

Evaluating Software Process Models for Multi-Agent Class-Level Code Generation

Wasique Islam Shafin
SPEAR Lab, Concordia University
Montreal, QC, Canada
w_shafin@encs.concordia.ca

Zhenhao Li
York University
Toronto, Ontario, Canada
lzhenhao@yorku.ca

Md Nakhla Rafi
SPEAR Lab, Concordia University
Montreal, QC, Canada
r_mdnakh@encs.concordia.ca

Tse-Hsun (Peter) Chen
SPEAR Lab, Concordia University
Montreal, QC, Canada
peterc@encs.concordia.ca

Abstract

Modern software systems require code that is not only functional but also maintainable and well-structured. Although Large Language Models (LLMs) are increasingly used to automate software development, most studies focus on isolated, single-agent function-level generation. This work examines how process structure and role specialization shape multi-agent LLM workflows for class-level code generation. We simulate a Waterfall-style development cycle covering Requirement, Design, Implementation, and Testing using three LLMs (GPT-4o-mini, DeepSeek-Chat, and Claude-3.5-Haiku) on 100 Python tasks from the ClassEval benchmark. Our findings show that multi-agent workflows reorganize, rather than consistently enhance, model performance. Waterfall-style collaboration produces cleaner and more maintainable code but often reduces functional correctness (−37.8% for GPT-4o-mini and −39.8% for DeepSeek-Chat), with Claude-3.5-Haiku as a notable exception (+9.5%). Importantly, process constraints shift failure characteristics: structural issues such as missing code decrease, while semantic and validation errors become more frequent. Among all stages, Testing exerts the strongest influence by improving verification coverage but also introducing new reasoning failures, whereas Requirement and Design have comparatively modest effects. Overall, this study provides empirical evidence that software process structure fundamentally alters how LLMs reason, collaborate, and fail, revealing inherent trade-offs between rigid workflow discipline and flexible problem-solving in multi-agent code generation.

Keywords

large language model, code generation, agents

ACM Reference Format:

Wasique Islam Shafin, Md Nakhla Rafi, Zhenhao Li, and Tse-Hsun (Peter) Chen. 2018. Evaluating Software Process Models for Multi-Agent Class-Level Code Generation. In *Proceedings of Make sure to enter the correct conference*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06

<https://doi.org/XXXXXXX.XXXXXXX>

title from your rights confirmation email (Conference acronym 'XX). ACM, New York, NY, USA, 12 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Large Language Models (LLMs) have become increasingly influential in Software Engineering (SE), assisting in diverse tasks such as code generation [11, 21, 29, 30, 34, 37], code completion and assistance [1, 14, 19, 40], fault localization and testing [24, 38, 41], and automated program repair [6, 7, 44, 46, 47, 49]. General-purpose models such as ChatGPT [36], Claude [4], and DeepSeek [15] have further extended the reach of LLMs into broader SE workflows, including debugging, documentation, and architecture planning [23, 50, 51]. As software systems continue to grow in complexity, the ability of LLMs to generate maintainable, reliable, and secure code has become central to their practical deployment in SE [2, 16].

Early advances in automated code generation primarily focused on function-level synthesis, where models such as Codex [13], AlphaCode [28], and CodeT5+ [42] demonstrated strong capabilities in producing short, self-contained functions. This line of research evolved through increasingly complex benchmarks like HumanEval [13], MBPP [5], and BigCodeBench [52], which standardize evaluations using metrics such as Pass@k. More recent frameworks, like Chain-of-Programming [22], interactive test-driven generation [18], and reasoning-enhanced prompting [27, 43], extend these capabilities toward more realistic and process-aware programming tasks. However, while these methods improve functional correctness, most evaluations still consider isolated functions rather than interconnected classes that reflect real-world software modularity and maintainability concerns.

To address this limitation, class-level benchmarks like *ClassEval* [17] have emerged to capture object-oriented code generation, assessing how well models handle multi-method dependencies and shared class state. Subsequent extensions such as Xue et al. [45] and Chen et al. [12] expand *ClassEval* in multiple languages and integrate error analysis frameworks, offering deeper insight into reasoning-based code generation. However, despite these advances, the majority of research remains focused on single-LLM paradigms. At the same time, multi-agent frameworks such as ChatDev [37], MetaGPT [21], SOEN-101 [29], and AgileCoder [35] have introduced collaborative, role-based approaches that emulate human software processes by assigning distinct agents to roles such as Product Manager, Architect, Developer, and Tester. These systems

have shown improvements in modularity, reasoning, and error mitigation at the function level. However, evaluations have not yet extended to class-level tasks, leaving an open question: *Do multi-agent LLM workflows enhance class-level code quality and correctness, and which development activities matter most?*

In this paper, we bridge these two research directions by studying how a multi-agent workflow can operate within a classical Waterfall software process model for class-level code generation using *ClassEval*. We use Waterfall because its sequential structure (Requirement → Design → Implementation → Testing) makes it easier to study the effect of each development activity. Our study assigns distinct roles (Requirement Engineer, Architect, Developer, and Tester) to specialized LLM agents, allowing us to investigate how each stage of the Waterfall process contributes to the final implementation. Through an ablation-based experimental design, we evaluate three LLM backends (GPT-4o-mini, DeepSeek-Chat, Claude-3.5-Haiku) and compare them against direct prompting as the baseline, measuring (1) functional correctness (pass@1), (2) software quality and maintainability using SonarQube metrics, (3) identify frequently encountered error types, and (4) detailed failure categories through a taxonomy-driven error analysis.

Our results show that multi-agent LLM workflows do not uniformly enhance class-level code correctness but significantly affect code quality and error characteristics. Waterfall-style workflows give cleaner, more maintainable code; however, pass@1 accuracy declines for gpt-4o-mini (-37.8%) and deepseek-chat (-39.8%), while claude-3.5-haiku improves (+9.5%), indicating model-dependent effects. These workflows also increase runtime errors by 10-53%, with *ValueError*, *AssertionError*, and *TypeError* occurring most frequently. Although structural issues such as *Missing Code* are reduced, *Semantic Failure* and *Return Mismatches* become more common. Among development activities, Testing has the strongest influence: including it improves verification but increases reasoning and validation errors; excluding it yields the poorest correctness yet highest code quality. In contrast, the Requirements and Design stages have minimal impact on correctness or error types. The complete Waterfall configuration produces the lowest correctness and greatest adaptability concerns, suggesting that over-structuring the workflow constrains the flexibility of the final code. Incorporating these development activities into a multi-agent workflow enforces stricter validation and verification, but this also makes the code more error-prone and reduces its overall adaptability, intentionality, and correctness.

Overall, this paper makes the following contributions:

- We propose the first systematic study of multi-agent LLM workflows applied to class-level code generation using *ClassEval* in the form of a Waterfall process model.
- We perform an empirical comparison of Waterfall-activity ablations that isolates the effect of every development activity on functional correctness and code quality.
- We found that multi-agent LLM workflows do not consistently improve class-level code correctness but do significantly influence code quality; testing has the greatest impact, strengthening verification while also increasing reasoning errors, whereas overly structured Waterfall workflows reduce both adaptability and correctness.

- We make all code and data from our experiments publicly available to enable reproduction, verification, and further research. [3]

2 Background and Related Work

LLMs have rapidly advanced the field of automated code generation, with growing research spanning model development, benchmarking, and multi-agent collaboration frameworks. This section first reviews the evolution of LLM-based code generation, then discusses recent progress in agent-driven workflows that simulate software engineering processes. We next summarize existing benchmarks and surveys, particularly those addressing class-level generation tasks. Finally, we identify the research gap motivating our study, which evaluates class-level code generation within structured multi-agent workflows, a dimension that remains missing in current literature.

2.1 LLM in Code Generation

The evolution of LLMs for code generation has developed rapidly since 2021, beginning with Codex [13], which demonstrated that models fine-tuned on large-scale code bases can translate natural language into executable code. Subsequent models, such as AlphaCode [28] and CodeT5+ [42], expanded further by handling competitive programming challenges and refining generation through encoder-decoder architectures and feedback-based ranking mechanisms. By 2023–2024, works like PANGU-CODER2 [39] and query-aligned prompting methods [27, 32, 43] improved model reasoning and task alignment. More recent research has shifted toward embedding LLMs within full software workflows, incorporating test-driven development and interactive debugging [18, 22]. Surveys published in 2025 [2, 16] highlight a transition from code snippet generation to end-to-end software development workflows, encompassing planning, optimization, and integration.

2.2 Multi-Agent Code Generation Frameworks

In parallel, multi-agent frameworks have emerged to emulate collaborative software engineering processes using role-specialized LLMs. Frameworks like ChatDev [37] and MetaGPT [21] assign agents roles such as Product Manager, Developer, and Tester, coordinating through dialogue (chatting) to design, implement, and test software solutions. These systems primarily operate at the function level, evaluating outcomes via completeness, executability, and pass@k metrics. SOEN-101 [29] extends this approach by structuring multi-agent workflows according to software process models (e.g., Waterfall, TDD, Scrum), while MAS [10] categorizes common failure types across frameworks, such as system design errors and inter-agent misalignments. AgileCoder [35] further introduces Agile-inspired iterative sprints, highlighting adaptability and incremental refinement. Collectively, these studies confirm that agent-based LLMs improve task division and communication in software generation, but they remain confined to function-level code, leaving class-level software development largely unexplored.

2.3 Benchmarks and Surveys

Benchmarking remains central to evaluating progress in LLM-based code generation. Widely adopted datasets such as HumanEval [13],

MBPP [5], and BigCodeBench [52] measure correctness using the pass@k metric, focusing on isolated, function-level tasks. Surveys by Jiang et al. [23], Zhang et al. [50], and Zheng et al. [51] systematically compare LLM performance across these benchmarks, revealing ongoing challenges in reasoning, building complex code, and error propagation.

Recognizing the limitations of function-level assessments, ClassEval [17] was introduced to evaluate class-level generation tasks involving interdependent methods, shared variables, and internal class logic. Follow-up works such as Xue et al. [45] expanded ClassEval to Java and C++, incorporating new metrics and error analysis, while Chen et al. [12] examined reasoning performance across HumanEval and ClassEval. Complementary frameworks like CoCoST [20] integrate automated test generation and web search to assess broader problem-solving capabilities. These advances collectively provide a foundation for studying class-level code generation, yet none explore how structured, agent-driven workflows like waterfall influence performance on such complex tasks.

2.4 Limitations and Research Gap

While multi-agent frameworks have proven effective for function-level collaboration and class-level benchmarks have captured higher program complexity, these two directions remain largely disconnected. Existing literature does not examine how structured agent workflows affect class-level code generation quality, nor how different software development activities (requirements analysis, design, implementation, and testing) contribute to correctness and maintainability. Our study bridges this gap by evaluating class-level code generation through a Waterfall-inspired multi-agent workflow. This structured setup enables activity-wise analysis of LLM behaviour by tracing how each development activity impacts the final code quality across models. In contrast to prior work emphasizing model architecture or function-level benchmarks, our contribution lies in characterizing how collaborative process design has an impact on what LLMs produce, offering new insights into class-level code generation that is aware of the software engineering process.

3 Methodology and Experiment Setup

To understand how different development activities influence the quality of code produced during *class-level* code generation, we use agents to emulate the **Waterfall** model. Its straightforward and sequential nature allows us to examine how individual development activities affect the resulting code. We follow a step-by-step linear process, where development flows downwards through distinct phases (from requirements to design, implementation, and testing). Figure 1 provides an overview of our study design. Our experiment design is organized around two main components: (1) **Agent Roles**, which define the LLM agents and their responsibilities, and (2) **Agent Communication**, which describes how these agents interact throughout the development phases under the Waterfall model.

3.1 Agent Roles

We employ four LLM agents to represent the primary activities of the software development life cycle (SDLC) within a *Waterfall* process: *requirements*, *design*, *implementation*, and *testing*. These agents

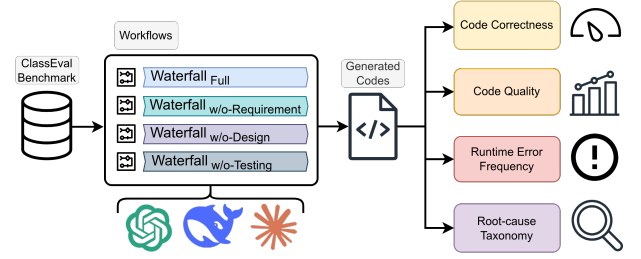


Figure 1: Approach Overview

are (1) **Requirement Engineer**, (2) **Architect**, (3) **Developer**, and (4) **Tester**, each focusing on a specific development activity and their interactions, mirror their real-life counterparts.

Every agent uses the same unified prompt template, with differences in role-specific details. We adapt and modify the prompt structure from SOEN-101 [29] to enable class-level generation workflows, whereas it focused only on function-level generation only. The prompt template defines how each agent interprets its assigned role, consumes relevant context, and generates the corresponding software artifact. A simplified example of the template is shown below:

```
{
  "Role": "You are a [role] delegated for [task]",
  "Instruction": "According to the Context,
                [role-specific-instruction]",
  "Example": "[document example]",
  "Context": "[context]",
  "Question": "Follow the instructions.
              The [document] must satisfy the requirements."
}
```

role defines the agent's role in the process model (e.g., Developer or Tester), *task* specifies the objective of that role, *document* represents the artifact to be produced, *example* provides a reference format, *role-specific-instruction* offers detailed guidance tailored to the task, and *context* supplies necessary inputs or artifacts from earlier stages.

Table 1 summarizes the *tasks*, *instructions*, and *contexts* assigned to each agent, showing how they collaborate to generate, refine, and verify software artifacts. By delegating well-defined tasks to specialized agents, our approach enables systematic experimentation within a Waterfall workflow, refining how agents collaborate and exchange information across sequential phases.

3.2 Agent Communication

For our study, the agent communication is based on the Waterfall process model. This process divides work into a sequence of well-defined activities (e.g., requirements, design, implementation, and testing) where each activity depends on the deliverables of the previous one. Within this process, the agents with specialized roles collaborate across the sequential stages with formal handoffs and documentations to produce a final product. In addition, we designed the agents to provide feedback for self-refinement [33].

This section describes how agents coordinate within these structured activities and how their interactions are organized into a complete Waterfall-inspired workflow.

Table 1: Tasks, instructions, and context associated with each role prompt

Role	Task	Instructions	Context
Requirement Engineer	Analyze the task description and write a requirements document.	1) Review the task description. 2) Write a requirements document that accounts for method signatures and class variables.	Task description (user requirement).
Architect	Design the high-level components and structure of the code.	1) Review the provided documents. 2) Write a design document that preserves method definitions and serves as a guide for developers.	Requirements document.
Developer	Implement Python code that meets all requirements.	1) Review the provided documents. 2) Write clean, efficient, and readable code following best practices. Use a single class structure and keep method definitions unchanged.	Design document.
	Fix code while ensuring all requirements are met.	1) Review the test reports and provided documents. 2) Revise the code to make it efficient, readable, and consistent with best practices.	Test report and original code.
Tester	Design test cases that verify all requirements.	1) Review the provided documents. 2) Write test cases that preserve method definitions, cover normal, edge, and error conditions, and include at least five test cases per method.	Task description (user requirement).
	Write a Python test script using the unittest framework.	1) Review the provided documents. 2) Write test scripts with one test case for each scenario, reusing existing modules where possible.	Test case document.
	Write a test failure report.	1) Review the test execution results. 2) Document the outcomes in a test report.	Test execution results.

3.2.1 Communication. The agent communication process is divided into four sequential activities: *Requirement*, *Design*, *Implementation*, and *Testing*, corresponding directly to the core stages of the *Waterfall* software development life cycle (SDLC). As illustrated in Figure 2, these four activities collectively form a complete Waterfall-inspired communication process among the LLM agents. **Requirement Activity.** The *Requirement Engineer* agent begins the process by transforming the User Requirement into a formal Requirement Document that captures the high-level functional and structural details, such as method signatures, class variables, and inter-dependencies. Communication in this phase primarily involves clarification and validation of user needs by the agent. The document is self-refined [33] through feedback from the *Architect* and *Tester* to ensure fulfillment of acceptance criteria before progressing to the next phase.

Design Activity. The *Architect* agent receives the completed Requirement Document and produces a Design Document that outlines the system’s architecture, class structure, and method interactions. This phase formalizes the blueprint for development, and feedback from the *Developer* and *Tester* is incorporated [33] only to resolve inconsistencies or design ambiguities. This phase reflects Waterfall’s controlled, document-driven communication before implementation begins.

Implementation Activity. The *Developer* agent implements the Design Document by producing a Code Document that the agent realizes the required functionality in executable code using a single class structure. Like that of previous phases, our framework allows self-refinement [33] based on suggestions from the *Architect* and *Tester* to correct implementation deviations or improve maintainability. This controlled interaction simulates real-world review cycles that occur before code is formally handed over for testing.

Testing Activity. The *Tester* agent derives Test Cases directly from the User Requirement and translates them into executable Test Scripts. These test cases are self-refined [33] using feedback from the *Architect* and *Developer* to resolve ambiguity and ensure accurate test coverage. The *Tester* executes the Test Script against the Code Document, producing a Test Report that summarizes the

detected faults and performance outcomes. Finally, the *Developer* may perform up to three bug-fix cycles based on this report, after which the process concludes. Thus, reflecting the validation and closure phase of the Waterfall cycle.

3.2.2 Studying the Effect of Each Development Activity. To understand the impact of each activity, we take the complete Waterfall and remove one activity at a time, and then observe how its absence affects overall performance, code quality, and the possible issues. This ablation-based approach allows us to identify which phases most strongly affect the correctness, completeness, and reliability of the final output. We retain the Implementation phase across all the configurations, as code generation is crucial to producing the executable artifact required for evaluation. For our study, we have 4 configurations of Waterfall, where *w/o* denotes the removal of the corresponding activity: 1) **Waterfall_{Full}**, 2) **Waterfall_{w/oRequirement}**, 3) **Waterfall_{w/oDesign}**, 4) **Waterfall_{w/oTesting}**.

3.3 Dataset

We use the **ClassEval** dataset [17], a human-curated benchmark for evaluating LLMs on *class-level Python coding tasks*. Unlike prior benchmarks such as **HumanEval** [13] and **MBPP** [5], which assess short, function-level code generation, **ClassEval** emphasizes the generation of complete classes that integrate multiple methods, fields, and dependencies. This class structure introduces more reasoning challenges related to design consistency, state management, and inter-method interaction, i.e., features that closely resemble real-world software development than that of isolated coding exercises. The dataset contains 100 programming assignments, each requiring the implementation of a complete Python class. It was developed over approximately 500 person-hours, emphasizing diversity and realism in software development scenarios.

Each ClassEval task provides: (1) *Class Skeleton*. This includes import statements, the class name, a brief description, and the class constructor, along with detailed specifications for each method. Each method specification outlines the method signature, functional description, parameters, and return values, and example

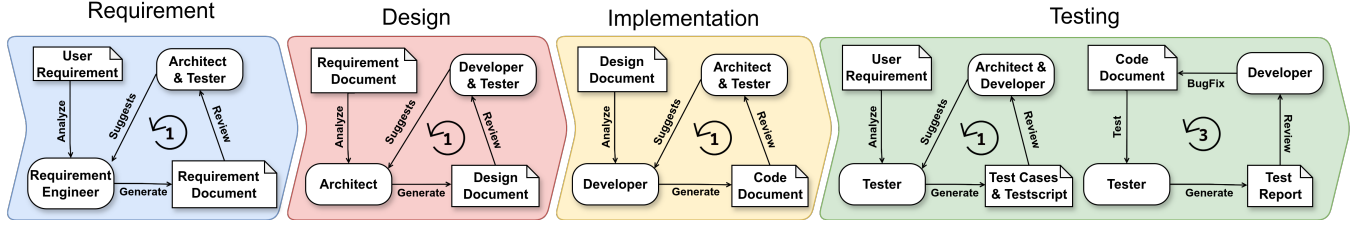


Figure 2: Activities of Waterfall Workflow

input/output. These skeletons serve as the task descriptions in our study. (2) *Ground-Truth Implementation*. A complete and correct reference implementation of the target class in Python, serving as the canonical solution. (3) *Test Cases*. An average of 33.1 test cases per class to rigorously assess code correctness. On average, each class comes with 33.1 test cases designed to rigorously evaluate correctness. These tests allow us to measure both function-level and class-level accuracy.

We want to study the impact of each activity on different quality aspects at class-level. For our experiments, we adopt the holistic class-generation approach, which generates an entire class at once. This strategy has been shown to achieve better performance [17, 45], particularly with the improved context handling capabilities of modern LLMs.

3.4 LLM Choice and Implementation Details

We evaluate our workflows using three contemporary LLMs: GPT-4o Mini (gpt-4o-mini-2024-07-18), deep seek-chat (DeepSeek-V3-0324), and claude-3.5-haiku (claude-3-5-haiku-20241022). All three models represent lightweight yet high-performance variants of their respective families, making them suitable for our multi-agent and workflow-based code generation study. By applying identical workflows across models, we hope to observe whether different LLMs show distinct sensitivities to different development activities in the Waterfall process model, and to determine which phases of the Waterfall pipeline influence their performance. For all models, the temperature is set to 0.8 to balance creativity and determinism during generation.

4 Results

In this section, we present the results to our research questions.

4.1 RQ1: What is the accuracy of the generated code?

Motivation: Class-level code generation differs from method-level generation, as it requires handling higher-level design decisions, inter-method interactions, and overall class organization rather than isolated function logic. In this RQ, we study how applying the *Waterfall* process model and enforcing/implementing each of its development activities affect the quality of generated code, and what trade-offs arise in terms of code correctness or quality compared to unstructured direct prompting.

Approach: For our class-level study on LLM agent-generated code, we compare the outputs of Waterfall-inspired variants with a baseline *RawPrompt* workflow, where the model generates code

directly from task descriptions without any intermediate development stages. We evaluate the effect of development activities by studying the code quality generated when each activity is removed one at a time. We evaluate the generated code based on two aspects: (1) **code correctness**, measured using the *Pass@1* metric from the ClassEval benchmark, and (2) **code quality**, analyzed using static analysis with SonarQube. This combined evaluation allows us to assess both the execution success of generated code and its overall code quality.

Code Correctness. We use *Pass@k* [13, 25] as the primary measure of code correctness, representing the expected proportion of problems correctly solved given k generated code samples:

$$\text{Pass@}k = \mathbb{E}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (1)$$

Here, n denotes the total number of generated samples, c represents the number of correct solutions, and k is the number of evaluated samples. We report both **class-level** and **function-level** *Pass@1* results. At the class level, correctness requires all associated methods in a class to pass their respective tests, whereas at the function level, each method is evaluated independently. Focusing on *Pass@1* provides a strict single-attempt measure of correctness, capturing how often the first generated code sample passes all tests. All evaluations are conducted using the ClassEval framework.

Code Quality. To complement test-based correctness, we analyze code quality using **SonarQube**¹, which detects and categorizes issues into two major groups: *Software Quality* and *Clean Code Attributes*. The *Software Quality*² group includes (1) **Security** - Safeguarding software against unauthorized access, use, or damage (protection from misuse), (2) **Reliability** - Ensuring software consistently performs as expected over time (consistent performance), and (3) **Maintainability** - Ease of understanding, fixing, and improving software code (ease of modification). The *Clean Code*³ group evaluates (1) **Consistency** - Writing code in a uniform and conventional manner (adherence to conventions), (2) **Intentionality** - Ensuring code is precise and purposeful (clarity and completeness), (3) **Adaptability** - Structuring code for easy evolution and confident development (modular and reusable structure), and (4) **Responsibility** - Considering ethical obligations and societal impact in code (lawful and ethical coding behavior).

¹<https://www.sonarsource.com/products/sonarqube/>

²<https://docs.sonarsource.com/sonarqube-server/10.6/user-guide/clean-code/software-quality>

³<https://docs.sonarsource.com/sonarqube-server/10.6/user-guide/clean-code/definition>

Table 2: Comparison of *gpt-4o-mini*, *deepseek-chat*, and *claude-3.5-haiku* across Waterfall variants. Each cell shows the raw metric and its relative change (%) from the *RawPrompt* baseline. Green indicates improvement; red indicates regression. Higher is better for *Pass@1*, lower for both *Software Quality* and *Clean Code Attributes*.

Model	Approach	Pass@1 (↑)		Software Quality (↓)		Clean Code Attribute (↓)		
		Class	Function	Reliability	Maintainability	Consistency	Intentionality	Adaptability
gpt-4o-mini	RawPrompt	0.35	0.6853	0	0.2254	0.1211	0.0908	0.0135
	Waterfall _{Full}	0.21 (-40%)	0.5478 (-20%)	0.0044	0.2188 (-3%)	0.0774 (-36%)	0.0796 (-12%)	0.0663 (+391%)
	Waterfall _{w/o-Requirement}	0.23 (-34%)	0.5458 (-20%)	0.0023	0.1991 (-12%)	0.0801 (-34%)	0.0618 (-32%)	0.0595 (+341%)
	Waterfall _{w/o-Design}	0.21 (-40%)	0.5896 (-14%)	0.0072	0.2141 (-5%)	0.0818 (-33%)	0.0987 (+9%)	0.0409 (+203%)
	Waterfall _{w/o-Testing}	0.22 (-37%)	0.5458 (-20%)	0	0.1820 (-19%)	0.0688 (-43%)	0.0738 (-19%)	0.0393 (+191%)
deepseek-chat	RawPrompt	0.46	0.7529	0	0.2015	0.1085	0.0558	0.0372
	Waterfall _{Full}	0.32 (-30%)	0.6255 (-17%)	0.0059	0.2075 (+3%)	0.0889 (-18%)	0.0692 (+24%)	0.0514 (+38%)
	Waterfall _{w/o-Requirement}	0.29 (-37%)	0.6554 (-13%)	0.0109	0.2049 (+2%)	0.0916 (-16%)	0.0632 (+13%)	0.0567 (+52%)
	Waterfall _{w/o-Design}	0.31 (-33%)	0.6793 (-10%)	0	0.2001 (-1%)	0.0703 (-35%)	0.0575 (+3%)	0.0724 (+95%)
	Waterfall _{w/o-Testing}	0.19 (-59%)	0.5378 (-29%)	0.0040	0.1564 (-22%)	0.0842 (-22%)	0.0321 (-43%)	0.0421 (+13%)
claude-3-5-haiku	RawPrompt	0.21	0.3685	0	0.1659	0.0869	0.0435	0.0356
	Waterfall _{Full}	0.22 (+5%)	0.6255 (+70%)	0	0.1736 (+5%)	0.0825 (-5%)	0.0282 (-35%)	0.0629 (+77%)
	Waterfall _{w/o-Requirement}	0.24 (+14%)	0.6056 (+64%)	0.0024	0.1591 (-4%)	0.0807 (-7%)	0.0546 (+26%)	0.0261 (-27%)
	Waterfall _{w/o-Design}	0.29 (+38%)	0.6275 (+70%)	0	0.1734 (+5%)	0.0759 (-13%)	0.0477 (+10%)	0.0499 (+40%)
	Waterfall _{w/o-Testing}	0.17 (-19%)	0.4343 (+18%)	0.0021	0.1501 (-10%)	0.0813 (-6%)	0.0396 (-9%)	0.0292 (-18%)

Note: The *Security* and *Responsibility* columns are omitted from the table as all their values were zero.

SonarQube reports issue density normalized per 10 non-comment lines of code (ncLOC), defined as:

$$\text{Issue Density per 10 ncLOC} = \frac{\text{SonarQube Issues}}{\text{Total ncLOC}} \times 10, \quad (2)$$

where *SonarQube Issues* denotes the number of detected problems and *Total ncLOC* refers to the total non-comment lines of code. Each workflow’s generated code is imported into SonarQube as a separate project, and the resulting metrics are collected from the SonarQube dashboard. Higher issue density indicates poorer code quality, providing a complementary perspective to the test-based correctness results.

Results: Table 2 presents the comparative performance of all models under different Waterfall configurations.

Accuracy. Pass@1 accuracy declines under Waterfall variants for gpt-4o-mini and deepseek-chat, whereas claude-3.5-haiku benefits from structured prompting. Both *gpt-4o-mini* and *deepseek-chat* show a consistent decrease in accuracy when the Waterfall workflow is applied. For these models, the *RawPrompt* approach yields the highest performance (0.35 and 0.46 at class level; 0.6853 and 0.7529 at function level, respectively). Across all Waterfall variants, class-level accuracy declines by roughly 30–40% and function-level accuracy by 10–25%. Among the variants, *w/o Requirement* and *w/o Design* retain comparatively better results, while *w/o Testing* produces the weakest performance for both models. In contrast, *claude-3.5-haiku* exhibits the opposite trend: Waterfall variants generally enhance performance relative to *RawPrompt*. Notably, *w/o Design* achieves the highest overall accuracy (0.29 class, 0.6275 function), and even the full Waterfall workflow provides substantial gains, particularly at the function level (+70% over baseline). Interestingly, the results reveal that **Pass@1 drops significantly from function-level to class-level**, suggesting that it is far easier for a model to generate a single correct function than to produce an entire class where all functions are correct.

Software Quality. Reliability decreases across all models under Waterfall workflows, while maintainability improves clearly for gpt-4o-mini but shows mixed outcomes for deepseek-chat and claude-3.5-haiku. For *gpt-4o-mini*, all Waterfall variants lead to better maintainability, with the strongest improvement observed in *w/o-Testing* (−19%). However, reliability consistently worsens across all variants, indicating that the generated code becomes slightly less stable. In contrast, both *deepseek-chat* and *claude-3.5-haiku* display mixed results for maintainability: certain variants, such as *w/o-Testing*, show moderate improvement, while others, like *Full* or *w/o-Design*, slightly regress. Despite these fluctuations, both models share a common trend of reduced reliability across all Waterfall settings, suggesting that the stage-by-stage process may help maintain code organization but at the expense of overall stability.

Clean Code Attributes. Consistency increases across all models under Waterfall approaches while Intentionality shows mixed results. Adaptability clearly decreases for both gpt-4o-mini and deepseek-chat but shows mixed outcomes for claude-3-5-haiku. *gpt-4o-mini* achieves strong reductions in consistency issues (−32% to −43%), with mixed intentionality results and severe adaptability regressions (+200–390%). *Deepseek-chat* improves less consistently (consistency −16% to −35%, intentionality best in *w/o-Testing* at −43%, adaptability increases only +13–95%). By contrast, *claude-3.5-haiku* offers balanced improvements, especially in *w/o-Testing* (−6% consistency, −9% intentionality, −18% adaptability). Smaller models may behave this way because they follow each Waterfall step too strictly, which boosts consistency but limits adaptability.

RQ1 Summary: Waterfall workflows improve software quality and code cleanliness but reduce adaptability, intentionality, and Pass@1 accuracy with more reliability issues. Their effectiveness also varies by model; gpt-4o-mini and deepseek-chat perform worse, whereas claude-3.5-haiku shows significant improvement.

4.2 RQ2: What kinds of errors occur during class-level code generation?

Motivation: In this research question, we focus on identifying and understanding the types of errors that arise during class-level code generation when following the Waterfall process model. In RQ1, we found that although Waterfall-based workflows improved code quality and cleanliness, they also led to more runtime reliability issues and stage inconsistencies. Building on these findings, here we investigate why such failures occur by examining the types of runtime errors that emerge during class-level code generation. By categorizing and comparing these errors across different LLMs and workflow variants, we aim to uncover the underlying causes of coordination breakdowns.

Approach: We manually evaluated all 1,500 generated code samples (100 ClassEval tasks $\times$ 3 models $\times$ 5 workflows) to identify the most frequent error types. Each code sample was executed against its corresponding unit tests from the ClassEval benchmark, and the resulting errors were recorded. Error types were categorized following methodologies from prior studies[29, 37]. Specifically, SOEN-101 [29] classified failure types according to the official Python Interpreter documentation⁴, while Chatdev [37] identified successful compilations and recorded compiler-reported errors based on feedback logs between the testing process. During testing, simple syntax errors, such as missing or extra parentheses and full stops, were corrected manually by reviewing the code, and the evaluation was rerun. Repeated occurrences of the same error type within a single class (e.g., multiple *AssertionErrors*) were counted once to prevent overcounting. Two extra categories were added: (a) *Time Limit Exceeded*, for tests running longer than eight minutes, and (b) *One-Off Errors*, grouping rare errors that appeared only once or twice per model across all workflows, like *ModuleNotFoundError*, *re.error*, *RuntimeError* etc. Finally, we compiled the results into an error frequency chart (Table 3), where each frequency value represents the number of classes in which that error occurred.

Results: *Among the evaluated workflows, three error types ValueError, AssertionError, and TypeError account for most failures, each showing substantial growth when using Waterfall (ranging from about +40% to over +700%). ValueError increases the most among all error types, increasing from around 7 to 25 cases in several configurations AssertionError also increases, in some cases by more than 100%. TypeError typically doubles. Meanwhile, simpler runtime issues decline sharply. ZeroDivisionError nearly disappears (reductions of up to -100%), IndexError often drops to zero, and NameError decreases by over 90% in several workflows. KeyError increases from almost none to as many as*

five occurrences per workflow (about +100–500%), and both *OperationalError* and grouped *One-Off Errors* also rise.

Across all models, introducing the Waterfall workflow increases total runtime errors by 10–53% compared to the Raw Prompt baseline. The overall rise in errors is most pronounced for *GPT-4o-mini* (+49%), *DeepSeek-Chat* displays a more balanced response (+16–19%). In contrast, *Claude-3.5-haiku* shows a limited overall increase (+10–25%). Notably, removing the testing stage (*w/o-Test*) consistently leads to the highest failure rates across all models up to +53% and shows the importance of test awareness.

Across workflow variants, the full Waterfall process leads to the highest number of runtime errors, while removing stages gives mixed and model-specific effects. In most cases, the full Waterfall setup raises failures compared to Raw Prompt. For example, *GPT-4o-mini* has about 40 more errors under Waterfall (Full) (82 to 122, +49%), and *DeepSeek-Chat* adds nearly 15 (+19%) compared to when using Raw Prompt. When stages are removed, changes are small and inconsistent. Skipping *requirements* (*w/o-Req.*) or *design* (*w/o-Des.*) gives slight drops for some models. *DeepSeek-Chat*, for instance, falls by only a few errors, while *Claude-3.5-haiku* is almost unchanged. The clearest pattern is with *w/o-Test*. It yields the biggest increase across models, reaching 124 errors for *DeepSeek-Chat* and 118 for *GPT-4o-mini*. Across all workflows, *w/o-Test* performs the worst, *w/o-Req.* and *w/o-Des.* stay within 5% difference of each other but *Full* varies wildly compared to baseline. Overall, each stage affects reliability differently. Removing early stages has little impact, but skipping testing consistently makes the system much more error-prone.

RQ2 Summary: Waterfall workflows increase runtime errors by 10–53% compared to RawPrompt. *ValueError*, *AssertionError*, and *TypeError* dominate the failures, while *ZeroDivisionError* and other low-level mistakes decline. *DeepSeek-Chat* is the comparatively more stable model, whereas *GPT-4o-mini* and *Claude-3.5-haiku* show greater sensitivity to workflow design, indicating that structured prompting must be carefully tuned for each model.

4.3 RQ3: What are the underlying causes behind these failures?

Motivation: While RQ2 identified the types of runtime errors produced during multi-agent code generation, it did not explain why these errors occurred. In this research question, we aim to identify the primary reasons behind these failures by examining how the various stages of the workflow interact with each other. Specifically, we examine whether the errors arise from missing or inconsistent information passed between agents, incorrect implementation of design decisions, or limitations in the models' reasoning abilities. By identifying these causes, we can gain a deeper understanding of how workflow structure, model behaviour, and stage coordination contribute to the overall reliability of multi-agent code generation.

Approach: We first isolated all failing classes from the 1,500 code samples analyzed in RQ2 (100 ClassEval tasks $\times$ 3 models $\times$ 5 workflows). Each failed class was manually inspected to identify the root cause of the failure. For every sample, we compared the generated code against three main references:

⁴<https://docs.python.org/3/library/exceptions.html>

Table 3: Comparison of error types across three different models. Each value reports the raw count and relative change (%) compared to the model’s *RawPrompt* baseline across Waterfall variants.

Error Type	gpt-4o-mini					deepseek-chat					claude-3.5-haiku				
	Raw	Waterfall				Raw	Waterfall				Raw	Waterfall			
		Full	w/o-Req.	w/o-Des.	w/o-Test.		Full	w/o-Req.	w/o-Des.	w/o-Test.		Full	w/o-Req.	w/o-Des.	w/o-Test.
AssertionError	53	60 (+13%)	54 (+2%)	62 (+17%)	51 (-4%)	51	50 (-2%)	55 (+8%)	58 (+14%)	60 (+18%)	27	58 (+115%)	61 (+126%)	54 (+100%)	54 (+100%)
ValueError	7	25 (+257%)	24 (+243%)	22 (+214%)	28 (+300%)	7	24 (+243%)	17 (+143%)	15 (+114%)	29 (+314%)	3	22 (+633%)	17 (+467%)	17 (+467%)	25 (+733%)
TypeError	4	7 (+75%)	8 (+100%)	9 (+125%)	9 (+125%)	5	4 (-20%)	4 (-20%)	5	7 (+40%)	1	3 (+200%)	4 (+300%)	5 (+400%)	3 (+200%)
AttributeError	5	5	6 (+20%)	6 (+20%)	10 (+100%)	3	3	2 (-33%)	4 (+33%)	3	1	2 (+100%)	2 (+100%)	4 (+300%)	2 (+100%)
KeyError	0	4 (+400%)	4 (+400%)	2 (+200%)	3 (+300%)	1	3 (+200%)	2 (+100%)	3 (+100%)	3 (+100%)	0	2 (+200%)	4 (+400%)	5 (+500%)	1 (+100%)
FileNotFoundError	2	5 (+150%)	3 (+50%)	3 (+50%)	4 (+100%)	1	2 (+100%)	3 (+200%)	2 (+100%)	3 (+200%)	1	2 (+100%)	1	2 (+100%)	1
IndexError	2	4 (+100%)	0 (-100%)	0 (-100%)	1 (-50%)	2	0 (-100%)	0 (-100%)	0 (-100%)	1 (-50%)	1	1	0 (-100%)	1	0 (-100%)
NameError	3	2 (-33%)	1 (-67%)	1 (-67%)	3	2	1 (-50%)	2	2	2	51	3 (-94%)	2 (-96%)	2 (-96%)	20 (-61%)
SyntaxError	0	0	1 (+100%)	2 (+200%)	2 (+200%)	1	5 (+400%)	6 (+500%)	5 (+400%)	6 (+500%)	0	0	4 (+400%)	2 (+200%)	1 (+100%)
ZeroDivisionError	2	0 (-100%)	0 (-100%)	0 (-100%)	0 (-100%)	3	0 (-100%)	0 (-100%)	0 (-100%)	0 (-100%)	2	2	1 (-50%)	1 (-50%)	0 (-100%)
OperationalError*	0	1 (+100%)	2 (+200%)	2 (+200%)	1 (+100%)	1	2 (+100%)	0 (-100%)	0 (-100%)	1 (+100%)	0	1 (+100%)	2 (+200%)	1 (+100%)	1 (+100%)
Time Limit Exceeded	2	1 (-50%)	1 (-50%)	0 (-100%)	0 (-100%)	2	0 (-100%)	1 (-50%)	1 (-50%)	2	1	0 (-100%)	1	1	1
Exception	0	1 (+100%)	1 (+100%)	0	1 (+100%)	0	0	0	0	2 (+200%)	0	1 (+100%)	0	0	0
DeprecationError*	1	0 (-100%)	0 (-100%)	0 (-100%)	0	1	1	1	0 (-100%)	1	0	0	1	1	0
OSError	0	1 (+100%)	0	0	0	1	0 (-100%)	0	0	1	1	0 (-100%)	1	1	1
One-Off Errors	1	6 (+500%)	1	0 (-100%)	4 (+300%)	0	1 (+100%)	1 (+100%)	0	3 (+300%)	0	2 (+200%)	1 (+100%)	1 (+100%)	1 (+100%)
Total	82	122 (+49%)	106 (+29%)	109 (+33%)	118 (+44%)	81	96 (+19%)	94 (+16%)	95 (+17%)	124 (+53%)	89	99 (+11%)	102 (+15%)	98 (+10%)	111 (+25%)

Note: PyPDF2.errors.DeprecationError is shortened to DeprecationError*. sqlite3.OperationalError is shortened to OperationalError*.

ClassEval Task Description. We examined the problem description in ClassEval to check whether the generated code met the specified requirements and implemented the intended functionality. We also reviewed the task descriptions to identify any ambiguities, inconsistencies, or missing details that could affect code generation. **Unit Tests.** We reviewed the corresponding unit tests to identify which behaviors failed during execution and to determine whether the issue came from missing functionality, logic errors, or incorrect handling of test cases. We also examined whether the unit tests were consistent with the problem description and verified that they did not contain contradictions or inherent design flaws.

Reference Implementation. We compared the generated code with the canonical reference implementation to detect deviations in logic, data flow, and structural design that led to incorrect behavior. We also examined the reference implementation to determine whether it provided additional details beyond those stated in the problem description.

Following prior work on LLM fault analysis [37, 45], we labeled each identified issue using a structured taxonomy of error categories. These categories were grouped into broader types, as summarized in Table 4. Each category includes fine-grained subtypes and a brief explanation of the issue’s nature. Each class was assigned one primary cause to avoid duplication, but when a failure stemmed from several interrelated problems, it was cross-referenced with multiple relevant categories. We applied this process across gpt-4o-mini, deepseek-chat, and claude-3.5-haiku under the agent-based Waterfall workflows.

The taxonomy covers a wide spectrum of errors commonly observed in LLM-generated code [48]. *Missing Code* and *Return Mismatch* represent structural gaps in the code (e.g., missing or renamed functions, variables, classes, etc.) and output inconsistencies. *Input Validation* and *Semantic Failure* capture reasoning and logical flaws, such as invalid checks or incorrect algorithms. *Dataset* and *Environment* categories describe issues related to problem and test design, dependencies, or version incompatibilities.

Results: Overall, Dataset-related issues were the most common source of failure (47–62 cases), followed by reasoning and

logic errors such as Semantic Failure (50–55 cases) and output inconsistencies under Return Mismatch (21–34 cases), with Input Validation errors also increasing notably under Waterfall workflows. As shown in Table 5, these three categories *Semantic Failure*, *Return Mismatch*, and *Dataset* consistently dominated across all Waterfall variants and models. This pattern reflects persistent challenges in algorithmic reasoning, output consistency, and handling of incomplete or ambiguous task inputs. In contrast, *Input Validation* errors were rare in the RawPrompt baseline (2 cases) but became considerably more frequent under Waterfall workflows, rising to 18 cases. This increase suggests that while structured multi-agent workflows improve validation coverage, they can also surface deeper logical or constraint-related issues during testing, likely due to tighter coordination and stricter specification enforcement between stages.

Across all models, a large number of the failures stemmed from reasoning and output errors. Semantic Failure rose to 50–55 cases under Waterfall (from 35–49 in RawPrompt), while Return Mismatch remained frequent (21–34 cases), mainly due to Type and Format issues (~75–83% of return errors). *Semantic Failure* was one of the most common sources of errors, increasing sharply across all models: for claude-3.5-haiku from 18 to 54 cases (+36; ~200%), for deepseek-chat from 29 to 53 (+24; ~83%), and for gpt-4o-mini from 35 to 55 (+20; ~57%). Most stemmed from *Wrong Algorithm* (19–30 cases) and *Wrong Edge Case Handling* (4–12), indicating persistent weaknesses in logical precision and boundary reasoning; notably, gpt-4o-mini also showed a sharp increase in *Specification Violations* (2→12). *Return Mismatch* was also a common source of errors. For gpt-4o-mini, the number of these cases increased slightly by 3–5 (from 23 to 26–28, about 13–22%). For deepseek-chat, the counts varied across Waterfall variants, ranging from 21 to 34. Claude-3.5-haiku showed the largest increase, rising errors by around 108%. While *Arity Mismatch* declined slightly in the full Waterfall setup, suggesting improved functional completeness, the coordination across stages also introduced new inconsistencies in return types and formatting, revealing a trade-off between structural rigor and semantic reliability.

Table 4: Taxonomy of error categories with corresponding motivations describing the underlying causes of failures in class-level code generation.

Error Category	Motivation
Missing Code	Code components such as variables, functions, or classes are missing, renamed, or only partially implemented.
Renamed Variable	A variable name has been changed from what the specification requires.
Missing Variable	A required variable from the specification is missing in the code.
Missing Function	A function is called but not defined anywhere in the code.
Renamed Function	A function was renamed, but the test still uses the old name.
Missing Class	A class used in the code/test is not defined.
Renamed Class	A class was renamed but tests still use the old name.
Missing Import	Code uses a module or function without importing it.
Incomplete Implementation	Only part of the instructed functionality is implemented.
Return Mismatch	Returned outputs differ from expected results due to incorrect arity, order, type, or format.
Arity Mismatch	Function returns too many or too few values (e.g., 3 items instead of 2).
Order Mismatch	Function returns values in the wrong order (e.g., (b, a) instead of (a, b)).
Type Mismatch	The return value is of the wrong Python type (e.g., list instead of tuple).
Format Mismatch	The return value has incorrect formatting (e.g., casing, spacing, symbols).
Input Validation	The code fails to correctly handle user input due to being too strict, too lenient, or logically flawed.
Overrestrictive Validation	Adds unnecessary constraints not required by the problem.
Faulty Validation	Validation logic is incorrect or incomplete.
Missing Input Validation	No validation occurs where it should.
Semantic Failure	Logical or algorithmic errors causing functionality to deviate from intended behavior.
Spec Violation	Code violates explicit problem instructions.
Signature Mismatch	Function parameters differ from the expected specification.
Wrong Algorithm	Code implements an incorrect algorithm or formula.
Wrong Edge Case Handling	Fails to handle boundary cases properly.
Timeout	Code runs indefinitely or exceeds time limits.
Syntax Error	Code cannot compile or parse due to syntax issues.
Integration Error	A newly added feature conflicts with existing code.
Dataset	Errors arise from ambiguous, flawed, or inconsistent problem specifications or test data.
Faulty Spec	The problem description is incorrect or misleading.
Faulty Test	Test cases contain incorrect or inconsistent logic.
Missing Import (Test)	Test scripts fail due to missing imports.
Spec-Test Mismatch	Tests contradict the written problem specification.
Environment	Failures caused by dependency, compatibility, or versioning issues.
Version Incompatibility	Uses unsupported features in the given Python version.
Undeclared Dependency	Uses undeclared external packages.
Deprecated Dependency	Uses outdated or removed dependencies.

Input Validation errors rose sharply under different Waterfall variants, while Dataset errors stayed high or increased depending on the model. For *Input Validation*, all models showed a notable rise under Waterfall. When using gpt-4o-mini as the LLM model, the error numbers increased from 2 to 16 cases, with the highest count observed when the *requirements stage* was removed. deepseek-chat expanded from 3 to 18 ($\approx 500\%$), peaking when the *testing phase* was excluded, while using claude-3.5-haiku, the errors rose from a single case to 11 ($\approx 900\%$), reaching its maximum in the full Waterfall setup. Most of these errors were *Overrestrictive Validation*, such as unnecessary input checks added by the agents (gpt-4o-mini: 0–9; deepseek-chat: up to 12), or *Faulty Validation*, where the conditions were incorrect or incomplete (gpt-4o-mini: 2–6; deepseek-chat: 1–6). *Missing Validation* errors were rare, appearing only occasionally in deepseek-chat (0–2 cases) and claude-3.5-haiku (0–1), and entirely absent in gpt-4o-mini. For *Dataset* errors, gpt-4o-mini showed a moderate increase from 47

to 62 cases ($\approx 32\%$), with the highest count appearing when the design phase was omitted. deepseek-chat remained relatively stable, fluctuating slightly between 49 and 52 cases across configurations. In contrast, claude-3.5-haiku showed a sharp rise from 24 in the baseline to 59 under Waterfall ($\approx 146\%$), driven largely by *Faulty Specifications* (rising from 6 to about 24) and, to a lesser extent, *Faulty Tests* (5–10).

Missing Code errors dropped sharply under Waterfall workflows, falling from 4–50 cases in RawPrompt to as few as 1–2 across different models, while Environment related issues remained low overall, typically between 1–5 cases. For *Missing Code*, gpt-4o-mini reduced the errors from 4 to 1, and deepseek-chat showed an even larger improvement, decreasing from 10 to just a single case compared to the raw prompt. The most pronounced reduction occurred in claude-3.5-haiku, where errors fell from 50 in the baseline to only 1–2 across most Waterfall variants, an improvement of over 95%. These results indicate that the multi-stage structure of Waterfall workflows helped prevent incomplete implementations and improved overall code completeness. In contrast, *Environment* related issues showed minor fluctuations but stayed consistently low. Claude-3.5-haiku fully eliminated these problems in the full Waterfall configuration, while gpt-4o-mini and deepseek-chat showed small variations depending on which stage was removed. Subcategories like *Version Incompatibility* and *Undeclared Dependency* appeared only a few times, indicating that all models generally managed their dependencies and environments correctly.

RQ3 Summary: Waterfall workflows reduce structural errors such as *Missing Code* but increase reasoning and validation issues such as *Semantic Failure* and *Return Mismatch*. *Dataset* related errors remain the most common overall. Each model show trade-offs; gpt-4o-mini and deepseek-chat improve completeness, while claude-3.5-haiku face higher logic and validation failures.

5 Discussion

5.1 Implications

For Developers. Waterfall-style, role-specialized agents reliably surface inconsistencies and promote disciplined design and validation practices, resulting in cleaner, more maintainable class implementations. However, certain Waterfall activities, particularly design and validation, introduce additional architectural structure and constraint checks, which improve code robustness and maintainability but simultaneously worsen execution success and functional accuracy by adding complexity and stricter logic not required by the original specification. The trade-off is that additional structure often introduces *extra validation logic*, for example, tighter input checks, refactored interfaces, or modified boundary conditions that were not defined in the original requirements. These changes increase code robustness but can degrade functional accuracy (Pass@1) because benchmark test suites expect strict *requirements conformance* rather than defensive or production-grade behaviour.

Practical guidance. (i) *Requirements-driven validation:* restrict defensive checks to what is explicitly stated in the task specification;

Table 5: Qualitative analysis of error categories across models and Waterfall variants. Each cell shows the relative frequency of errors compared to the *RawPrompt* baseline, where red indicates an increase and green indicates a decrease.

Error Category	gpt-4o-mini					deepseek-chat					claude-3-5-haiku				
	Raw	Waterfall				Raw	Waterfall				Raw	Waterfall			
		Full	w/o-Req.	w/o-Des.	w/o-Test.		Full	w/o-Req.	w/o-Des.	w/o-Test.		Full	w/o-Req.	w/o-Des.	w/o-Test.
Missing Code	4	1	4	0	10	1	0	0	1	1	50	2	1	1	17
Renamed Variable	1	1	2	0	0	0	0	0	0	0	0	0	0	0	0
Missing Variable	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
Missing Function	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0
Renamed Function	0	0	0	0	6	1	0	0	0	1	0	0	0	0	0
Missing Class	1	0	0	0	0	0	0	0	0	0	49	2	1	1	17
Renamed Class	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
Missing Import	2	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Incomplete Implementation	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
Return Mismatch	23	26	28	27	26	21	23	25	34	32	13	22	27	27	26
Arity Mismatch	4	1	6	3	3	5	4	5	7	3	3	5	5	4	4
Order Mismatch	1	2	1	0	1	1	0	1	1	1	1	0	3	3	1
Type Mismatch	8	13	14	13	16	8	7	9	11	13	3	5	7	9	7
Format Mismatch	10	10	7	11	6	7	12	10	15	14	6	12	12	11	14
Input Validation	2	14	16	10	13	3	9	8	8	18	1	10	6	3	11
Overrestrictive Validation	0	9	10	6	7	0	6	5	4	12	0	6	3	2	7
Faulty Validation	2	5	6	4	6	1	3	2	2	6	1	3	3	1	4
Missing Input Validation	0	0	0	0	0	2	0	1	2	0	0	1	0	0	0
Semantic Failure	35	51	49	55	50	29	52	50	47	53	18	43	54	51	39
Spec Violation	2	12	7	9	7	1	5	3	3	9	0	4	4	3	2
Signature Mismatch	0	0	1	2	3	0	0	0	0	1	0	0	0	1	0
Wrong Algorithm	22	25	25	25	23	19	24	22	24	23	13	29	26	30	23
Wrong Edge Case Handling	9	10	8	12	8	4	5	5	5	1	4	6	9	7	7
Timeout	2	1	1	0	0	2	0	1	0	2	1	0	1	1	1
Syntax Error	0	0	3	6	6	3	14	17	13	14	0	1	12	6	3
Integration Error	0	3	4	1	3	0	4	2	1	3	0	3	2	3	3
Dataset	47	59	51	62	52	52	49	52	48	52	24	49	59	51	44
Faulty Spec	15	22	20	26	21	19	19	21	21	21	6	22	24	22	20
Faulty Test	6	7	7	7	6	9	8	9	7	9	5	8	10	7	5
Missing Import (Test)	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0
Environment	1	1	2	1	5	1	1	2	1	2	1	0	1	1	3
Version Incompatibility	0	0	2	1	3	0	0	0	0	1	0	0	0	0	0
Undeclared Dependency	0	1	0	0	1	0	0	1	0	0	1	0	0	0	3
Deprecated Dependency	1	0	0	0	1	1	1	1	0	1	0	0	1	1	0

treat any extended policy as optional. (ii) *Stage optimization*: retain requirement clarification and test planning stages, but relax rigid architectural enforcement when functional correctness is the priority. (iii) *Interface conformance check*: include a lightweight verification step to ensure output signatures, data types, and edge-case handling remain aligned with the defined contract. (iv) *Model-stage alignment*: select models that respond positively to structured inputs, and use role-specific exemplars or context windows to prevent over-specification during design and validation phases.

For Researchers and Benchmark Designers. Class-level generation shifts errors from low-level slips (e.g., missing code) toward higher-level reasoning and contract violations (semantic failures, return/type/format mismatches), revealing that stricter process *changes* the failure profile rather than uniformly improving it. This has two consequences. *For researchers*: (i) Explore *adaptive/conditional pipelines* that enable or skip stages only when they raise correctness; (ii) add *cross-stage memory and reconciliation* so design/validation can't silently override the specified contract; (iii) introduce *conflict-resolution* (a meta-agent) that prefers benchmark contracts when spec vs. design disagree. *For benchmark designers*: (i) Publish explicit *contract policies* (input ranges, error handling, formatting/typing rules) to reduce "reasonable but wrong" over-validation; (ii) report *dual scores*: correctness (Pass@k) and design/cleanliness (e.g., maintainability/consistency metrics) so process benefits aren't invisible; (iii) add *spec-faithfulness checks* (API/signature/return-format conformance) and tag cases where

extra policy is allowed vs. penalized; (iv) include *ambiguity labels* or test rationales so agents learn when *not* to tighten behaviour.

5.2 Threats to Validity

Internal Validity. We performed qualitative analysis to identify failure-inducing factors for RQ3. However, such analysis is inherently subjective and may not always reveal the exact cause of errors. Future work could incorporate LLM-based evaluators to diagnose error sources and reduce manual effort [26]. Minor inconsistencies may also arise in how errors are categorized across models. To minimize this, we randomly sampled generated outputs, applied consistent labelling, and reviewed all samples within the same time frame to reduce bias.

External Validity. A limitation of this work is the use of Pass@1 as the primary measure of functional correctness. Although Pass@1 evaluates only the first generated output and can be volatile, it remains a widely adopted and interpretable metric in LLM-based code generation research, used by Codex [13], AlphaCode [28], and CodeT5+ [42]. This choice allows for consistent comparison across workflows and models.

Construct Validity. Our Multi-agent Waterfall workflow approximates its real-world counterpart, with LLMs playing the roles typically performed by humans. Different versions of the Waterfall model and varying phase combinations could affect results. For test generation, we used LLM agents to produce unit-test suites, following prior studies that employ multi-agent LLM workflows for software development such as ChatDev [37], MetaGPT [21],

and SOEN-101 [29]. These works demonstrate that LLM agents can generate runnable tests by interpreting code behaviour, compiler feedback, and execution traces, which we adopt in our setup. Future work could incorporate automated test-generation tools such as Pynguin [31] or explore hybrid LLM plus search-based methods [8] to further improve coverage and fault-detection capability [9].

6 Conclusion

In this paper, we have evaluated multi-agent LLM workflows within the Waterfall model for class-level code generation using ClassEval. The experiments show that Waterfall workflows do not consistently improve functional correctness but yield cleaner, more maintainable code with fewer structural issues such as missing methods. Runtime and logical errors increase slightly, often due to stricter validation, semantic failures, and incorrect returns. Among development stages, Testing has the biggest impact, improving verification but reducing maintainability, while Requirement and Design play minor roles. Overall, the complete Waterfall workflow produces structured and readable code but limits flexibility and correctness, revealing a trade-off between disciplined process control and adaptive reasoning in multi-agent LLM systems.

References

- [1] Windsurf AI. 2025. Windsurf. <https://windsurf.ai>.
- [2] Maha Alharbi and Mohammad Alshayeb. 2025. Automatic Code Generation Techniques: A Systematic Literature Review. *Automated Software Engineering* 33, 1 (Sept. 2025), 4. doi:10.1007/s10515-025-00551-3
- [3] Anonymous. 2025. Repository & dataset. <https://anonymous.4open.science/r/LLMCLWA-3XB/README.md>
- [4] Anthropic. 2025. Claude. <https://claude.ai>.
- [5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
- [6] Berkay Berabi, Alexey Gronskey, Veselin Raychev, Gishor Sivanrupan, Victor Chibotaru, and Martin Vechev. 2024. Deepcode AI fix: Fixing security vulnerabilities with large language models. *arXiv preprint arXiv:2402.13291* (2024).
- [7] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. *arXiv:2403.17134*
- [8] Lior Broide and Roni Stern. 2025. EvoGPT: Enhancing Test Suite Robustness via LLM-Based Generation and Genetic Optimization. *arXiv preprint arXiv:2505.12424* (2025).
- [9] Arda Celik and Qusay H. Mahmoud. 2025. A Review of Large Language Models for Automated Test Case Generation. *Machine Learning and Knowledge Extraction* 7, 3 (2025). doi:10.3390/make7030097
- [10] Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. 2025. Why Do Multi-Agent LLM Systems Fail? *arXiv preprint arXiv:2503.13657* (2025).
- [11] Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. 2024. Code search is all you need? improving code suggestions with code search. In *Proceedings of the IEEE/ACM 46th international conference on software engineering*. 1–13.
- [12] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2024. Reasoning runtime behavior of a program with llm: How far are we? *arXiv preprint arXiv:2403.16437* (2024).
- [13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [14] Cursor. 2025. Cursor. <https://www.cursor.sh>.
- [15] DeepSeek. 2025. DeepSeek Chat. <https://chat.deepseek.com>.
- [16] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083* (2025).
- [17] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861* (2023).
- [18] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering* (2024).
- [19] GitHub. 2025. GitHub Copilot. <https://github.com/features/copilot>.
- [20] Xinyi He, Jiaru Zou, Yun Lin, Mengyu Zhou, Shi Han, Zejian Yuan, and Dongmei Zhang. 2024. Cocost: Automatic complex code generation with online searching and correctness testing. *arXiv preprint arXiv:2403.13583* (2024).
- [21] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiwu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. International Conference on Learning Representations, ICLR.
- [22] Shuyang Hou, Haoyue Jiao, Zhangxiao Shen, Jianyuan Liang, Anqi Zhao, Xiaopu Zhang, Jianxun Wang, and Huayi Wu. 2024. Chain-of-Programming (CoP): Empowering Large Language Models for Geospatial Code Generation. *arXiv preprint arXiv:2411.10753* (2024).
- [23] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515* (2024).
- [24] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of LLM-based explainable fault localization. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1424–1446.
- [25] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).
- [26] Haitao Li, Qian Dong, Junjie Chen, Huixue Su, Yujia Zhou, Qingyao Ai, Ziyi Ye, and Yiqun Liu. 2024. Llm-as-judges: a comprehensive survey on llm-based evaluation methods. *arXiv preprint arXiv:2412.05579* (2024).
- [27] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–23.
- [28] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustín Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. doi:10.1126/science.abq1158 [arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158](https://www.science.org/doi/pdf/10.1126/science.abq1158)
- [29] Feng Lin, Dong Jae Kim, et al. 2024. SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents. *arXiv preprint arXiv:2403.15852* (2024).
- [30] Yizhou Liu, Pengfei Gao, Xincheng Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. 2024. MarsCode Agent: AI-native Automated Bug Fixing. *arXiv:2409.00899*
- [31] Stephan Lukacszyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.
- [32] Xinbei Ma, Yeyun Gong, Pengcheng He, Nan Duan, et al. 2023. Query rewriting in retrieval-augmented large language models. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.
- [33] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. *arXiv:2303.17651*
- [34] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. 2023. Clarifygpt: Empowering llm-based code generation with intention clarification. *arXiv preprint arXiv:2310.10996* (2023).
- [35] Minh Huynh Nguyen, Thang Phan Chau, Phong X Nguyen, and Nghi DQ Bui. 2025. Agilecoder: Dynamic collaborative agents for software development based on agile methodology. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 156–167.
- [36] OpenAI. 2025. ChatGPT. <https://chat.openai.com>.
- [37] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924* (2023).
- [38] Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. The impact of input order bias on large language models for software fault localization. *arXiv preprint arXiv:2412.18750* (2024).
- [39] Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936* (2023).
- [40] Tabnine. 2025. Tabnine. <https://www.tabnine.com>.
- [41] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936.

- [42] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).
- [43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [44] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).
- [45] Pengyu Xue, Linhao Wu, Zhen Yang, Chengyi Wang, Xiang Li, Yuxiang Zhang, Jia Li, Ruikai Jin, Yifei Pei, Zhaoyan Shen, et al. 2025. ClassEval-T: Evaluating Large Language Models in Class-Level Code Translation. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1421–1444.
- [46] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *arXiv:2405.15793*
- [47] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. ThinkRepair: Self-Directed Automated Program Repair. In *ISSTA 2024: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, United States, 1274–1286.
- [48] Adam Yuen, John Pangas, Md Mainul Hasan Polash, and Ahmad Abdellatif. [n. d.]. Prompting Matters: Assessing the Effect of Prompting Techniques on LLM-Generated Class Code. ([n. d.]).
- [49] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (Vienna, Austria) (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604. doi:10.1145/3650212.3680384
- [50] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989* (2023).
- [51] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372* (2023).
- [52] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877* (2024).