stdin = original_stdin
        sys.stdout = original_stdout

        self.assertEqual(captured_output.getvalue(),
           expected_output)

    def test_main_function_non_integer_input(self):
        """Test the main function with non-integer input."""
        user_input = "abc\n6\n2024\n"
        expected_output = "Invalid␣input.␣Please␣enter␣valid␣
           integer␣values␣for␣day,␣month,␣and␣year.\n"

        original_stdin = sys.stdin
        original_stdout = sys.stdout
        sys.stdin = StringIO(user_input)
        sys.stdout = captured_output = StringIO()

        main()

        sys.stdin = original_stdin
        sys.stdout = original_stdout

        self.assertEqual(captured_output.getvalue(),
           expected_output)
```

```
def run_tests():
    """Function to run the unit tests."""
    suite = unittest.TestSuite()
    suite.addTest(unittest.makeSuite(TestNextDay))
    runner = unittest.TextTestRunner()
    runner.run(suite)

if __name__ == "__main__":
    run_tests()
```

Listing 70: Test Code — GEMINI 2.5

GROK 3 Response

Generated Code

```
# Main script to calculate the next day from a given date
def is_leap_year(year):
    return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)

def days_in_month(month, year):
    if month in [4, 6, 9, 11]:
        return 30
    elif month == 2:
        return 29 if is_leap_year(year) else 28
    else:
        return 31

def is_valid_date(day, month, year):
    if not (1 <= month <= 12):
        return False
    if not (1 <= year <= 9999):
        return False
    if not (1 <= day <= days_in_month(month, year)):
        return False
    return True

def next_day(day, month, year):
    if not is_valid_date(day, month, year):
        return None

    # Check if it's the last day of the month
    if day == days_in_month(month, year):
        day = 1
        # Check if it's December
        if month == 12:
            month = 1
            year += 1
        else:
            month += 1
    else:
        day += 1

    return day, month, year

def main():
    try:
        # Get input from console
        day = int(input("Enter␣day:␣"))
        month = int(input("Enter␣month:␣"))
        year = int(input("Enter␣year:␣"))

        result = next_day(day, month, year)

        if result is None:
```

```
                print("Invalid date")
            else:
                next_d, next_m, next_y = result
                print(f"Next day: {next_d}/{next_m}/{next_y}")

    except ValueError:
        print("Invalid input: Please enter numeric values")

if __name__ == "__main__":
    main()
```

Listing 71: Generated Code — GROK 3

Test Code

```
# Unit tests section
import unittest

class TestNextDay(unittest.TestCase):
    def test_regular_day(self):
        self.assertEqual(next_day(15, 6, 2023), (16, 6, 2023))

    def test_last_day_of_month(self):
        self.assertEqual(next_day(30, 6, 2023), (1, 7, 2023))

    def test_last_day_of_year(self):
        self.assertEqual(next_day(31, 12, 2023), (1, 1, 2024))

    def test_february_non_leap(self):
        self.assertEqual(next_day(28, 2, 2023), (1, 3, 2023))

    def test_february_leap(self):
        self.assertEqual(next_day(28, 2, 2024), (29, 2, 2024))
        self.assertEqual(next_day(29, 2, 2024), (1, 3, 2024))

    def test_invalid_day(self):
        self.assertIsNone(next_day(31, 4, 2023))
        self.assertIsNone(next_day(0, 6, 2023))

    def test_invalid_month(self):
        self.assertIsNone(next_day(15, 13, 2023))
        self.assertIsNone(next_day(15, 0, 2023))

    def test_invalid_year(self):
        self.assertIsNone(next_day(15, 6, 0))
        self.assertIsNone(next_day(15, 6, 10000))

    def test_is_leap_year(self):
        self.assertTrue(is_leap_year(2024))
        self.assertFalse(is_leap_year(2023))
        self.assertTrue(is_leap_year(2000))
        self.assertFalse(is_leap_year(1900))
```

```
    def test_days_in_month(self):
        self.assertEqual(days_in_month(2, 2024), 29)
        self.assertEqual(days_in_month(2, 2023), 28)
        self.assertEqual(days_in_month(4, 2023), 30)
        self.assertEqual(days_in_month(1, 2023), 31)

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 72: Test Code — GROK 3

Prompt #9: QR Generator

Prompt:
You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked with developing a simple QR code generator given text `input`. Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone; they are run as follows: `exec(main_code.py, ns)`; `exec(test_code.py, ns)`. Each, the tests and script classes must have its own guard: `if __name__ == "__main__": main()`.

Reference Code

```
import pyqrcode, png
# from pyqrcode import QRCode
# no need to import same library again and again

# Creating QR code after given text "input"
url = pyqrcode.create(input("Enter␣text␣to␣convert:␣"))
# Saving QR code as a png file
url.show()
# Name of QR code png file "input"
url.png(input("Enter␣image␣name␣to␣save:␣") + ".png", scale=6)
```

Listing 73: Reference Code — GPT4o

GPT4o Response

Generated Code

```
# main_code.py
import qrcode

class QRCodeGenerator:
    def __init__(self):
        pass

    def generate_qr(self, text_input: str, output_path: str = "
       qrcode.png") -> str:
        """
        Generates a QR code from the given text input and saves
           it as an image.

        Parameters:
            text_input (str): The text to encode in the QR code.
            output_path (str): The file path to save the QR code
               image.
```

```
        Returns:
            str: The path where the QR code image is saved.
        """
        if not isinstance(text_input, str) or not text_input.
           strip():
            raise ValueError("Input␣must␣be␣a␣non-empty␣string.")

        qr = qrcode.QRCode(
            version=1,
            error_correction=qrcode.constants.ERROR_CORRECT_L,
            box_size=10,
            border=4
        )
        qr.add_data(text_input)
        qr.make(fit=True)

        img = qr.make_image(fill="black", back_color="white")
        img.save(output_path)
        return output_path

def main():
    generator = QRCodeGenerator()
    text = input("Enter␣text␣to␣generate␣QR␣code:␣")
    path = generator.generate_qr(text)
    print(f"QR␣Code␣saved␣to␣{path}")

if __name__ == "__main__":
    main()
```

Listing 74: Generated Code — GPT4o

Test Code

```
# test_code.py
import os
import unittest

class TestQRCodeGenerator(unittest.TestCase):


    def tearDown(self):
        if os.path.exists(self.test_output):
            os.remove(self.test_output)

    def test_generate_valid_qr(self):
        result_path = self.generator.generate_qr("Hello␣World",
           self.test_output)
        self.assertTrue(os.path.exists(result_path))
        self.assertEqual(result_path, self.test_output)

    def test_generate_empty_string(self):
```

```
        with self.assertRaises(ValueError):
            self.generator.generate_qr("")

    def test_generate_non_string_input(self):
        with self.assertRaises(ValueError):
            self.generator.generate_qr(12345)

    def test_generate_whitespace_string(self):
        with self.assertRaises(ValueError):
            self.generator.generate_qr("␣␣␣␣␣")

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 75: Test Code — GPT4o

GPT o4-mini-high Response

Generated Code

```
# main_code.py

#!/usr/bin/env python3
"""
Simple QR Code generator.
Prompts the user to enter text in the console and writes a QR
    code image to ’qrcode.png’.
"""

import sys
import qrcode


def generate_qr(text, output_file=’qrcode.png’):
    """
    Generate a QR code image from the provided text and save it
        to output_file.
    Returns the PIL Image object.
    """
    if not isinstance(text, str):
        raise TypeError("Text␣must␣be␣a␣string.")
    img = qrcode.make(text)
    img.save(output_file)
    return img


def main():
    """
    Main entry point: prompt for text, generate QR code, and save
        it.
    """
    try:
        text = input("Enter␣text␣to␣encode␣as␣QR:␣").strip()
    except EOFError:
        text = ""
    if not text:
        print("Error:␣No␣input␣text␣provided.", file=sys.stderr)
        sys.exit(1)

    output_file = ’qrcode.png’
    try:
        generate_qr(text, output_file)
        print(f"QR␣code␣successfully␣saved␣to␣{output_file}")
    except Exception as e:
        print(f"Failed␣to␣generate␣QR␣code:␣{e}", file=sys.stderr
            )
        sys.exit(1)
```

```
if __name__ == "__main__":
    main()
```

Listing 76: Generated Code — GPTo4-mini-high

Test Code

```
# test_code.py

#!/usr/bin/env python3
"""
Unit tests for the QR Code generator.
Assumes generate_qr and main are loaded into globals via exec(
   main_code.py, ns).
"""

import os
import sys
import tempfile
import unittest
from io import BytesIO
from PIL import Image
from unittest.mock import patch


class TestQRCodeGenerator(unittest.TestCase):
    def test_generate_qr_saves_file_and_returns_image(self):
        # Create a temporary file path
        with tempfile.NamedTemporaryFile(suffix=".png", delete=
           False) as tmp:
            tmp_path = tmp.name
        try:
            img = generate_qr("Test␣QR", tmp_path)
            # verify return type
            self.assertIsInstance(img, Image.Image)
            # verify file exists
            self.assertTrue(os.path.exists(tmp_path))
            # verify PNG header
            with open(tmp_path, "rb") as f:
                header = f.read(8)
            self.assertEqual(header, b"\x89PNG\r\n\x1a\n")
        finally:
            os.remove(tmp_path)

    def test_type_error_for_non_string_text(self):
        with self.assertRaises(TypeError):
            generate_qr(12345, "dummy.png")

    def test_empty_string_generates_code(self):
        # Even empty string is valid QR (it encodes an empty
           payload)
```

```
        with tempfile.NamedTemporaryFile(suffix=".png", delete=
           False) as tmp:
            tmp_path = tmp.name
        try:
            img = generate_qr("", tmp_path)
            self.assertIsInstance(img, Image.Image)
            self.assertTrue(os.path.exists(tmp_path))
        finally:
            os.remove(tmp_path)

    def test_main_no_input_exits_with_error(self):
        # Patch input() to return empty
        with patch('builtins.input', return_value=""):
            saved_stdout, saved_stderr = sys.stdout, sys.stderr
            try:
                sys.stdout = BytesIO()
                sys.stderr = BytesIO()
                with self.assertRaises(SystemExit) as cm:
                    main()
                self.assertEqual(cm.exception.code, 1)
                err = sys.stderr.getvalue().decode()
                self.assertIn("Error:␣No␣input␣text␣provided",
                   err)
            finally:
                sys.stdout, sys.stderr = saved_stdout,
                   saved_stderr

    def test_main_successful(self):
        # Patch input() to return some text
        with patch('builtins.input', return_value="Hello␣QR"):
            saved_stdout, saved_stderr = sys.stdout, sys.stderr
            try:
                sys.stdout = BytesIO()
                sys.stderr = BytesIO()
                # ensure no leftover file
                if os.path.exists("qrcode.png"):
                    os.remove("qrcode.png")
                main()
                out = sys.stdout.getvalue().decode()
                self.assertIn("QR␣code␣successfully␣saved␣to␣
                   qrcode.png", out)
                self.assertTrue(os.path.exists("qrcode.png"))
            finally:
                sys.stdout, sys.stderr = saved_stdout,
                   saved_stderr
                if os.path.exists("qrcode.png"):
                    os.remove("qrcode.png")


def main():
    unittest.main()
```

```


if __name__ == "__main__":
    main()
```

Listing 77: Test Code — GPTo4-mini-high

GEMINI 2.5 Response

Generated Code

```
"""
This single Python script contains both a QR code generation
   application and its corresponding unit tests.
The script is divided into two main parts:
1. The QRCodeApp: Handles the generation of QR codes. Its main
   entry point function, main_script_entry(),
   when invoked, creates an instance of QRCodeApp and calls its
      main() method, which generates a
   QR code for the predefined text "input".
2. Unit Tests: A suite of tests for the QRCodeApp, using the
   unittest framework. Its main entry
   point function, main_tests_entry(), runs these tests.

Both parts are structured to have their respective entry point
   functions called via
'if __name__ == "__main__":' blocks. When this entire script is
   executed directly (e.g., `python your_script_name.py`),
the main application logic (main_script_entry) will run, followed
    by the unit tests (main_tests_entry).

The evaluator is expected to execute these parts, potentially by
   loading them into a shared
namespace and then calling the respective entry point functions (
   main_script_entry, main_tests_entry) by name.

Required library: qrcode (install with: pip install qrcode[pil])
Pillow (a PIL fork) is a dependency of qrcode for image
   operations.
"""

# Necessary imports for the main script
import qrcode
import os
import sys # Used by test runner for exit status (optional)

#
   #######################################################################

# #                             MAIN SCRIPT CODE
                                 #
#
   #######################################################################


class QRCodeApp:
    """
    A simple application to generate QR codes.
    This class can be considered the 'main class' for the script'
       s functionality.
```

```
    """
    def generate_qr_code(self, text: str, file_path: str = "qrcode.png") -> bool:
        """
        Generates a QR code from the given text and saves it as a PNG file.

        Args:
            text (str): The text to encode in the QR code.
            file_path (str): The path to save the generated QR code image.

        Returns:
            bool: True if QR code generation was successful, False otherwise.
        """
        try:
            # Create QR code instance
            qr = qrcode.QRCode(
                version=1,  # QR code version (1-40), controls complexity/size
                error_correction=qrcode.constants.ERROR_CORRECT_L, # Error correction: L, M, Q, H
                box_size=10, # Size of each box (pixel) in the QR code
                border=4,    # Thickness of the border (in boxes)
            )
            qr.add_data(text)
            qr.make(fit=True) # Fit the data to the QR code version

            # Create an image from the QR Code instance
            img = qr.make_image(fill_color="black", back_color="white")

            # Save the image
            img.save(file_path)
            return True
        except Exception as e:
            # In a real app, might use logging instead of print
            print(f"Error␣generating␣QR␣code␣for␣'{text[:20]}...':␣{e}", file=sys.stderr)
            return False

    def main(self):
        """
        Main method for the QRCodeApp class.
        This is the primary operational method of the 'main class'.
        It generates a QR code for the specific text "input" as per requirements.
```

```
        """
        text_to_encode = "input" # Requirement: "after given text
            ’input’"
        output_filename = "generated_qr_code_for_input.png"

        print(f"[QRCodeApp]␣Attempting␣to␣generate␣QR␣code␣for␣
           text:␣’{text_to_encode}’")
        print(f"[QRCodeApp]␣Output␣file:␣’{output_filename}’")

        success = self.generate_qr_code(text_to_encode,
           output_filename)

        if success:
            print(f"[QRCodeApp]␣Successfully␣generated␣’{
               output_filename}’")
        else:
            print(f"[QRCodeApp]␣Failed␣to␣generate␣’{
               output_filename}’")

def main_script_entry(): # This is the ’main()’ function for the
   script part
    """
    Entry point for the main QR code generation script logic.
    """
    app = QRCodeApp()
    app.main()

if __name__ == "__main__": # Standard guard for the main script
   part
    print("---␣Executing␣Main␣QR␣Code␣Script␣Logic␣---")
    main_script_entry()
    print("---␣Main␣QR␣Code␣Script␣Logic␣Finished␣---")
```

Listing 78: Generated Code — GEMINI 2.5

Test Code

```
import unittest
import tempfile
import shutil

# --- Test Class ---

#
   #######################################################################

# #                             UNIT TESTS
                                       #
#
   #######################################################################

```

```
# # This section contains the unit tests for the QRCodeApp.
                      #
# # It uses the ’unittest’ module.
                                                 #
#
   ######################################################################


import unittest
# ’os’ and ’sys’ are already imported above and will be available
    in this scope
# when the entire file is processed.

class TestQRCodeGeneratorFunctionality(unittest.TestCase):
    """
    Unit tests for the QRCodeApp.generate_qr_code method.
    This class is one of the ’main classes’ for the testing part.
    """
    def setUp(self):
        """
        Set up for each test method.
        This method is called before each test function.
        """
        self.qr_app = QRCodeApp() # Instance of the class to be
           tested
        self.test_file_path = "test_qr_code_output.png"
        # Ensure no pre-existing test file from a previous run
        if os.path.exists(self.test_file_path):
            os.remove(self.test_file_path)

    def tearDown(self):
        """
        Clean up after each test method.
        This method is called after each test function.
        """
        if os.path.exists(self.test_file_path):
            os.remove(self.test_file_path)

    def test_successful_generation(self):
        """Test successful QR code generation with standard text.
           """
        text = "Hello,␣QR␣World!"
        self.assertTrue(self.qr_app.generate_qr_code(text, self.
           test_file_path),
                        "generate_qr_code␣should␣return␣True␣on␣
                           success.")
        self.assertTrue(os.path.exists(self.test_file_path),
                        "QR␣code␣file␣should␣be␣created␣after␣
                           successful␣generation.")
        self.assertTrue(os.path.getsize(self.test_file_path) > 0,
```

```
                        "Generated␣QR␣code␣file␣should␣not␣be␣
                           empty.")

    def test_generation_for_empty_string(self):
        """Test QR code generation for an empty string."""
        # The ’qrcode’ library can generate a QR code for an
           empty string.
        text = ""
        self.assertTrue(self.qr_app.generate_qr_code(text, self.
           test_file_path),
                        "generate_qr_code␣should␣handle␣empty␣
                           string␣input.")
        self.assertTrue(os.path.exists(self.test_file_path),
                        "QR␣code␣file␣for␣empty␣string␣should␣be␣
                           created.")
        self.assertTrue(os.path.getsize(self.test_file_path) > 0,
                        "QR␣code␣file␣for␣empty␣string␣should␣not
                           ␣be␣empty.")

    def test_generation_for_long_string(self):
        """Test QR code generation with a longer string."""
        text = ("This␣is␣a␣rather␣long␣string␣designed␣to␣test␣
           the␣QR␣code␣generation␣"
                "capabilities␣for␣a␣somewhat␣larger␣amount␣of␣
                   data.␣The␣quick␣brown␣fox␣"
                "jumps␣over␣the␣lazy␣dog.␣" * 5)
        self.assertTrue(self.qr_app.generate_qr_code(text, self.
           test_file_path),
                        "generate_qr_code␣should␣handle␣long␣
                           strings.")
        self.assertTrue(os.path.exists(self.test_file_path),
                        "QR␣code␣file␣for␣long␣string␣should␣be␣
                           created.")
        self.assertTrue(os.path.getsize(self.test_file_path) > 0,
                        "QR␣code␣file␣for␣long␣string␣should␣not␣
                           be␣empty.")

    def test_generation_with_various_special_characters(self):
        """Test QR code generation with a string containing
           various special characters."""
        text = "!@#$%^&*()_+-=[]{};’:\",./<>?‘~\\|0123456789
           abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
        self.assertTrue(self.qr_app.generate_qr_code(text, self.
           test_file_path),
                        "generate_qr_code␣should␣handle␣strings␣
                           with␣special␣characters.")
        self.assertTrue(os.path.exists(self.test_file_path),
                        "QR␣code␣file␣for␣string␣with␣special␣
                           chars␣should␣be␣created.")
        self.assertTrue(os.path.getsize(self.test_file_path) > 0,
```

```
                        "QR␣code␣file␣for␣string␣with␣special␣
                           chars␣should␣not␣be␣empty.")

    def test_file_overwrite_capability(self):
        """Test that an existing file can be overwritten by a new
            QR code."""
        text1 = "First␣QR␣Code␣Version"
        self.assertTrue(self.qr_app.generate_qr_code(text1, self.
           test_file_path), "Initial␣generation␣failed.")
        self.assertTrue(os.path.exists(self.test_file_path), "
           Initial␣file␣not␣created.")
        # To robustly check overwrite, ideally content would be
           verified.
        # For simplicity, we check that generation succeeds again
           .
        text2 = "Second␣QR␣Code␣Version␣-␣Overwrite"
        self.assertTrue(self.qr_app.generate_qr_code(text2, self.
           test_file_path), "Overwrite␣generation␣failed.")
        self.assertTrue(os.path.exists(self.test_file_path), "
           File␣should␣still␣exist␣after␣overwrite.")

class TestMainScriptAppExecution(unittest.TestCase):
    """
    Tests for the main execution flow of the script part,
       specifically QRCodeApp.main().
    This class is another 'main class' for the testing part.
    """
    def setUp(self):
        self.app_instance = QRCodeApp()
        # This is the specific filename generated by QRCodeApp.
           main()
        self.expected_main_output_file = "
           generated_qr_code_for_input.png"
        if os.path.exists(self.expected_main_output_file):
            os.remove(self.expected_main_output_file)

    def tearDown(self):
        if os.path.exists(self.expected_main_output_file):
            os.remove(self.expected_main_output_file)

    def test_main_method_creates_file_for_literal_input_text(self
       ):
        """
        Test that QRCodeApp.main() (called via main_script_entry)
            correctly
        generates a QR code for the literal text 'input'.
        """
        # We call the main method of the app instance directly
           for this test.
        # Alternatively, we could call main_script_entry() but
           that adds print statements
```

```
        # not directly part of the QRCodeApp.main() functionality
           .
        self.app_instance.main()

        self.assertTrue(os.path.exists(self.
           expected_main_output_file),
                        f"The␣main␣app␣method␣should␣create␣'{
                           self.expected_main_output_file}'.")
        self.assertTrue(os.path.getsize(self.
           expected_main_output_file) > 0,
                        f"The␣file␣'{self.
                           expected_main_output_file}'␣should␣not␣
                           be␣empty.")

        # A more advanced test could involve a QR code reader to
           verify the content is indeed "input",
        # but for this scope, file existence and non-zero size
           are primary checks.

def main_tests_entry(): # This is the 'main()' function for the
   test part
    """
    Entry point for running all unit tests.
    """
    print("\n---␣Executing␣Unit␣Tests␣---")

    # Create a TestSuite by discovering tests
    suite = unittest.TestSuite()
    loader = unittest.TestLoader()
    suite.addTest(loader.loadTestsFromTestCase(
       TestQRCodeGeneratorFunctionality))
    suite.addTest(loader.loadTestsFromTestCase(
       TestMainScriptAppExecution))

    # Create a TextTestRunner to run the suite
    # verbosity=2 provides more detailed output
    runner = unittest.TextTestRunner(verbosity=2, stream=sys.
       stdout)
    result = runner.run(suite)

    # Optionally, exit with an error code if tests fail, useful
       for CI environments.
    # If the evaluator doesn't like sys.exit(), this part can be
       removed.
    # if not result.wasSuccessful():
    #     sys.exit(1) # A non-zero exit code indicates test
       failures.

if __name__ == "__main__": # Standard guard for the test script
   part
```

```
    # When this single script is run directly (python this_script
       .py):
    # 1. The first 'if __name__ == "__main__":' block (for
       main_script_entry) executes.
    # 2. Then, this second 'if __name__ == "__main__":' block (
       for main_tests_entry) executes.
    # This sequential execution is a consequence of the single-
       file structure with multiple
    # __main__ guards. The evaluator is expected to handle the
       invocation of script/test
    # entry points in a more controlled manner, likely by name
       after exec-ing the code.
    main_tests_entry()
    print("---␣Unit␣Tests␣Finished␣---")
```

Listing 79: Test Code — GEMINI 2.5

GROK 3 Response

Generated Code

```
import qrcode
from PIL import Image
import os
import io

class QRCodeGenerator:
    def __init__(self):
        self.qr = qrcode.QRCode(
            version=1,
            error_correction=qrcode.constants.ERROR_CORRECT_L,
            box_size=10,
            border=4,
        )

    def generate_qr_code(self, input_text, output_path="qrcode.
       png"):
        """Generate a QR code from input text and save it to
           output_path."""
        if not input_text:
            raise ValueError("Input␣text␣cannot␣be␣empty")

        self.qr.clear()
        self.qr.add_data(input_text)
        self.qr.make(fit=True)

        img = self.qr.make_image(fill_color="black", back_color="
           white")
        img.save(output_path)
        return output_path

    def generate_qr_code_bytes(self, input_text):
        """Generate a QR code and return it as bytes."""
        if not input_text:
            raise ValueError("Input␣text␣cannot␣be␣empty")

        self.qr.clear()
        self.qr.add_data(input_text)
        self.qr.make(fit=True)

        img = self.qr.make_image(fill_color="black", back_color="
           white")
        img_byte_arr = io.BytesIO()
        img.save(img_byte_arr, format='PNG')
        return img_byte_arr.getvalue()

def main():
    generator = QRCodeGenerator()
    # Example usage
    try:
```

```
        output_path = generator.generate_qr_code("https://example
           .com", "example_qr.png")
        print(f"QR␣code␣generated␣successfully␣at:␣{output_path}"
           )
    except Exception as e:
        print(f"Error␣generating␣QR␣code:␣{str(e)}")

if __name__ == "__main__":
    main()
```

Listing 80: Generated Code — GROK 3

Test Code

```
# Unit Tests
import unittest
import tempfile

class TestQRCodeGenerator(unittest.TestCase):
    def setUp(self):
        self.generator = QRCodeGenerator()
        self.temp_dir = tempfile.mkdtemp()

    def test_generate_qr_code_valid_input(self):
        output_path = os.path.join(self.temp_dir, "test_qr.png")
        result = self.generator.generate_qr_code("test␣input",
           output_path)
        self.assertEqual(result, output_path)
        self.assertTrue(os.path.exists(output_path))

        # Verify the image can be opened
        with Image.open(output_path) as img:
            self.assertEqual(img.format, 'PNG')

    def test_generate_qr_code_empty_input(self):
        with self.assertRaises(ValueError):
            self.generator.generate_qr_code("")

    def test_generate_qr_code_bytes_valid_input(self):
        result = self.generator.generate_qr_code_bytes("test␣
           input")
        self.assertIsInstance(result, bytes)
        self.assertGreater(len(result), 0)

        # Verify the bytes represent a valid PNG
        img = Image.open(io.BytesIO(result))
        self.assertEqual(img.format, 'PNG')

    def test_generate_qr_code_bytes_empty_input(self):
        with self.assertRaises(ValueError):
            self.generator.generate_qr_code_bytes("")
```

```
    def test_multiple_generations(self):
        output_path1 = os.path.join(self.temp_dir, "test_qr1.png"
           )
        output_path2 = os.path.join(self.temp_dir, "test_qr2.png"
           )

        self.generator.generate_qr_code("input1", output_path1)
        self.generator.generate_qr_code("input2", output_path2)

        self.assertTrue(os.path.exists(output_path1))
        self.assertTrue(os.path.exists(output_path2))

        # Verify images are different
        with Image.open(output_path1) as img1, Image.open(
           output_path2) as img2:
            self.assertNotEqual(img1.tobytes(), img2.tobytes())

def main_test():
    unittest.main()

if __name__ == "__main__":
    main_test()
```

Listing 81: Test Code — GROK 3

Prompt #10: Sort Positive Values

Prompt:
You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked with developing a script that, given a list, sorts first the positive numbers and then the negative numbers. Here is an example of execution: The original list is: [4, -8, -6, 3, -5, 8, 10, 5, -19] Result after performing sort operation: [3, 4, 5, 8, 10, -19, -8, -6, -5]. Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone; they are run as follows: `exec(main_code.py, ns); exec(test_code.py, ns)`. Each, the tests and script classes must have its own guard: `if __name__ == "__main__": main()`.

Reference Code

```python
# Python3 code to demonstrate working of
# Sort while keeping Positive elements before negatives
# using sorted() + lambda

# initialize list
test_list = [4, -8, -6, 3, -5, 8, 10, 5, -19]

# printing original list
print("The␣original␣list␣is␣:␣" + str(test_list))

# Sort while keeping Positive elements before negatives
# using sorted() + lambda
res = sorted(test_list, key = lambda i: 0 if i == 0 else -1 / i)

# printing result
print("Result␣after␣performing␣sort␣operation␣:␣" + str(res))
```

Listing 82: Reference Code — GPT4o

GPT4o Response

Generated Code

```python
# main_code.py

from typing import List

def sort_positive_then_negative(lst: List[int]) -> List[int]:
    positives = sorted([x for x in lst if x >= 0])
    negatives = sorted([x for x in lst if x < 0], reverse=True)
    return positives + negatives

```

```
def main():
    original_list = [4, -8, -6, 3, -5, 8, 10, 5, -19]
    result = sort_positive_then_negative(original_list)
    print("The␣original␣list␣is␣:", original_list)
    print("Result␣after␣performing␣sort␣operation␣:", result)

if __name__ == "__main__":
    main()
```

Listing 83: Generated Code — GPT4o

Test Code

```
# test_code.py

import unittest

class TestSortPositiveThenNegative(unittest.TestCase):

    def test_mixed_list(self):
        self.assertEqual(
            sort_positive_then_negative([4, -8, -6, 3, -5, 8, 10,
                5, -19]),
            [3, 4, 5, 8, 10, -19, -8, -6, -5]
        )

    def test_all_positive(self):
        self.assertEqual(
            sort_positive_then_negative([3, 1, 4, 1, 5]),
            [1, 1, 3, 4, 5]
        )

    def test_all_negative(self):
        self.assertEqual(
            sort_positive_then_negative([-5, -1, -3, -4]),
            [-1, -3, -4, -5]
        )

    def test_empty_list(self):
        self.assertEqual(
            sort_positive_then_negative([]),
            []
        )

    def test_duplicates(self):
        self.assertEqual(
            sort_positive_then_negative([1, -1, 1, -1, 2, -2]),
            [1, 1, 2, -1, -1, -2]
        )

    def test_single_element_positive(self):
        self.assertEqual(
```

```
            sort_positive_then_negative([7]),
            [7]
        )

    def test_single_element_negative(self):
        self.assertEqual(
            sort_positive_then_negative([-7]),
            [-7]
        )

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 84: Test Code — GPT4o

GPT o4-mini-high Response

Generated Code

```
# === main_code.py ===

def sort_list(lst):
    """
    Sorts the list such that all non-negative numbers (zeros and
       positives)
    come first in ascending order, followed by all negative
       numbers in ascending order.
    """
    non_negative = sorted(x for x in lst if x >= 0)
    negative     = sorted(x for x in lst if x <  0)
    return non_negative + negative

def main():
    original = [4, -8, -6, 3, -5, 8, 10, 5, -19]
    print("The␣original␣list␣is␣:", original)
    result = sort_list(original)
    print("Result␣after␣performing␣sort␣operation␣:", result)

if __name__ == "__main__":
    main()
```

Listing 85: Generated Code — GPTo4-mini-high

Test Code

```
# === test_code.py ===

import unittest

# Assumes main_code.py has already been execd in the same
   namespace
# so that sort_list is available here.

class TestSortList(unittest.TestCase):
    def test_mixed(self):
        self.assertEqual(
            sort_list([4, -8, -6, 3, -5, 8, 10, 5, -19]),
            [3, 4, 5, 8, 10, -19, -8, -6, -5]
        )

    def test_empty(self):
        self.assertEqual(sort_list([]), [])

    def test_all_positive(self):
        self.assertEqual(sort_list([7, 1, 3, 2]), [1, 2, 3, 7])

    def test_all_negative(self):
        self.assertEqual(sort_list([-7, -1, -3, -2]), [-7, -3,
           -2, -1])
```

```
    def test_with_zero(self):
        # zeros are treated as non-negative and sorted among positives
        self.assertEqual(sort_list([0, -1, 5, 0, -3]), [0, 0, 5, -3, -1])

    def test_duplicates(self):
        self.assertEqual(sort_list([2, -2, 2, -2]), [2, 2, -2, -2])

    def test_single_positive(self):
        self.assertEqual(sort_list([42]), [42])

    def test_single_negative(self):
        self.assertEqual(sort_list([-42]), [-42])

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 86: Test Code — GPTo4-mini-high

GEMINI 2.5 Response

Generated Code

```
import sys
import io

# Main Code
class CustomSort:
    """
    Sorts a list by positive then negative numbers.
    """

    def sort_positive_then_negative(self, numbers):
        """
        Sorts a list of numbers with positives first, then
           negatives.

        Args:
            numbers: A list of integers.

        Returns:
            A new list with positive numbers sorted, followed by
               negative numbers sorted.
        """
        positive_numbers = sorted([num for num in numbers if num
           >= 0])
        negative_numbers = sorted([num for num in numbers if num
           < 0])
        return positive_numbers + negative_numbers

def main():
    """
    Main function to demonstrate the sorting logic.
    """
    sorter = CustomSort()
    original_list = [4, -8, -6, 3, -5, 8, 10, 5, -19]
    print(f"The␣original␣list␣is␣:␣{original_list}")
    sorted_list = sorter.sort_positive_then_negative(
       original_list)
    print(f"Result␣after␣performing␣sort␣operation␣:␣{sorted_list
       }")

if __name__ == "__main__":
    main()
```

Listing 87: Generated Code — GEMINI 2.5

Test Code

```
import unittest
import sys
import io
```

```python
# Unit Tests
class TestCustomSort(unittest.TestCase):
    """
    Unit tests for the CustomSort class.
    """

    def setUp(self):
        """
        Set up a new CustomSort object for each test.
        """
        self.sorter = CustomSort()

    def test_example_list(self):
        """
        Test with the provided example list.
        """
        input_list = [4, -8, -6, 3, -5, 8, 10, 5, -19]
        expected_output = [3, 4, 5, 8, 10, -19, -8, -6, -5]
        self.assertEqual(self.sorter.sort_positive_then_negative(
           input_list), expected_output)

    def test_empty_list(self):
        """
        Test with an empty list.
        """
        self.assertEqual(self.sorter.sort_positive_then_negative
           ([]), [])

    def test_all_positive(self):
        """
        Test with a list of only positive numbers.
        """
        self.assertEqual(self.sorter.sort_positive_then_negative
           ([5, 1, 3, 2, 4]), [1, 2, 3, 4, 5])

    def test_all_negative(self):
        """
        Test with a list of only negative numbers.
        """
        self.assertEqual(self.sorter.sort_positive_then_negative
           ([-5, -1, -3, -2, -4]), [-5, -4, -3, -2, -1])

    def test_with_zero(self):
        """
        Test with a list that includes zero.
        """
        input_list = [4, -1, 0, 3, -2]
        expected_output = [0, 3, 4, -2, -1]
        self.assertEqual(self.sorter.sort_positive_then_negative(
           input_list), expected_output)
```

```
    def test_already_sorted(self):
        """
        Test with a list that is already sorted according to the desired criteria.
        """
        input_list = [1, 2, 3, -3, -2, -1]
        self.assertEqual(self.sorter.sort_positive_then_negative(input_list), input_list)

def test_main():
    """
    Main function to run the unit tests.
    """
    # Redirect stdout to suppress test runner output during normal execution
    original_stdout = sys.stdout
    sys.stdout = io.StringIO()

    suite = unittest.TestSuite()
    suite.addTest(unittest.makeSuite(TestCustomSort))
    runner = unittest.TextTestRunner()
    runner.run(suite)

    # Restore stdout
    sys.stdout = original_stdout

if __name__ == "__main__":
    test_main()
```

Listing 88: Test Code — GEMINI 2.5

GROK 3 Response

Generated Code

```python
def sort_positive_negative(numbers):
    """
    Sort a list such that positive numbers are sorted in
       ascending order,
    followed by negative numbers sorted in ascending order.

    Args:
        numbers (list): List of integers (positive and negative)

    Returns:
        list: Sorted list with positive numbers first, then
           negative numbers
    """
    # Separate positive and negative numbers
    positive = [num for num in numbers if num >= 0]
    negative = [num for num in numbers if num < 0]

    # Sort both lists in ascending order
    positive.sort()
    negative.sort()

    # Combine the sorted lists
    return positive + negative

def main():
    # Example usage
    input_list = [4, -8, -6, 3, -5, 8, 10, 5, -19]
    result = sort_positive_negative(input_list)
    print(f"The␣original␣list␣is␣:␣{input_list}")
    print(f"Result␣after␣performing␣sort␣operation␣:␣{result}")

if __name__ == "__main__":
    main()
```

Listing 89: Generated Code — GROK 3

Test Code

```python
# Unit Tests
import unittest

class TestSortPositiveNegative(unittest.TestCase):
    def test_mixed_numbers(self):
        """Test with a mix of positive and negative numbers"""
        input_list = [4, -8, -6, 3, -5, 8, 10, 5, -19]
        expected = [3, 4, 5, 8, 10, -19, -8, -6, -5]
        self.assertEqual(sort_positive_negative(input_list),
           expected)

```

```
    def test_only_positive(self):
        """Test with only positive numbers"""
        input_list = [4, 3, 8, 10, 5]
        expected = [3, 4, 5, 8, 10]
        self.assertEqual(sort_positive_negative(input_list),
           expected)

    def test_only_negative(self):
        """Test with only negative numbers"""
        input_list = [-8, -6, -5, -19]
        expected = [-19, -8, -6, -5]
        self.assertEqual(sort_positive_negative(input_list),
           expected)

    def test_empty_list(self):
        """Test with an empty list"""
        input_list = []
        expected = []
        self.assertEqual(sort_positive_negative(input_list),
           expected)

    def test_single_number(self):
        """Test with a single number"""
        input_list = [5]
        expected = [5]
        self.assertEqual(sort_positive_negative(input_list),
           expected)
        input_list = [-5]
        expected = [-5]
        self.assertEqual(sort_positive_negative(input_list),
           expected)

    def test_zeros(self):
        """Test with zeros and other numbers"""
        input_list = [0, -1, 2, 0, -3, 4]
        expected = [0, 0, 2, 4, -3, -1]
        self.assertEqual(sort_positive_negative(input_list),
           expected)

    def test_duplicate_numbers(self):
        """Test with duplicate numbers"""
        input_list = [3, -2, 3, -2, 1, -1, 1]
        expected = [1, 1, 3, 3, -2, -2, -1]
        self.assertEqual(sort_positive_negative(input_list),
           expected)

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 90: Test Code — GROK 3

Prompt #11: Voice Repeater

Prompt:
You are a software developer tasked with implementing a complete solution for a given problem. Your objective is to generate a single, self-contained script in Python that solves the problem as described below, and to include a comprehensive suite of unit tests that verify its functionality. Below is your task description: You are tasked with developing a simple voice repeater. This voice repeater must listen so when the user runs the code and speaks something, the script will repeat exactly what the user said. It creates a folder and stores the user speech as a sound file and plays it. Some requirements for this script are `Python`, `SpeechRecognition` and `playsound`. Generate the complete script followed by its corresponding unit tests in a single output. Clearly separate the main code from the unit tests (for example, using comments or distinct sections) so that both parts can be easily identified. Both scripts should be standalone; they are run as follows: `exec(main_code.py, ns)`; `exec(test_code.py, ns)`. Each, the tests and script classes must have its own guard: `if __name__ == "__main__": main()`.

Reference Code

```
import time

import speech_recognition as sr
import os
import playsound
import shutil

shutil.rmtree("spoken")
os.mkdir("spoken")

speeches = []


def callback(recognizer, audio):
    with open("spoken/" + str(len(speeches)) + ".wav", "wb") as
       file:
        file.write(audio.get_wav_data())

    playsound.playsound("spoken/" + str(len(speeches)) + ".wav")
    speeches.append(1)
    print("____")


r = sr.Recognizer()
m = sr.Microphone()
with m as source:
    r.adjust_for_ambient_noise(source)

stop_listening = r.listen_in_background(m, callback)
print("say:")
```

```
while True:
    time.sleep(0.1)
```

Listing 91: Reference Code — GPT4o

GPT4o Response

Generated Code

```
# main_code.py

import os
import datetime
import speech_recognition as sr
from playsound import playsound

class VoiceRepeater:
    def __init__(self, folder_name="recordings"):
        self.folder_name = folder_name
        if not os.path.exists(self.folder_name):
            os.makedirs(self.folder_name)

    def record_and_save(self, filename=None):
        recognizer = sr.Recognizer()
        with sr.Microphone() as source:
            print("Listening...␣Speak␣now.")
            audio = recognizer.listen(source)
            print("Recording␣complete.")

        if not filename:
            timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
            filename = f"{self.folder_name}/recording_{timestamp}.wav"
        else:
            filename = os.path.join(self.folder_name, filename)

        with open(filename, "wb") as f:
            f.write(audio.get_wav_data())

        return filename

    def play_audio(self, filepath):
        print(f"Playing:␣{filepath}")
        playsound(filepath)

def main():
    repeater = VoiceRepeater()
    audio_file = repeater.record_and_save()
    repeater.play_audio(audio_file)

if __name__ == "__main__":
```

```
    main()
```

Listing 92: Generated Code — GPT4o

Test Code

```
# test_code.py

import os
import unittest
import shutil
from unittest import mock

class TestVoiceRepeater(unittest.TestCase):
    def setUp(self):
        self.folder = "test_recordings"
        self.repeater = VoiceRepeater(folder_name=self.folder)

    def tearDown(self):
        if os.path.exists(self.folder):
            shutil.rmtree(self.folder)

    @mock.patch("speech_recognition.Recognizer.listen")
    @mock.patch("speech_recognition.Microphone")
    def test_record_and_save_creates_file(self, mock_microphone,
       mock_listen):
        dummy_audio = mock.MagicMock()
        dummy_audio.get_wav_data.return_value = b"fake_wav_data"
        mock_listen.return_value = dummy_audio

        filepath = self.repeater.record_and_save(filename="test.
           wav")
        self.assertTrue(os.path.exists(filepath))
        with open(filepath, "rb") as f:
            self.assertEqual(f.read(), b"fake_wav_data")

    @mock.patch("playsound.playsound")
    def test_play_audio(self, mock_playsound):
        dummy_file = os.path.join(self.folder, "dummy.wav")
        with open(dummy_file, "wb") as f:
            f.write(b"dummy_audio")

        self.repeater.play_audio(dummy_file)
        mock_playsound.assert_called_once_with(dummy_file)

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 93: Test Code — GPT4o

GPT o4-mini-high Response

Generated Code

```
# ==========================
# main_code.py
# ==========================

import os
import datetime
import speech_recognition as sr
from playsound import playsound

def listen_and_repeat(folder="recordings"):
    """
    Listens to the microphone, saves the audio as a WAV file in ‘folder‘,
    plays it back, and returns the filename and recognized text.
    """
    # Ensure the recordings folder exists
    os.makedirs(folder, exist_ok=True)

    recognizer = sr.Recognizer()
    with sr.Microphone() as source:
        print("Listening...")
        audio = recognizer.listen(source)

    # Recognize speech using Google Web Speech API
    recognized_text = recognizer.recognize_google(audio)
    print(f"You␣said:␣{recognized_text}")

    # Save the raw WAV data
    timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
    filename = os.path.join(folder, f"recording_{timestamp}.wav")
    wav_data = audio.get_wav_data()
    with open(filename, "wb") as f:
        f.write(wav_data)

    # Play back the recorded file
    playsound(filename)

    return filename, recognized_text

def main():
    listen_and_repeat()

if __name__ == "__main__":
    main()
```

Listing 94: Generated Code — GPTo4-mini-high

Test Code

```
# =========================
# test_code.py
# =========================

import os
import shutil
import unittest
import speech_recognition as sr
from unittest.mock import MagicMock
import io
import sys

# We assume main_code.py has been exec'd into this namespace, so
   listen_and_repeat and main are available.

class TestVoiceRepeater(unittest.TestCase):
    TEST_FOLDER = "test_recordings"

    def setUp(self):
        # Create a fake AudioData with dummy WAV bytes
        self.audio_mock = MagicMock()
        self.audio_mock.get_wav_data.return_value = b"
           DUMMY_WAV_BYTES"

        # Patch recognizer methods to use our fake audio and text
        sr.Recognizer.listen = lambda self, source: self.
           audio_mock
        sr.Recognizer.recognize_google = lambda self, audio: "
           hello␣test"

        # Patch playsound in the global namespace
        self.playsound_calls = []
        globals()['playsound'] = lambda path: self.
           playsound_calls.append(path)

        # Ensure clean test folder
        if os.path.exists(self.TEST_FOLDER):
            shutil.rmtree(self.TEST_FOLDER)

    def tearDown(self):
        # Clean up test folder after each test
        if os.path.exists(self.TEST_FOLDER):
            shutil.rmtree(self.TEST_FOLDER)

    def test_listen_and_repeat_creates_file_and_plays_sound(self)
       :
        filename, text = listen_and_repeat(folder=self.
           TEST_FOLDER)

        # Check recognized text
        self.assertEqual(text, "hello␣test")
```

```
        # Folder and file should exist
        self.assertTrue(os.path.isdir(self.TEST_FOLDER))
        self.assertTrue(os.path.isfile(filename))

        # File contents should match our dummy WAV bytes
        with open(filename, "rb") as f:
            data = f.read()
        self.assertEqual(data, b"DUMMY_WAV_BYTES")

        # playsound should have been called once with the correct
            filename
        self.assertEqual(len(self.playsound_calls), 1)
        self.assertEqual(self.playsound_calls[0], filename)

    def test_main_prints_output(self):
        # Patch listen_and_repeat to avoid actual audio work
        globals()['listen_and_repeat'] = lambda folder="x": ("
           fake_file.wav", "test␣output")

        # Capture stdout
        captured = io.StringIO()
        sys_stdout = sys.stdout
        sys.stdout = captured

        try:
            main()
        finally:
            sys.stdout = sys_stdout

        output = captured.getvalue()
        self.assertIn("You␣said:␣test␣output", output)

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 95: Test Code — GPTo4-mini-high

GEMINI 2.5 Response

Generated Code

```
# === MAIN SCRIPT (voice_repeater.py) ===

import speech_recognition as sr
import os
import datetime
import sys
# Note: playsound can sometimes have issues (blocking, platform
   differences).
# Consider sounddevice + soundfile for more robust playback if
   needed.
try:
    import playsound
except ImportError:
    print("Warning:␣'playsound'␣library␣not␣found␣or␣could␣not␣be
       ␣imported.")
    print("Playback␣functionality␣will␣be␣skipped.")
    print("Install␣it␣using:␣pip␣install␣playsound")
    playsound = None
except Exception as e:
    print(f"Warning:␣Error␣importing␣playsound:␣{e}")
    print("Playback␣functionality␣might␣be␣affected.")
    playsound = None


# --- Configuration ---
RECORDINGS_DIR = "voice_recordings"
# Timeout in seconds for listening phase (None means listen until
    silence)
PHRASE_TIMEOUT = None
# Time limit in seconds for listening (stops regardless of
   silence)
PHRASE_TIME_LIMIT = 10 # Stop listening after 10 seconds max

# --- Core Logic ---

def repeat_voice():
    """
    Listens for user speech, saves it to a WAV file, and plays it
        back.
    """
    r = sr.Recognizer()
    # Ensure the recordings directory exists
    try:
        os.makedirs(RECORDINGS_DIR, exist_ok=True)
        print(f"Recordings␣will␣be␣saved␣in:␣{os.path.abspath(
           RECORDINGS_DIR)}")
    except OSError as e:
        print(f"Error␣creating␣directory␣{RECORDINGS_DIR}:␣{e}",
           file=sys.stderr)
```

```
        return # Cannot proceed without directory

    with sr.Microphone() as source:
        print("Adjusting␣for␣ambient␣noise...␣Please␣wait.")
        try:
            # Adjust for ambient noise for better recognition
               quality
            r.adjust_for_ambient_noise(source, duration=1)
        except Exception as e:
            print(f"Warning:␣Could␣not␣adjust␣for␣ambient␣noise:␣
               {e}", file=sys.stderr)
            # Continue anyway, but quality might be lower

        print(f"Say␣something!␣(Listening␣for␣up␣to␣{
           PHRASE_TIME_LIMIT}␣seconds)")

        audio_data = None
        try:
            # Listen for the user's input
            audio_data = r.listen(source, timeout=PHRASE_TIMEOUT,
                phrase_time_limit=PHRASE_TIME_LIMIT)
            print("Got␣it!␣Processing...")
        except sr.WaitTimeoutError:
            print("No␣speech␣detected␣within␣the␣time␣limit.",
               file=sys.stderr)
            return # Nothing to repeat
        except Exception as e:
            print(f"An␣error␣occurred␣during␣listening:␣{e}",
               file=sys.stderr)
            return # Cannot proceed

    if audio_data:
        # Generate a unique filename using timestamp
        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M
           %S")
        filename = os.path.join(RECORDINGS_DIR, f"recording_{
           timestamp}.wav")

        # Save the audio data to a WAV file
        try:
            with open(filename, "wb") as f:
                f.write(audio_data.get_wav_data())
            print(f"Speech␣saved␣to:␣{filename}")

            # Play the saved audio file
            if playsound and hasattr(playsound, 'playsound'):
                try:
                    print("Playing␣back...")
                    # Use blocking=True if your playsound version
                        supports it and you need sequential
                       execution
```

```
                    # playsound.playsound(filename, block=True)
                    playsound.playsound(filename)
                    print("Playback␣complete.")
                except Exception as e:
                    print(f"Error␣playing␣sound:␣{e}", file=sys.
                       stderr)
                    print("You␣might␣need␣to␣install␣platform-
                       specific␣dependencies␣for␣playsound␣(e.g.,␣
                       GStreamer␣on␣Linux,␣AppKit␣on␣macOS).")
            else:
                 print("Skipping␣playback␣because␣'playsound'␣is␣
                    not␣available␣or␣imported␣incorrectly.")

        except IOError as e:
            print(f"Error␣writing␣audio␣file␣{filename}:␣{e}",
               file=sys.stderr)
        except Exception as e:
            # Catch potential errors from get_wav_data() or other
                issues
            print(f"An␣unexpected␣error␣occurred␣during␣
               processing/playback:␣{e}", file=sys.stderr)

# --- Main Execution ---

def main():
    """
    Main function to run the voice repeater.
    """
    print("Starting␣Simple␣Voice␣Repeater...")
    # Check for microphone availability (optional but good
       practice)
    try:
        mics = sr.Microphone.list_microphone_names()
        if not mics:
            print("Error:␣No␣microphones␣found.␣Please␣check␣your
               ␣audio␣hardware␣and␣drivers.", file=sys.stderr)
            return
        # print(f"Available microphones: {mics}") # Optional:
           list mics
        # print(f"Using default microphone.")
    except Exception as e:
         print(f"Warning:␣Could␣not␣list␣microphones:␣{e}.␣
            Attempting␣to␣proceed...", file=sys.stderr)

    repeat_voice()
    print("Voice␣Repeater␣finished.")

if __name__ == "__main__":
    main()

# === END OF MAIN SCRIPT ===
```

Listing 96: Generated Code — GEMINI 2.5

Test Code

```
# === UNIT TESTS (test_voice_repeater.py) ===

import unittest
import os
import datetime
from unittest.mock import patch, MagicMock, mock_open

# --- Mock Objects ---
# Define mock objects for external dependencies used in the main
   script.
# These will replace the actual libraries during testing.

# Mock for speech_recognition's AudioData
class MockAudioData:
    def __init__(self, wav_data=b'mock_wav_data'):
        self._wav_data = wav_data

    def get_wav_data(self):
        return self._wav_data

# Mock for speech_recognition's Recognizer
class MockRecognizer:
    def adjust_for_ambient_noise(self, source, duration=1):
        print(f"Mock:␣Adjusting␣noise␣for␣source:␣{source}␣
           duration:␣{duration}") # Simulate action
        pass # Do nothing in mock

    def listen(self, source, timeout=None, phrase_time_limit=None
       ):
        print(f"Mock:␣Listening␣on␣source:␣{source}␣timeout:␣{
           timeout}␣limit:␣{phrase_time_limit}") # Simulate action
        # Simulate returning valid audio data in the success case
        if self.simulate_listen_error:
            raise self.simulate_listen_error()
        if self.simulate_timeout:
             raise sr.WaitTimeoutError("Mock␣timeout") # Use the
                actual error type if available
        return MockAudioData() # Return mock audio data by
           default

# Mock for speech_recognition's Microphone
class MockMicrophone:
    def __enter__(self):
        print("Mock:␣Entering␣Microphone␣context")
        return self # Return self to mimic 'as source'

    def __exit__(self, exc_type, exc_val, exc_tb):
```

```
        print("Mock:␣Exiting␣Microphone␣context")
        pass # Do nothing special on exit

    def __str__(self):
        return "MockMicrophoneInstance"


# --- Test Suite ---

class TestVoiceRepeater(unittest.TestCase):

    def setUp(self):
        """Set up test environment before each test."""
        # Define a fixed datetime for predictable filenames
        self.test_datetime = datetime.datetime(2025, 4, 22, 17,
           30, 0)
        self.expected_timestamp = self.test_datetime.strftime("%Y
           %m%d_%H%M%S")
        # Ensure the global RECORDINGS_DIR exists for the test
           namespace
        # We assume the main script code has been executed first
           by the evaluator,
        # so RECORDINGS_DIR should be defined. If not, provide a
           default.
        global RECORDINGS_DIR
        try:
             self.recordings_dir = RECORDINGS_DIR
        except NameError:
             print("Warning:␣RECORDINGS_DIR␣not␣found␣in␣
                namespace,␣using␣default␣'test_recordings'")
             self.recordings_dir = "test_recordings"
             RECORDINGS_DIR = self.recordings_dir # Define it for
                 the test scope if missing


    @patch('__main__.sr.Recognizer', return_value=MockRecognizer
       ())
    @patch('__main__.sr.Microphone', return_value=MockMicrophone
       ())
    @patch('__main__.os.makedirs')
    @patch('__main__.datetime.datetime')
    @patch('builtins.open', new_callable=mock_open)
    @patch('__main__.playsound.playsound') # Patch the function
       directly
    def test_successful_repeat(self, mock_playsound,
       mock_open_func, mock_datetime, mock_makedirs, mock_mic_cls,
        mock_rec_cls):
        """Test the normal workflow: listen, save, play."""
        print("\n---␣Running␣test_successful_repeat␣---")
        # Configure mocks
        mock_datetime.now.return_value = self.test_datetime
```

```
        # Get the Recognizer instance created by the patch to
           modify its behavior
        mock_recognizer_instance = mock_rec_cls.return_value
        mock_recognizer_instance.simulate_listen_error = None
        mock_recognizer_instance.simulate_timeout = False

        # Assume repeat_voice is available in the execution
           namespace
        repeat_voice()

        # Assertions
        mock_makedirs.assert_called_once_with(self.recordings_dir
           , exist_ok=True)
        mock_rec_cls.assert_called_once() # Recognizer
           initialized
        mock_mic_cls.assert_called_once() # Microphone used
        mock_recognizer_instance.adjust_for_ambient_noise.
           assert_called_once()
        mock_recognizer_instance.listen.assert_called_once()

        expected_filename = os.path.join(self.recordings_dir, f"
           recording_{self.expected_timestamp}.wav")
        mock_open_func.assert_called_once_with(expected_filename,
            "wb")
        mock_open_func().write.assert_called_once_with(b'
           mock_wav_data')
        mock_playsound.assert_called_once_with(expected_filename)
        print("---␣test_successful_repeat␣finished␣---")


    @patch('__main__.sr.Recognizer', return_value=MockRecognizer
       ())
    @patch('__main__.sr.Microphone', return_value=MockMicrophone
       ())
    @patch('__main__.os.makedirs')
    @patch('builtins.open', new_callable=mock_open)
    @patch('__main__.playsound.playsound')
    def test_listen_timeout(self, mock_playsound, mock_open_func,
        mock_makedirs, mock_mic_cls, mock_rec_cls):
        """Test the case where listening times out (no speech).
           """
        print("\n---␣Running␣test_listen_timeout␣---")
        # Configure mocks
        # Import the actual exception type if needed for the test
            environment
        try:
            from speech_recognition import WaitTimeoutError
        except ImportError:
            # Define a dummy exception if SpeechRecognition isn't
                installed in the test env
            class WaitTimeoutError(Exception): pass
```

```
            # Add it to the global scope for the mocked class to
               use
            global sr # Assume sr was the import alias in main
               script
            if 'sr' not in globals(): sr = MagicMock() # Mock the
                module if not present
            sr.WaitTimeoutError = WaitTimeoutError # Assign the
               dummy class

        mock_recognizer_instance = mock_rec_cls.return_value
        mock_recognizer_instance.simulate_listen_error = None
        mock_recognizer_instance.simulate_timeout = True #
           Simulate timeout

        # Assume repeat_voice is available in the execution
           namespace
        repeat_voice()

        # Assertions
        mock_makedirs.assert_called_once_with(self.recordings_dir
           , exist_ok=True)
        mock_recognizer_instance.listen.assert_called_once()
        mock_open_func.assert_not_called() # File should not be
           opened
        mock_playsound.assert_not_called() # Playback should not
           happen
        print("---␣test_listen_timeout␣finished␣---")


    @patch('__main__.sr.Recognizer', return_value=MockRecognizer
       ())
    @patch('__main__.sr.Microphone', return_value=MockMicrophone
       ())
    @patch('__main__.os.makedirs', side_effect=OSError("Mock␣
       permission␣denied"))
    @patch('builtins.open', new_callable=mock_open)
    @patch('__main__.playsound.playsound')
    def test_directory_creation_error(self, mock_playsound,
       mock_open_func, mock_makedirs, mock_mic_cls, mock_rec_cls):
        """Test the case where directory creation fails."""
        print("\n---␣Running␣test_directory_creation_error␣---")
        # Configure mocks (makedirs already configured to raise
           OSError)

        # Assume repeat_voice is available in the execution
           namespace
        repeat_voice()

        # Assertions
        mock_makedirs.assert_called_once_with(self.recordings_dir
           , exist_ok=True)
```

```
        # Execution should stop after makedirs fails
        mock_mic_cls.assert_not_called()
        mock_rec_cls.assert_not_called() # Recognizer is created
           *before* makedirs in the original code
        mock_open_func.assert_not_called()
        mock_playsound.assert_not_called()
        print("---␣test_directory_creation_error␣finished␣---")


    @patch('__main__.sr.Recognizer', return_value=MockRecognizer
       ())
    @patch('__main__.sr.Microphone', return_value=MockMicrophone
       ())
    @patch('__main__.os.makedirs')
    @patch('__main__.datetime.datetime')
    @patch('builtins.open', new_callable=mock_open, side_effect=
       IOError("Mock␣write␣error"))
    @patch('__main__.playsound.playsound')
    def test_file_write_error(self, mock_playsound,
       mock_open_func, mock_datetime, mock_makedirs, mock_mic_cls,
        mock_rec_cls):
        """Test the case where writing the WAV file fails."""
        print("\n---␣Running␣test_file_write_error␣---")
        # Configure mocks
        mock_datetime.now.return_value = self.test_datetime
        mock_recognizer_instance = mock_rec_cls.return_value
        mock_recognizer_instance.simulate_listen_error = None
        mock_recognizer_instance.simulate_timeout = False

        # Assume repeat_voice is available in the execution
           namespace
        repeat_voice()

        # Assertions
        mock_makedirs.assert_called_once_with(self.recordings_dir
           , exist_ok=True)
        mock_recognizer_instance.listen.assert_called_once()
        expected_filename = os.path.join(self.recordings_dir, f"
           recording_{self.expected_timestamp}.wav")
        mock_open_func.assert_called_once_with(expected_filename,
            "wb")
        # write() would be called inside the 'with open...'
           context, but open itself raises error here
        mock_playsound.assert_not_called() # Playback should not
           happen if writing fails
        print("---␣test_file_write_error␣finished␣---")

    @patch('__main__.sr.Recognizer', return_value=MockRecognizer
       ())
    @patch('__main__.sr.Microphone', return_value=MockMicrophone
       ())
```

```
    @patch('__main__.os.makedirs')
    @patch('__main__.datetime.datetime')
    @patch('builtins.open', new_callable=mock_open)
    @patch('__main__.playsound.playsound', side_effect=Exception(
       "Mock␣playback␣device␣error")) # Simulate playback failure
    def test_playback_error(self, mock_playsound, mock_open_func,
        mock_datetime, mock_makedirs, mock_mic_cls, mock_rec_cls):
        """Test the case where playsound fails."""
        print("\n---␣Running␣test_playback_error␣---")
        # Configure mocks
        mock_datetime.now.return_value = self.test_datetime
        mock_recognizer_instance = mock_rec_cls.return_value
        mock_recognizer_instance.simulate_listen_error = None
        mock_recognizer_instance.simulate_timeout = False

        # Assume repeat_voice is available in the execution
           namespace
        repeat_voice()

        # Assertions
        mock_makedirs.assert_called_once_with(self.recordings_dir
           , exist_ok=True)
        mock_recognizer_instance.listen.assert_called_once()
        expected_filename = os.path.join(self.recordings_dir, f"
           recording_{self.expected_timestamp}.wav")
        mock_open_func.assert_called_once_with(expected_filename,
            "wb")
        mock_open_func().write.assert_called_once_with(b'
           mock_wav_data')
        # Check that playsound was called, even though it raised
           an error
        mock_playsound.assert_called_once_with(expected_filename)
        print("---␣test_playback_error␣finished␣---")

# --- Test Runner ---

def main():
    """
    Main function to run the unit tests.
    """
    print("Starting␣Unit␣Tests␣for␣Voice␣Repeater...")
    # Need to mock the sr module globally if it's not installed
       in the test env
    # or if the main script wasn't executed first in the same
       namespace
    global sr
    if 'sr' not in globals():
        print("Mocking␣'sr'␣module␣globally␣for␣tests.")
        sr = MagicMock()
        # Define necessary attributes/classes on the mock module
           if needed by tests
```

```
        sr.Recognizer = MockRecognizer
        sr.Microphone = MockMicrophone
        # Add the exception type if it was defined locally
        try:
            from speech_recognition import WaitTimeoutError
            sr.WaitTimeoutError = WaitTimeoutError
        except ImportError:
            class WaitTimeoutError(Exception): pass
            sr.WaitTimeoutError = WaitTimeoutError

    # Mock playsound if not installed/imported correctly
    global playsound
    if ’playsound’ not in globals() or playsound is None:
         print("Mocking␣’playsound’␣module␣globally␣for␣tests.")
         # Create a mock object that mimics the playsound module
            structure
         playsound = MagicMock()
         # Define the playsound function on the mock module
         playsound.playsound = MagicMock()

    # Run the tests
    suite = unittest.TestSuite()
    suite.addTest(unittest.makeSuite(TestVoiceRepeater))
    runner = unittest.TextTestRunner()
    runner.run(suite)
    print("Unit␣Tests␣Finished.")

if __name__ == "__main__":
    # This ensures the tests run when the script is executed
       directly.
    # The evaluator will likely use exec() on the code string,
    # but this structure follows the requirement.
    main()


# === END OF UNIT TESTS ===
```

Listing 97: Test Code — GEMINI 2.5

GROK 3 Response

Generated Code

```
import os
import speech_recognition as sr
from playsound import playsound
import wave
import pyaudio
import time
from datetime import datetime

class VoiceRepeater:
    def __init__(self, output_dir="recordings"):
        self.output_dir = output_dir
        self.recognizer = sr.Recognizer()
        self.chunk = 1024
        self.sample_format = pyaudio.paInt16
        self.channels = 1
        self.fs = 44100
        self.max_duration = 10  # Maximum recording duration in
            seconds

        # Create output directory if it doesn't exist
        if not os.path.exists(self.output_dir):
            os.makedirs(self.output_dir)

    def record_audio(self):
        """Record audio from microphone and save as WAV file"""
        p = pyaudio.PyAudio()
        stream = p.open(format=self.sample_format,
                        channels=self.channels,
                        rate=self.fs,
                        frames_per_buffer=self.chunk,
                        input=True)

        frames = []
        print("Recording...␣Speak␣now!")

        # Record for max_duration seconds
        start_time = time.time()
        while time.time() - start_time < self.max_duration:
            data = stream.read(self.chunk, exception_on_overflow=
                False)
            frames.append(data)

        print("Recording␣finished.")

        # Stop and close the stream
        stream.stop_stream()
        stream.close()
        p.terminate()
```

```
        # Save the recorded audio
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        filename = os.path.join(self.output_dir, f"recording_{
           timestamp}.wav")

        wf = wave.open(filename, ’wb’)
        wf.setnchannels(self.channels)
        wf.setsampwidth(p.get_sample_size(self.sample_format))
        wf.setframerate(self.fs)
        wf.writeframes(b’’.join(frames))
        wf.close()

        return filename

    def play_audio(self, filename):
        """Play the specified audio file"""
        try:
            playsound(filename)
            return True
        except Exception as e:
            print(f"Error␣playing␣audio:␣{e}")
            return False

    def recognize_speech(self, audio_file):
        """Recognize speech from audio file"""
        try:
            with sr.AudioFile(audio_file) as source:
                audio = self.recognizer.record(source)
                text = self.recognizer.recognize_google(audio)
                return text
        except sr.UnknownValueError:
            return "Could␣not␣understand␣audio"
        except sr.RequestError as e:
            return f"Speech␣recognition␣error:␣{e}"
        except Exception as e:
            return f"Error␣in␣speech␣recognition:␣{e}"

    def run(self):
        """Main method to run the voice repeater"""
        try:
            # Record audio
            audio_file = self.record_audio()

            # Recognize speech
            recognized_text = self.recognize_speech(audio_file)
            print(f"You␣said:␣{recognized_text}")

            # Play back the recording
            print("Playing␣back␣your␣recording...")
            self.play_audio(audio_file)
```

```
            return recognized_text, audio_file
        except Exception as e:
            print(f"Error␣in␣voice␣repeater:␣{e}")
            return None, None

def main():
    repeater = VoiceRepeater()
    repeater.run()

if __name__ == "__main__":
    main()
```

Listing 98: Generated Code — GROK 3

Test Code

```
import unittest
import tempfile
import shutil

class TestVoiceRepeater(unittest.TestCase):
    def setUp(self):
        # Create temporary directory for recordings
        self.temp_dir = tempfile.mkdtemp()
        self.repeater = VoiceRepeater(output_dir=self.temp_dir)

    def tearDown(self):
        # Clean up temporary directory
        shutil.rmtree(self.temp_dir)

    def test_output_directory_creation(self):
        """Test if output directory is created"""
        self.assertTrue(os.path.exists(self.temp_dir))

    def test_record_audio(self):
        """Test if recording creates a valid WAV file"""
        audio_file = self.repeater.record_audio()
        self.assertTrue(os.path.exists(audio_file))
        self.assertTrue(audio_file.endswith('.wav'))

        # Check if file is a valid WAV file
        try:
            with wave.open(audio_file, 'rb') as wf:
                self.assertEqual(wf.getnchannels(), self.repeater
                   .channels)
                self.assertEqual(wf.getframerate(), self.repeater
                   .fs)
        except wave.Error:
            self.fail("Invalid␣WAV␣file␣format")

    def test_play_audio_invalid_file(self):
        """Test playing non-existent audio file"""
```

```
        result = self.repeater.play_audio("non_existent.wav")
        self.assertFalse(result)

    def test_recognize_speech_invalid_file(self):
        """Test speech recognition with invalid file"""
        result = self.repeater.recognize_speech("non_existent.wav
            ")
        self.assertTrue(isinstance(result, str))
        self.assertIn("Error", result)

    def test_run_complete_flow(self):
        """Test the complete run flow"""
        recognized_text, audio_file = self.repeater.run()
        self.assertIsNotNone(audio_file)
        self.assertTrue(os.path.exists(audio_file))

def main():
    unittest.main()

if __name__ == "__main__":
    main()
```

Listing 99: Test Code — GROK 3