Automated Fault-Tolerant Code Generation via LLMs: A Diversity-Enhanced and Quality-Assured Approach

Ding Wenjie[a], Wei Zhenghe[b], Liu Zhihao[a], Cai Yi[a], Ma Xiangyue[a], Zheng Zheng[a,*]

[a]*School of Automation Science and Electrical Engineering, Beihang university, Beijing, 100091, China*
[b]*Beijing University of Technology, Beijing, 100124, China*

Abstract

N-version programming has been extensively employed in safety-critical domains for fault tolerance. However, traditional manual approaches demand substantial development effort. The advent Large Language Models(LLMs) trained on extensive code corpora, has demonstrated revolutionary code generation capabilities at significantly reduced cost, rendering LLM-driven fault-tolerant code development feasible. Nevertheless, LLMs are inherently constrained by their training and inferring paradigms, tending to produce homogeneous outputs with limited diversity. To address these limitations, we propose DeQoG, a framework of **D**iversity-**E**nhanced **Q**uality-Assured fault-tolerance C**o**de **G**eneration. We introduce the Hierarchical Isolation and Local Expansion (HILE) and Iterative Retention, Questioning, and Negation (IRQN) methods to rigorously stratify the LLM's output space and compel deep exploration, thereby maximizing diversity across all stages of fault-tolerant N-version code development. In addition, DeQoG incorporates a feedback-based iterative repair(FBIR) mechanism and a structured workflow orchestration, ensuring superior code quality and enhancing the output certainty and controllability of LLMs. Extensive experiments on five benchmark datasets demonstrate that DeQoG significantly surpasses baseline methods in improving N-version code diversity and correctness, as well as system reliability during fault injection.

Keywords: Software Fault Tolerance, LLM Diversity Enhancement, N-version Code Generation, Gen-AI Engineering

1. Introduction

Software reliability is paramount in safety-critical domains where failures can cause catastrophic consequences in aerospace[1, 2], healthcare[3], and autonomous systems[4, 5, 6]. However, eliminating all defects through testing and verification alone is infeasible given the inherent complexity of modern software[7, 8, 9]. Consequently, fault-tolerant design, which enabling systems to maintain functionality despite failures, has become essential in contemporary software engineering practice.

N-version programming addresses this challenge by developing functionally equivalent yet diverse implementations that fail independently under identical conditions. This redundancy-based approach significantly enhances system survivability when confronted with defects, hardware failures, and malicious attacks[10, 11]. However, traditional manual N-version development faces two critical barriers. First, employing multiple independent teams incurs prohibitive costs that scale linearly with version count[12, 13]. Second, ensuring genuine diversity remains difficult[14, 15, 16], as developers with similar training tend toward algorithmically similar solutions, undermining the fundamental premise of independent failures. While early automated synthesis approaches—from rule-based methods[17] to neural program synthesis[18]—reduced development costs, they inherited the single-solution focus of conventional code generation, leaving the diversity-cost dilemma fundamentally unresolved.

Recent LLMs trained on extensive code corpora present unprecedented opportunities for automated N-version programming[19]. Their remarkable code generation capabilities[20, 21, 22, 23], combined with dramatically reduced costs compared to manual development, suggest a potential breakthrough for scalable fault-tolerant systems. However, most LLMs are inherently optimized during training to maximize single-solution correctness[24], exhibiting strong bias toward

[*]Corresponding author.

methodologically similar implementations despite superficial syntactic variations[25]. As diversity requirements intensify, this limitation manifests in two failure modes, that is, LLMs either produce shallow variants sharing identical algorithmic logic—offering illusory diversity that provides no fault tolerance—or resort to hallucinated solutions that sacrifice correctness for novelty[26, 27]. Moreover, LLMs lack the determinism and controllability required for production deployment, relying on brittle natural language prompts whose behavior remains unpredictable across generation attempts[28, 27]. These challenges reveal a critical insight: powerful as they are, LLMs cannot serve as standalone solutions for N-version fault-tolerance code generation. So, directly applying LLMs or employing rudimentary prompts will produce substantial proportions of incorrect and redundant implementations, with effectiveness severely constrained by task complexity and domain characteristics, and even has strict code location and type requirements for the oriented tasks[19]. Therefore, there is still a certain research gap in how to use LLMs to generate general N-version fault-tolerant code.

To bridge this gap, we propose DeQoG (**D**iversity-**e**nhanced **Q**uality-**a**ssured **G**eneration), a systematic framework that integrates established fault-tolerant software engineering methodologies with LLM capabilities. Unlike existing prompt engineering approaches that treat LLMs as complete black boxes[29], DeQoG introduces three synergistic algorithmic and engineering innovations grounded in LLM reasoning mechanisms. **Hierarchical Isolation and Local Expansion** method optimizes token allocation across cognitive levels, preventing the resource waste caused by LLMs' tendency to prematurely fixate on conventional paradigms, thereby maximizing diversity exploration across solution space, implementation strategy, and code structure. **Iterative Retain-Question-Negate** method partially counteracts LLMs' inherent bias toward over-rewarding optimal solutions by systematically challenging and negating generated outputs, forcing exploration of broader solution spaces beyond high-probability responses. **Feedback-Based Iterative Repair** method, inspired by Test-Driven Development[30], ensures correctness without sacrificing diversity through generation-test-repair cycles. Finally, we designed a controllable process to convert the unpredictable LLM output in the above three methods into deterministic templated output.

This design exemplifies how software engineering principles—requirements decomposition, iterative refinement, and deterministic control—can systematically enhance LLM capabilities for fault-tolerant code generation engineering practice.

We make the following contributions:

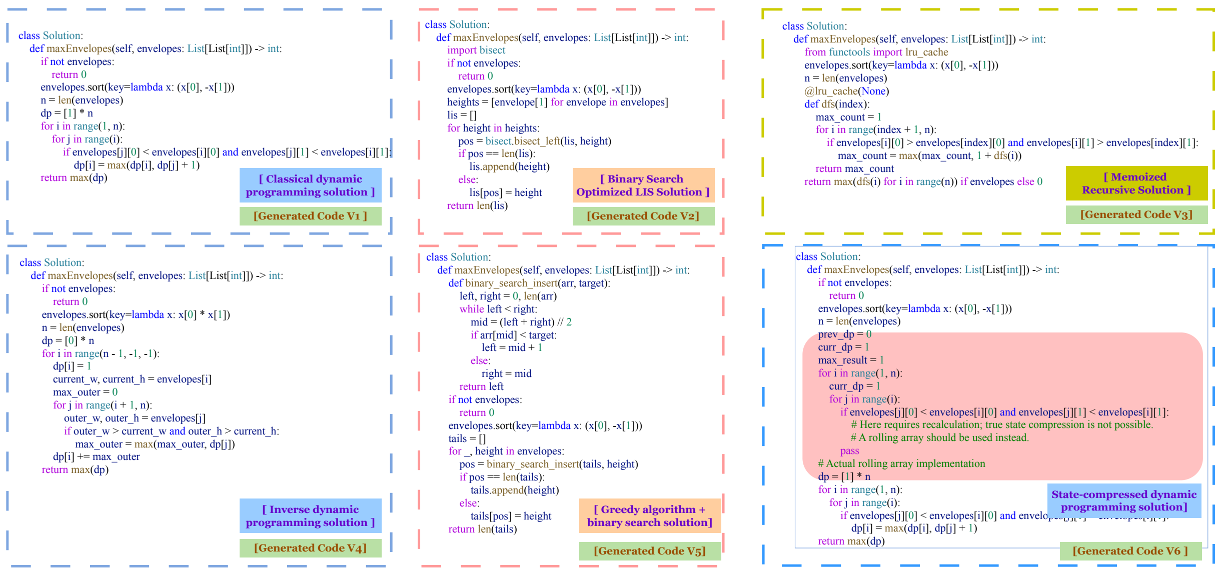
- We propose DeQoG, the first systematic GenAI-enabled framework specifically engineered for automated N-version fault-tolerant code generation. By integrating HILE and IRQN for diversity enhancement, along with FBIF and deterministic workflow orchestration for quality assurance and system reliability, DeQoG demonstrates a novel paradigm of addressing the diversity-correctness challenge in LLM-based fault-tolerant software engineering.

- We construct MIPD (Multi-Implementation Programming Dataset), a specialized benchmark of 100 tasks specifically designed for N-version fault-tolerant code generation. Unlike existing benchmarks that predominantly contain single-solution tasks, each MIPD task guarantees at least three diverse algorithmic implementations, providing an appropriate testbed for evaluating diversity generation and conducting fault injection experiments. We publicly release MIPD, fault injection protocols, and implementation components of DeQoG at https://github.com/dinger4/DeQoG.

- Extensive experiments across five benchmarks and four state-of-the-art LLMs demonstrate DeQoG's superiority over vanilla LLM and Best-of-N baselines. Compared to generated N-versions code of Vanilla LLMs, DeQoG improves the diversity by 35.65% on average, the correctness by 15.9% on average, and the reliability by 3.06% to 25.64% under the two-level fault injection of the system. Ablation studies validate the necessity of both diversity enhancement and quality assurance components.

2. Motivation

As noted in Section 1, while the potential of LLMs for automating N-version programming has been preliminarily established[19, 31], current LLM-based approaches still suffer from fundamental deficiencies in code diversity [32, 33, 34]. We now detail the challenges constraining LLM-based N-version fault-tolerant code generation through concrete examples. On the one hand, similar to human developers, LLMs exhibit inherent bias toward optimal or common solutions due to overlapping training data distributions[35, 36]. Even with temperature tuning and prompt engineering, LLMs


[Classical dynamic programming solution]
[Generated Code V1]
[Binary Search Optimized LIS Solution]
[Generated Code V2]
[Memoized Recursive Solution]
[Generated Code V3]
[Inverse dynamic programming solution]
[Generated Code V4]
[Greedy algorithm + binary search solution]
[Generated Code V5]
State-compressed dynamic programming solution]
[Generated Code V6]

```python
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        if not envelopes:
            return 0
        envelopes.sort(key=lambda x: (x[0], -x[1]))
        n = len(envelopes)
        dp = [1] * n
        for i in range(1, n):
            for j in range(i):
                if envelopes[j][0] < envelopes[i][0] and envelopes[j][1] < envelopes[i][1]:
                    dp[i] = max(dp[i], dp[j] + 1)
        return max(dp)
```

```python
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        import bisect
        if not envelopes:
            return 0
        envelopes.sort(key=lambda x: (x[0], -x[1]))
        heights = [envelope[1] for envelope in envelopes]
        lis = []
        for height in heights:
            pos = bisect.bisect_left(lis, height)
            if pos == len(lis):
                lis.append(height)
            else:
                lis[pos] = height
        return len(lis)
```

```python
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        from functools import lru_cache
        envelopes.sort(key=lambda x: (x[0], -x[1]))
        n = len(envelopes)
        @lru_cache(None)
        def dfs(index):
            max_count = 1
            for i in range(index + 1, n):
                if envelopes[i][0] > envelopes[index][0] and envelopes[i][1] > envelopes[index][1]:
                    max_count = max(max_count, 1 + dfs(i))
            return max_count
        return max(dfs(i) for i in range(n)) if envelopes else 0
```

```python
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        if not envelopes:
            return 0
        envelopes.sort(key=lambda x: x[0] * x[1])
        n = len(envelopes)
        dp = [0] * n
        for i in range(n - 1, -1, -1):
            dp[i] = 1
            current_w, current_h = envelopes[i]
            max_outer = 0
            for j in range(i + 1, n):
                outer_w, outer_h = envelopes[j]
                if outer_w > current_w and outer_h > current_h:
                    max_outer = max(max_outer, dp[j])
            dp[i] += max_outer
        return max(dp)
```

```python
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        def binary_search_insert(arr, target):
            left, right = 0, len(arr)
            while left < right:
                mid = (left + right) // 2
                if arr[mid] < target:
                    left = mid + 1
                else:
                    right = mid
            return left
        if not envelopes:
            return 0
        envelopes.sort(key=lambda x: (x[0], -x[1]))
        tails = []
        for _, height in envelopes:
            pos = binary_search_insert(tails, height)
            if pos == len(tails):
                tails.append(height)
            else:
                tails[pos] = height
        return len(tails)
```

```python
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        if not envelopes:
            return 0
        envelopes.sort(key=lambda x: (x[0], -x[1]))
        n = len(envelopes)
        prev_dp = 0
        curr_dp = 1
        max_result = 1
        for i in range(1, n):
            curr_dp = 1
            for j in range(i):
                if envelopes[j][0] < envelopes[i][0] and envelopes[j][1] < envelopes[i][1]:
                    # Here requires recalculation; true state compression is not possible.
                    # A rolling array should be used instead.
                    pass
        # Actual rolling array implementation
        dp = [1] * n
        for i in range(1, n):
            for j in range(i):
                if envelopes[j][0] < envelopes[i][0] and envelo
                    dp[i] = max(dp[i], dp[j] + 1)
        return max(dp)
```

Figure 1: Superficial Variations and Reliability Issues in Single LLM

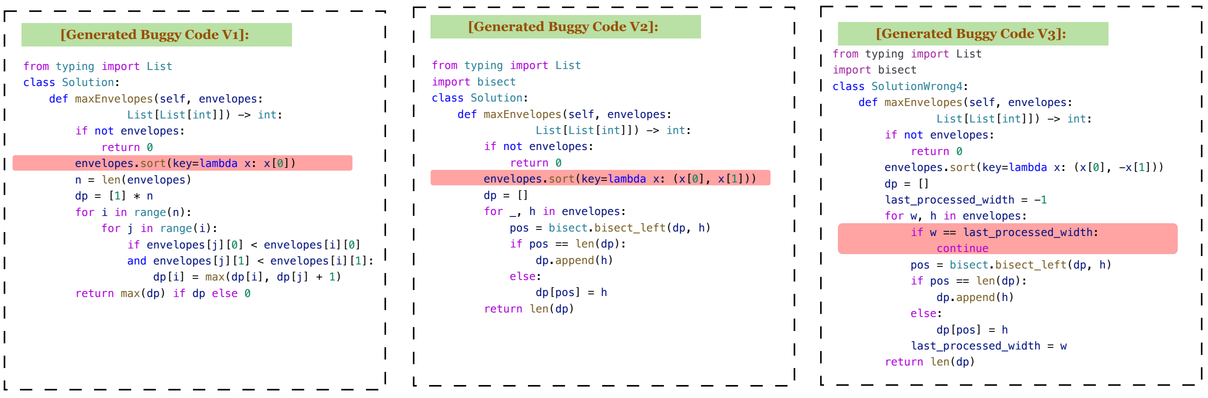
typically generate only syntactic variations or superficial logical changes while preserving highly similar algorithms[37, 38]. On the other hand, despite dramatically reduced development costs, code quality also simultaneously degrades compared to manual development. However, in the process of generating N-version fault-tolerant code, the requirement for diversity consumes a large amount of reasoning ability, causing LLMs to sacrifice correctness for novelty[39, 33], and increasingly creating seemingly reasonable but inefficient or even wrong solutions, exacerbating this correctness crisis.

As shown in Figure 1, we prompted Claude-Sonnet 4 to generate multiple algorithmic implementations for a dynamic programming competition task as shown in LeetCode Problem 354. Versions 1, 4, and 6 all employ dynamic programming approaches with only minor implementation differences. While Version 4 differs from Version 1 in sorting strategy and loop direction, both rely on identical underlying algorithmic logic. Version 6 presents a particularly instructive case: although the LLM claims to provide a state-compressed dynamic programming solution, the actual code expose that state compression proved infeasible during implementation (highlighted in the red code, each $dp[i]$ calculation depends on all previous $dp[j]$ values), forcing the implementation to regress to the same standard DP approach as Version 1 while introducing multiple lines of redundant code. Versions 2 and 5 are essentially identical greedy-based binary search methods, differing only in that Version 2 invokes the bisect library while Version 5 manually implements the binary search. Version 3 adopts a recursive approach distinct from the previous versions; however, it encountered timeout errors during testing, indicating that the recursive method is not entirely suitable for this task and generates inefficient code.

This pattern extends across models. When tasked with implementing large number arithmetic, GPT-4, Claude-Opus-4, and DeepSeek-R1 converge to identical digit-by-digit addition strategies with carry propagation, differing only in peripheral details (negative number handling, loop direction)—demonstrating cross-model training bias.

This phenomenon confirms the first challenge. When prompted to generate multiple implementations, LLMs exhibit strong bias toward algorithmically similar solutions despite syntactic variations. This homogeneity stems from two sources: shared training data distributions[35] and RLHF[40]-induced convergence toward optimal solutions [41]. Temperature tuning and prompt engineering produce only shallow variants that preserve core algorithmic structures [37, 38]. Such shallow diversity provides minimal fault tolerance. If the shared algorithmic core contains a design flaw or becomes inapplicable under specific conditions, all versions relying on the same methodology fail simultaneously, severely undermining the fault-tolerant system's reliability.

Figure 1 also reveals a critical trade-off: as diversity requirements intensify and push LLMs toward their reasoning capacity limits, models increasingly sacrifice correctness for novelty. More concerning, however, is that even when operating well within their capabilities, LLMs generate code of insufficient reliability for production deployment. We substantiate this claim



[Generated Buggy Code V1]:

```python
from typing import List
class Solution:
    def maxEnvelopes(self, envelopes:
                List[List[int]]) -> int:
        if not envelopes:
            return 0
        envelopes.sort(key=lambda x: x[0])
        n = len(envelopes)
        dp = [1] * n
        for i in range(n):
            for j in range(i):
                if envelopes[j][0] < envelopes[i][0]
                and envelopes[j][1] < envelopes[i][1]:
                    dp[i] = max(dp[i], dp[j] + 1)
        return max(dp) if dp else 0
```

[Generated Buggy Code V2]:

```python
from typing import List
import bisect
class Solution:
    def maxEnvelopes(self, envelopes:
                List[List[int]]) -> int:
        if not envelopes:
            return 0
        envelopes.sort(key=lambda x: (x[0], x[1]))
        dp = []
        for _, h in envelopes:
            pos = bisect.bisect_left(dp, h)
            if pos == len(dp):
                dp.append(h)
            else:
                dp[pos] = h
        return len(dp)
```

[Generated Buggy Code V3]:

```python
from typing import List
import bisect
class SolutionWrong4:
    def maxEnvelopes(self, envelopes:
                List[List[int]]) -> int:
        if not envelopes:
            return 0
        envelopes.sort(key=lambda x: (x[0], -x[1]))
        dp = []
        last_processed_width = -1
        for w, h in envelopes:
            if w == last_processed_width:
                continue
            pos = bisect.bisect_left(dp, h)
            if pos == len(dp):
                dp.append(h)
            else:
                dp[pos] = h
            last_processed_width = w
        return len(dp)
```

Figure 2: Typical Error Patterns in LLM-Generated N-Version Code (LeetCode Problem 354, CodeLLaMA)

through an experiment using CodeLLaMA to generate N-version fault-tolerant implementations for LeetCode Problem 354. Analysis reveals three distinct defect categories (highlighted in red in Figure 2): Version 1 exhibits algorithmic misunderstanding—sorting envelopes by width only while the dynamic programming logic compares both dimensions, creating logical inconsistency that violates problem constraints. Version 2 contains a sorting direction error where the comparison operator is inverted, fundamentally compromising algorithmic correctness. Version 3 introduces erroneous optimization logic that skips envelopes with identical widths. While seemingly reasonable (since equal-width envelopes cannot nest), this logic is redundant—the sorting procedure already handles this constraint—causing the algorithm to miss valid state transitions and produce incorrect results. These error patterns align with extensive documentation in prior code generation research, confirming that despite rapid quality improvements, LLM-generated code correctness remains unreliable. For N-version fault-tolerant systems where reliability depends on collective version correctness, such deficiencies pose critical risks. This quality assurance gap represents a fundamental bottleneck constraining LLM adoption in fault-tolerant programming domains.

In addition, current LLM-based GenAI systems to rely on brittle natural language prompts whose logic and behavior lack transparency, making predictability, reproducibility, and system quality difficult to guarantee[41, 40, 42]. The challenges above collectively reveal a critical insight: LLMs cannot serve as standalone solutions for fault tolerance code generation but require systematic software engineering frameworks to govern determinism and constrain unreliable outputs[43]. This motivates our proposal of DeQoG, a comprehensive GenAI-enabled framework specifically engineered for fault-tolerant N-version code generation.

3. Approach

3.1. Overview

To address the diversity and correctness challenges identified in Sections 1 and 2, DeQoG is designed around three core principles: **(i)** treating diversity as a fundamental requirement through hierarchical decomposition across thinking space, solution space, and implementation space; **(ii)** transforming quality assurance into a proactive improvement mechanism through closed-loop generation-test-repair cycles; and **(iii)** converting probabilistic LLM behaviors into controllable processes through structured workflow orchestration.

Building on these principles, DeQoG adopts a five-stage architecture that transforms task descriptions into N-version fault-tolerant code, as illustrated in Figure 3. The workflow is coordinated by a Decision-making and task-dispatching LLM that establishes deterministic transition mechanisms through structured control: stage transitions are governed by condition evaluation, with retry loops triggered by recoverable failures and rollback mechanisms activated by fundamental defects. Cross-stage memory continuously preserves task context, generation history, and feedback accumulation, while dynamic prompt engineering combines static segments (identity, objectives, constraints) with real-time updated dynamic segments (state descriptions, operation feedback, historical records).

Stage 1: Understanding & Collecting. This stage establishes deep problem comprehension by parsing task descriptions and leveraging knowledge search tools for domain-relevant information. Critically, the design enforces strict prohibition of premature solution exploration, forcing cognitive resources to focus on "understanding what to solve" rather than "how to solve it", thereby avoiding implicit constraint omissions caused by cognitive fixation.

Stage 2: Diversity Enhancing. Building upon problem understanding, this stage generates multi-level di-

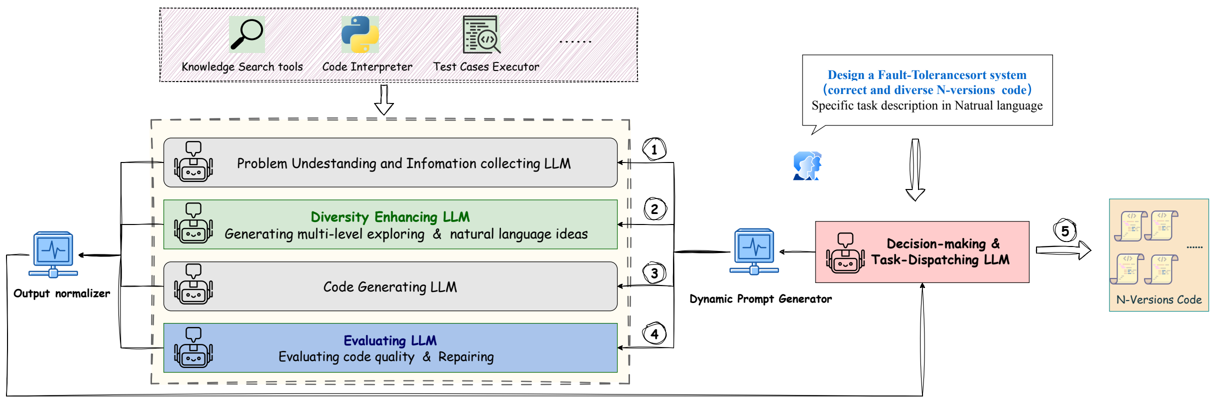

Knowledge Search tools
Code Interpreter
Test Cases Executor
......
Design a Fault-Tolerancesort system
(correct and diverse N-versions code)
Specific task description in Natrual language
Problem Undestanding and Infomation collecting LLM
Diversity Enhancing LLM
Generating multi-level exploring & natural language ideas
Code Generating LLM
Evaluating LLM
Evaluating code quality & Repairing
1
2
3
4
5
Output normalizer
Dynamic Prompt Generator
Decision-making & Task-Dispatching LLM
......
N-Versions Code

Figure 3: DeQoG System Architecture from a Software Engineering Perspective

verse solutions encompassing natural language algorithmic approaches, pseudocode-level implementation strategies, and code construction schemes. To address LLMs' tendency toward implementation homogeneity, we deploy two synergistic mechanisms: Hierarchical Isolation and Local Expansion vertically partitions generation into independent cognitive levels with isolation barriers, maximizing token utilization for intra-level reasoning; Iterative Retention, Questioning, and Negation , embedded as an adversarial method within each level, systematically challenges and negates existing outputs to force exploration beyond high-probability responses. This stage strictly prohibits generating executable code to prevent language syntax from reverse-constraining algorithmic choices. *Detailed algorithm and theoretical foundations are presented in Section 3.2*

Stage 3: Code Generating. LLMs map each abstract strategy from Stage 2 to concrete executable code implementations. The final code collection encompasses both macro-level algorithmic differences and micro-level implementation variations, achieving comprehensive diversity coverage.

Stage 4: Evaluating & Repairing. This stage validates functional correctness through comprehensive test execution. To address code quality degradation from LLM hallucinations, we implement a Feedback-Based Iterative Repair mechanism that collects structured failure information (syntax errors, runtime exceptions, logic errors with diagnostic reports) and generates updated prompts for targeted repairs. The critical design ensures repairs remain *localized and minimal*—prompt constraints prevent algorithmic replacements, preserving diversity while improving correctness. When repair iterations exceed thresholds or fundamental defects are detected, rollback mechanisms return to Stage 2 or Stage 1, carrying accumulated knowledge to prevent repeated errors. *The complete FBIR workflow and quality assurance mechanisms are detailed in Section 3.3.*

Stage 5: Output Collection. A code collector aggregates all validated versions to generate the deployable N-version fault-tolerant code set.

Through these five processing stages, DeQoG systematically addresses diversity, quality, and controllability challenges, leveraging the task dispatcher's deterministic control, LLMs' specialized processing capabilities, and auxiliary tools to establish a complete automated framework for N-version fault-tolerant code generation.

3.2. *Diversity Enhancement Strategy*

Diversity represents the foundational challenge and primary requirement in N-version fault-tolerant code generation. Building upon the five-stage architecture established in Section 3.1, Stage 2: Diversity Enhancing represents the core innovation of DeQoG. This section presents the algorithmic foundations and implementation details of our diversity enhancement strategy, which systematically addresses the LLM homogeneity challenge identified in Section 2 through two synergistic mechanisms.

3.2.1. *Theoretical Foundation and Design Rationale*

From a requirements engineering perspective, we apply structured analysis, decomposition, and prioritization methods to the diversity generation problem. Given a code generation task $\tau = (P, T, C_{\text{ref}})$, where P denotes the problem description, $T = \{t_1, t_2, \ldots, t_m\}$ represents the test case set, and C_{ref} is the reference solution (if available), we recognize that code diversity manifests across multiple abstraction levels rather than as a single-dimensional implementation characteristic.

We decompose diversity generation into three abstraction levels:

$$\mathcal{L} = \{\mathcal{L}_{\text{Thought}}, \mathcal{L}_{\text{Solution}}, \mathcal{L}_{\text{Implementation}}\} \tag{1}$$

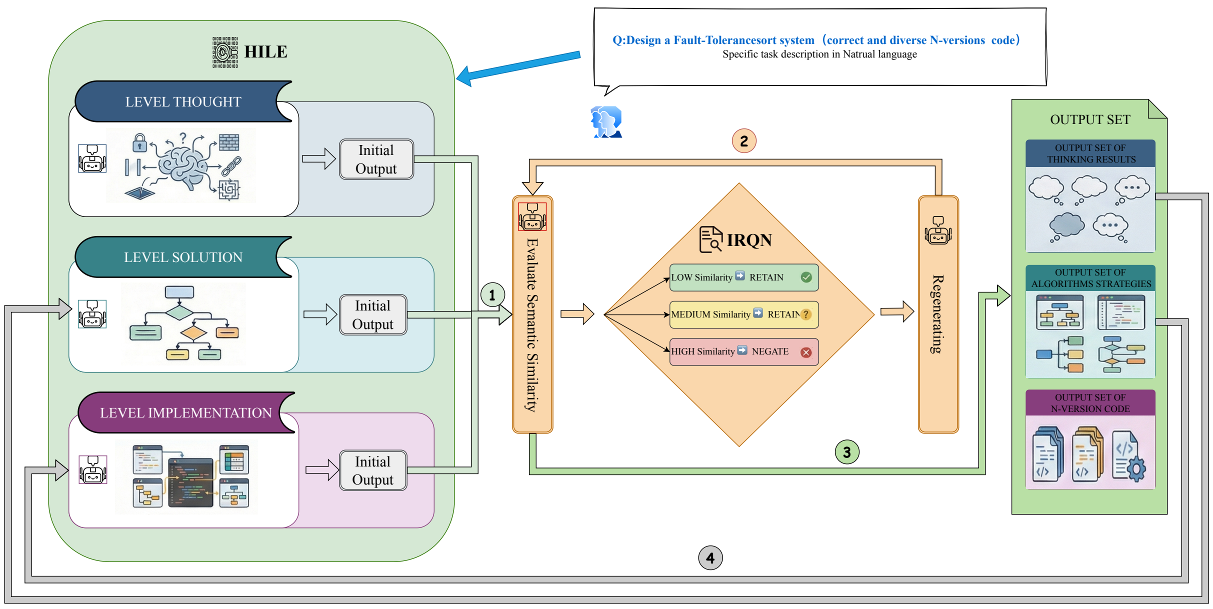

HILE
LEVEL THOUGHT
Initial Output
LEVEL SOLUTION
Initial Output
LEVEL IMPLEMENTATION
Initial Output
Q:Design a Fault-Tolerancesort system（correct and diverse N-versions code）
Specific task description in Natrual language
2
1
Evaluate Semantic Similarity
IRQN
LOW Similarity RETAIN
MEDIUM Similarity RETAIN
HIGH Similarity NEGATE
Regenerating
3
OUTPUT SET
OUTPUT SET OF THINKING RESULTS
OUTPUT SET OF ALGORITHMS STRATEGIES
OUTPUT SET OF N-VERSION CODE
4

Figure 4: DeQoG Diversity Enhancement Architecture: Synergistic Integration of HILE and IRQN across Multi-Level Abstraction Hierarchy

corresponding to solution space exploration, strategy formulation, and code implementation, respectively. This hierarchical decomposition enables systematic diversity enhancement from high-level algorithmic choices to low-level syntactic variations, achieving comprehensive diversity that encompasses both algorithmic paradigms and implementation details.

Our diversity enhancement approach is grounded in two fundamental hypotheses derived from fault-tolerance theory:

Hypothesis 1. The diversity potential at position p of task τ exhibits positive correlation with task difficulty:

$$D_{\text{potential}}(\tau, p) \propto H(\tau, p) \tag{2}$$

where $H(\tau, p)$ represents the complexity or difficulty level at position p.

Hypothesis 2. The diversity potential correlates with error probability at specific code locations:

$$D_{\text{potential}}(\tau, c, p) \propto EP(\tau, c, p) \tag{3}$$

where $EP(\tau, c, p)$ denotes the error probability of code c at position p.

These hypotheses suggest that challenging problem regions and error-prone code locations offer higher potential for meaningful diversity. Consequently, we transform the diversity requirement into two operational objectives: identifying task difficulty points and complexity characteristics, and verifying the correctness of the generated code set C to locate error-prone regions.

For difficulty identification according to hypothesis 1, we adopt a **Hierarchical Isolation and Local Expansion Method**, which leverages curated prior knowledge bases $\mathcal{K} = \{\mathcal{K}_{\text{algo}}, \mathcal{K}_{\text{impl}}, \mathcal{K}_{\text{f-t}}\}$, encompassing algorithmic patterns, implementation techniques, and fault-tolerance strategies. More importantly, HILE performs task-specific explorations $\mathcal{E} = \{\mathcal{E}_{(thg)}, \mathcal{E}_{(sol)}, \mathcal{E}_{(imp)}\}$ at multiple abstraction levels, generating contextual understanding tailored to the current problem instance.

For error identification according to hypothesis 2, we combine traditional dynamic execution testing with a novel **Iterative Retention, Questioning and Negation Method**. While dynamic execution TEST(c, T) provides direct correctness verification (detailed in Section 3.3), we recognize its limitations in comprehensiveness and efficiency. The IRQN Method $QN(\mathcal{X}, \mathcal{K}, p_{qr}, J) \rightarrow \mathcal{X}'$ systematically challenge and negate the output generated by LLM to generate new differential results while controlling costs.

The rationale for the IRQN method from a key insight: for any two distinct implementation approaches A and B, regardless of whether A contains internal errors, from the perspective of implementing B, the entirety of A can be conceptually treated as incorrect. This flood irrigation approach forces the exploration of alternative solution paths. While increasing LLM computational costs, this static diversification proves more efficient than the complex pipeline of exhaustive test case generation and execution, particularly for identifying subtle algorithmic differences that may not manifest as execution failures.

Next, we introduce the HILE and the IRQN method. And finally achieve our diversity enhancement through the combination of the two methods. As shown in Figure 4, The critical innovation lies in the synergistic design: HILE provides the structural scaffolding for diversity generation across abstraction levels, while IRQN acts as a dynamic adversary embedded within each level, actively combating LLMs' inherent bias toward high-probability solutions. This dual-mechanism approach transforms diversity from a passive outcome of random sampling into an actively engineered system property.

3.2.2. Hierarchical Isolation and Local Expansion Method (HILE)

To overcome the tendency of LLMs to prematurely converge toward common solutions during end-to-end generation, we propose a Hierarchical Isolation and Local Expansion method. This approach is based on the principle of cognitive decoupling, which forcibly partitions the code generation process vertically into three cognitive levels as defined in Equation 1, establishing isolation barriers at each level that means strictly prohibit cross-level cognitive jumps. The purpose of this design is to maximize the utilization of limited tokens for intra-level reasoning, decreasing the thinking constraints and computational waste caused by LLM's linear thought processes.

Figure 4 details our complete HILE method, which defines the exploration of the solution space for a generation task τ as a hierarchically progressive process $\Delta : \mathcal{L}_{think} \rightarrow \mathcal{L}_{sol} \rightarrow \mathcal{L}_{imp}$, where each level is dedicated to maximizing the local output diversity at that level without interference from the concrete implementations of subsequent levels.

The objective of Problem Thinking level $\mathcal{L}_{thn}$ is to establish a deep cognitive understanding of task $\tau = (P, C_{example})$ while strictly prohibiting premature entry into specific solution path exploration. We employ dynamic prompting to force the LLM to engage in multi-dimensional thinking about the problem description P and example code $C_{example}$. Let the thinking set be $T = \{t_1, t_2, \ldots, t_k\}$, where each t_i represents an independent thinking result regarding task constraints, potential pitfalls, or boundary conditions. By isolating the cognitive interference of how to solve, HILE forces the model to concentrate entirely on what to solve, thereby avoiding the oversight of hidden constraints due to mental fixation, which objectively ensures the quality of subsequent solution and final code.

Based on the thinking results T from Level $\mathcal{L}_{thn}$, $\mathcal{L}_{sol}$ focuses on the breadth expansion of algorithmic strategies while strictly prohibiting the generation of any executable code. This isolation prevents specific code syntax (such as Python-specific library functions) from reversely constraining algorithmic choices. At this level, we construct an algorithmic solution set $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$. Each strategy solution e_i consists solely of natural language descriptions and high-level pseudocode, describing an algorithmic paradigm (such as dynamic programming, greedy algorithms, divide-and-conquer, etc.). The HILE method requires the model to exhaust possible algorithmic paths at this stage, transforming different aspects of problem understanding into distinctly different algorithmic strategies through the mapping function $f : T \rightarrow \mathcal{S}$, thereby establishing the essential diversity of N-version code at the logical level.

Only after the strategy set $\mathcal{G}$ is determined does the system lift the restriction on code generation. $\mathcal{L}_{imp}$ maps each abstract strategy $s_i \in \mathcal{G}$ to a specific executable code set C_i. To further enhance diversity, we introduce micro-level expansion at the implementation layer: for the same strategy s_i, we explore different data structure choices, control flow patterns (such as recursion versus iteration), and library function calls. The final complete code set C_{all} encompasses both macro-level algorithmic differences and micro-level implementation differences, achieving comprehensive diversity coverage from cognition to implementation.

3.2.3. Iterative Retention, Questioning and Negation Method (IRQN)

If only forward generation were performed at each level of HILE, the LLM might still be constrained by the probability distribution of its training data. To address this, we embed an active Iterative Retention, Questioning and Negation(IRQN) method within each level. IRQN is not merely a filter but functions as an adversarial generation operator.

As shown in Algorithm 1, IRQN is defined as an operational function $\mathrm{RQN}(O, \mathcal{K}, p_{qn_1}, p_{qn_2}, J) \rightarrow O_{final}$, where O is the initial output set of the current level (such as thinking set T or solution set $\mathcal{E}$), $\mathcal{K}$ is the domain knowledge base, p_{qn_1} and p_{qn_2} are two custom triggering probabilities related to budget and diversity requirements, and J is the maximum number of iteration rounds. The algorithm flow is as follows: Initialize the final output set $O_{final} = \emptyset$, pending set $O_{pending} = O$, and current iteration round $iter = 0$. For each element o in $O_{pending}$, the system decides whether to initiate deep judgment based on probability p_{qn_1}. If not triggered ($rand() > p_{qn_1}$), o is directly accepted into O_{final}. If judgment is triggered, the system invokes the evaluation function Eval($o, O_{history} \cup O_{final}$). This evaluation function employs an LLM-as-a-judge methodology, where the system calculates the semantic similarity Sim between the current output and historical rounds plus currently retained results using semantic similarity methods. Different operations are executed based on the Sim value:

- **Retain:** If $Sim < \theta_{diff}$ (completely different), it is considered valid diversity, and o is added to O_{retain}. Elements in O_{retain} are then subject to probability p_{qn_2} to determine whether to proceed directly to negation. If not, O_{retain} is added to O_{final}.

- **Question:** If $\theta_{diff} \leq Sim \leq \theta_{ident}$ (partial differences but with homogeneity), it is judged as a questionable result. The system issues a questioning instruction to the LLM, pointing out existing overlaps and potential differences, requesting targeted modifications and regeneration.

- **Negate:** If $Sim > \theta_{ident}$ (nearly identical), it is judged as invalid redundancy. The system issues a negation instruction to the LLM, explicitly stating that this result already exists or is invalid, requesting regeneration.

For items that are negated or questioned, the LLM regenerates output o_{new}, which is added to the next round's pending set O_{next}. The system updates $iter \leftarrow iter + 1$. If $iter < J$ and $O_{next} \neq \emptyset$, the system recursively executes Step 2 on O_{next}. The loop continues until all results are judged as completely different, or the iteration count reaches J. Finally, the algorithm returns O_{final} (containing the original retained portion and the differentiated portion generated through J rounds of adversarial iteration) to proceed to the next level of HILE. By repeatedly applying RQN at the $\mathcal{L}_{think}$, $\mathcal{L}_{exp}$, and $\mathcal{L}_{imp}$ levels of HILE, we achieve deep mining and flood irrigation coverage of the thinking space, solution space, and implementation space, forcing the LLM to excavate those long-tail, diverse solutions that are masked by high-probability solutions in conventional reasoning under the pressure of repeated generation-being questioned-regeneration cycles.

It is worth noting that the reason for forming such nested loops is to control meaningless token and time consumption according to appropriate thresholds. Even if semantic verification confirms complete difference, it does not guarantee that the natural language solution in the solution space is truly different, and the high cost

Algorithm 1 Iterative Retention, Questioning and Negation (IRQN)

```
Input: Initial output set $O$, probabilities $p_{qn1}, p_{qn2}$,
    thresholds $\theta_{diff}, \theta_{ident}$, max iterations $J$
Output: Refined diverse output set $O_{final}$
 1: $O_{final} \leftarrow \emptyset$, $O_{pending} \leftarrow O$, $O_{history} \leftarrow \emptyset$, $iter \leftarrow 0$
 2: while $iter < J$ & $O_{pending} \neq \emptyset$ do
 3:     $O_{next} \leftarrow \emptyset$
 4:     for each $o \in O_{pending}$ do
 5:         if rand() $> p_{qn1}$ then
 6:             $O_{final} \leftarrow O_{final} \cup \{o\}$
 7:         else
 8:             $s \leftarrow$ Sim($o, O_{history} \cup O_{final}$)
 9:             if $s < \theta_{diff}$ then
10:                 if rand() $< p_{qn2}$ then
11:                     $o_{new} \leftarrow$ LLM(Pmt$_{ne}$($o, O_{history}$))
12:                     $O_{next} \leftarrow O_{next} \cup \{o_{new}\}$
13:                 else
14:                     $O_{final} \leftarrow O_{final} \cup \{o\}$
15:                 end if
16:             else if $\theta_{diff} \leq s \leq \theta_{ident}$ then
17:                 $o_{new} \leftarrow$ LLM(Pmt$_{qu}$($o, O_{history}$))
18:                 $O_{next} \leftarrow O_{next} \cup \{o_{new}\}$
19:             else
20:                 $o_{new} \leftarrow$ LLM(Pmt$_{ne}$($o, O_{history}$))
21:                 $O_{next} \leftarrow O_{next} \cup \{o_{new}\}$
22:             end if
23:         end if
24:         $O_{history} \leftarrow O_{history} \cup \{o\}$
25:     end for
26:     $O_{pending} \leftarrow O_{next}$, $iter \leftarrow iter + 1$
27: end while
        return $O_{final}$
```

of universal questioning and negation output is foreseeable. Therefore, we need to introduce the randomized negation factor p_{qn_2} to adjust execution based on budget and diversity requirements. Similarly, at lower code-level hierarchies, the capability of semantic similarity verification is trustworthy to a certain extent, so the importance of our p_{qn_2} decreases as the levels deepen according to Section 3.2.2.

3.3. Feedback-Driven Quality Assurance

After obtaining a sufficiently diverse candidate set through HILE and IRQN strategies, the next core challenge faced by N-version programming is how to ensure the correctness of each version without sacrificing diversity. Traditional generation methods often tend to overthrow and restart when encountering errors, which easily leads to code degradation back to a single standard solution. DeQoG proposes a Feedback-Driven Quality Assurance mechanism that integrates the concept of Test-Driven Development (TDD) into an automated workflow, expanding and integrating upon the DeQoG framework.

DeQoG's quality assurance is no longer a unidirectional filtering step but a closed-loop iterative system containing execution-diagnosis-repair. This system leverages the self-correction capability of LLMs, transforming test feedback into the driving force for code evolution.

Based on the dynamic prompting mechanism and structured failure feedback information collection, this process is designed as a recursive function $\text{Fix}(C_{fail}, \mathcal{F}_{info}) \rightarrow C_{pass}$. When a generated code variant $c \in C_{all}$ fails to pass the test suite T, the system does not directly discard c but enters a repair loop until the state finally transitions to $PASS$ after passing all test cases, indicating that all tests have passed and code quality meets standards, whereupon the system transitions to State 5 to output this version.

The system utilizes collected structured failure feedback information for failure analysis. Failure analysis adopts different handling strategies based on error types. As the most basic code defects, syntax errors prompt the system to extract error line numbers, error messages, and problematic code segments, directly identifying violations of syntax rules. Runtime errors indicate that code encounters exceptional situations during execution. The DeQoG will capture exception types, stack traces, and specific inputs that triggered the error, helping to locate improper boundary condition handling or resource access violations. Logic errors are more complex failure types where code can execute but produces incorrect output. The DeQoG identifies failed test cases and performs detailed comparison of expected versus actual outputs.

After systematic analysis, the system performs dynamic prompt updates, generating updated prompts $P^{(r+1)}$ enriched with feedback information, conducts repairs, generates corrected code $C^{(r+1)}$, while the system records this round's error information in historical records, then proceeds to the next step.

At this time, a key point is constraining the LLM to repair specific errors rather than abandoning the current implementation approach. This mechanism ensures that repair operations are locally minimally repaired rather than globally reconstructive. If the LLM attempts to solve errors by switching algorithms (e.g., changing a buggy quick-sort to bubble sort), the system will reject this through prompt constraints, thereby protecting the diversity painstakingly established during the HILE stage; otherwise, LLM's repairs would lead to solution convergence.

However, when the number of repair iterations exceeds threshold N_{err}, indicating this is not a simple syntax or parameter error, this reminds us of the content in Hypothesis 2 discussed in Section 3.2.1, where we believe that locations with errors have higher diversity potential. The system triggers a rollback mechanism, which also occurs when fundamental algorithmic defects are detected, returning to HILE's second level to regenerate the solutions or even to the first level to understand and think the problem again.

This process implements two termination mechanisms. One is success termination which occurs when code passes all tests before reaching the maximum iteration count N_{max}. The other is active termination which occurs when the maximum iteration count N_{max} is reached, avoiding infinite loops that consume resources, collecting existing successful repairs, and abandoning failed repairs.

4. Experimental Design

This section presents a systematic experimental framework to evaluate DeQoG's effectiveness in generating fault-tolerant N-version code. We define research questions, describe evaluation datasets and fault injection methods, establish baselines, and specify evaluation metrics.

4.1. Research Questions

We formulate four research questions to comprehensively assess DeQoG

RQ1 (Diversity and Quality): How does DeQoG perform in generating diverse and correct N-version code compared to baseline approaches?

RQ2 (Fault Tolerance): What is the reliability of DeQoG-generated N-version systems under systematic fault injection scenarios?

RQ3 (Component Contribution): What is the individual contribution of diversity enhancement and quality assurance components to overall performance?

4.2. Datasets

We evaluate DeQoG across five datasets representing varying complexity levels and programming scenarios. Table 1 summarizes their key characteristics.

Table 1: Characteristics of Four Code Generation Datasets

Dataset	Size	Avg. TC	Avg. LoC	Source
HE+	164	7.7	6.3	Hand-Written
MBPP+	974	3.0	6.7	Hand-Written
ClassEval	100	33.1	45.7	Hand-Written
LCBD	150	15.3	30.4	Online Judge
MIPD	150	8.9	64.2	Mixed Sources

4.2.1. Existing Benchmark Datasets

HumanEval+ (HE+) comprises 164 manually-crafted Python programming problems with extended test suites, designed specifically for evaluating LLM code generation capabilities.

Mostly Basic Python Programming+ (MBPP+) contains 974 diverse Python programming tasks covering fundamental programming concepts, each with task descriptions and validation test cases.

ClassEvaltargets class-level code generation with 100 Python classes (410 methods total), representing higher complexity than function-level tasks with inter-dependent method implementations.

While these benchmarks are widely adopted, they predominantly contain single-solution or constrained-solution tasks, limiting their suitability for evaluating N-version diversity. We therefore supplement them with two specialized datasets.

4.2.2. Constructed Datasets

LeetCode-based Dataset (LCBD) comprises 100 algorithmic problems from LeetCode, emphasizing computational complexity and solution strategy diversity. We deliberately selected challenging tasks (approximately 60% hard, 30% medium, 10% easy) to maximize diversity potential according to Hypothesis in Section 3.2.1.

Multi-Implementation Programming Dataset (MIPD) This dataset has been specifically curated for N-version fault-tolerant code generation tasks, comprising 100 code problems that guarantee the existence of at least three diverse implementation approaches for each task. **Data Sources:** The primary data sources for this dataset include three established benchmark datasets: MBPP+, HumanEval+, and the LeetCode problem repository, contributing 20 problems each, along with 20 additional problems designed by two experienced Python developers (each with over 5 years of professional development experience) to represent real-world application scenarios of higher complexity that are underrepresented in existing benchmarks. **Selection Criteria:** Tasks were manually screened based on the multi-implementation criterion: each problem must possess multiple viable solution methodologies employing fundamentally different algorithmic paradigms. To ensure comprehensive algorithmic coverage, we systematically selected problems spanning diverse categories including sorting, searching, dynamic programming, graph algorithms, mathematical computations, data structures, string processing, and etc. This deliberate diversity ensures MIPD captures implementation variations characteristic of real-world N-version programming and eliminates the confusion effect of single solution tasks. **Fault Injection Augmentation:** MIPD serves as the primary foundation for our fault injection experiments (detailed in Section 4.3), enabling us to rigorously validate algorithm-level diversity and its impact on fault tolerance capabilities.

4.3. Fault Injection Strategy

To rigorously evaluate fault tolerance capabilities, we design a two-tier fault injection framework.

Code-Level Faults. We inject eight high-frequency defect types identified from empirical studies on LLM code generation errors: syntax errors, conditional logic errors, constant value errors, missing statements, boundary errors, and parameter ordering errors. These task-agnostic faults enable systematic evaluation across all programming tasks.

Algorithm-Level Faults. We construct a comprehensive Common Mode Failure (CMF) library based on MIPD's task distribution, covering 20 typical algorithmic scenarios. For each scenario, we define 3-5 CMF injection schemes targeting mainstream solution algorithms. For example, for range query tasks, CMFs include: linear scan approaches violating efficiency constraints, incorrect binary search boundary handling, and unoptimized nested loops. The complete CMF taxonomy is provided in the Appendix.

4.4. Baselines

Existing approaches in LLM-based N-version programming typically rely on rudimentary prompt engineering techniques[19]. Given the nascency of research, we establish two representative baselines:

Vanilla LLM: Standard single-shot generation using low-temperature sampling (like *temperature* = 0.6, top_p = 0.90), representing raw LLM performance without post-processing.

Best-of-N (BoN): Generates N candidates with moderate temperature (0.6-0.8), selecting the best solution via LLM-as-a-judge evaluation, representing current best practices in inference-time sampling.

We instantiate each baseline with four representative LLMs: OpenAI-o1-mini and Claude-Sonnet-3.7 (leading closed-source models), DeepSeek-Coder-6.7B (specialized open-source code model), and CodeLLaMA-7B (general-purpose open-source model). This selection encompasses diverse architectural paradigms and deployment scenarios. Additionally, we carefully design the prompts for all baseline methods following established best practices from existing code generation literature and widely adopted LLM prompting techniques, including clear task descriptions, appropriate few-shot examples where beneficial, and explicit output format specifications. This rigorous prompt engineering ensures fair comparison and prevents underestimating the inherent capabilities of these baseline LLMs, allowing us to accurately assess the added value of DeQoG's diversity enhancement and quality assurance mechanisms.

4.5. Metrics

Levenshtein Similarity (LS)[]. LS Measures syntactic diversity via normalized edit distance: $\mathrm{LD}_{\mathrm{norm}}(c_i, c_j) = 1 - \frac{\mathrm{LD}(c_i,c_j)}{\max(|c_i|,|c_j|)}$, where $\mathrm{LD}(c_i, c_i)$ is the minimum number of single-character edits required to transform $|c_i|$ into $|c_j|$. The metric ranges from [0, 1], where lower values indicate higher syntactic diversity. And we compute the average pairwise LS value for N versions.

Solutions Difference Probability (SDP).[] SDP Quantifies algorithmic diversity using LLM-based evaluation of implementation strategies: $\mathrm{SDP} = 1 - \frac{\sum_{i<j} S(c_i,c_j)}{\binom{n}{2}}$, where $S(c_i, c_j) \in \{0, 1\}$ indicates strategy similarity. Higher values indicate greater methodological diversity.

Pass@k.[]: Estimates the probability that at least one correct N-version set exists in k trials. In practice, our Pass@k requires all N versions in a set to pass the complete test suite, reflecting practical N-version deployment requirements.

Failure Rate (FR). Percentage of tasks where majority voting produces correct output despite injected faults:$\mathrm{FR} = 1 - \frac{N_c}{N_t} \times 100\%$ where N_c represents the number of tasks with correct majority voting output,N_t is total number of tasks.

Additional Failure Versions Ratio (AFVR). AFVR measures system degradation by counting newly introduced failures: $\mathrm{AFVR} = \frac{N_{failpost} - N_{failpre}}{N_{vers}}$, where $N_{failpost}$ represents the number of failing versions post-injection,$N_{failpre}$ represents the number of failing versions pre-injection,N_{vers} is the number of total versions.

Token Cost: Total input and output tokens consumed.

4.6. Experimental Configuration

We conduct three complementary experimental studies: Main Experiments (RQ1-RQ2): For each of 5 datasets $N = 5$ using DeQoG and both baselines across all 4 LLMs, yielding $5 * 4 * 3 = 60$ configurations. Each generated N-version set undergoes diversity assessment (LS, SDP) and quality evaluation (Pass@k). For RQ2, we construct fault-tolerant systems from part MIPD-generated code and inject faults following two methods: (1) Code-level: five patterns from no faults to all versions faulty; (2) Algorithm-level: five CMF distribution patterns from no CMFs to all available CMFs. Ablation Study (RQ3): On MBPP+ and MIPD with $N = 3$, we compare complete DeQoG against Vanilla LLMs and two variants: DeQoG-NoDiv (removes HILE+IRQN, using simple prompt-based diversification) and DeQoG-NoQA (removes FBIR, directly outputting generated code without iterative refinement), quantifying each component's contribution to diversity and correctness.

5. Experimental Results

5.1. Answer to RQ1: Diversity and Code Quality

To comprehensively address RQ1, we conducted large-scale systematic experiments to evaluate the diversity and quality of N-version fault-tolerant code generated by DeQoG. We benchmarked DeQoG against two prevailing baselines—Vanilla LLMs (direct generation) and BoN Sampling (Best-of-N)—across four Large Language Models (LLMs) of varying scales. To ensure statistical significance, the evaluation covers five code generation datasets of varying complexity, with 100 randomly sampled tasks per dataset. For each task, we generated a multi-version code set of $N = 3, 5, \text{and} 7$ and evaluated its diversity and code quality.

Diversity. Table 2 summarizes the statistical average results for diversity generation with $N = 5$. DeQoG effectively circumvents the homogeneity bottleneck inherent in LLM generation, establishing substantial diversity advantages across all models and datasets. Statistical analysis reveals that DeQoG achieves an overall average SDP improvement of 35.65% over the Vanilla baseline and 26.06% over the Best-of-N (BoN) sampling method. This advantage is particularly pronounced in complex scenarios. For instance, on the MIPD dataset, both Vanilla and BoN approaches, regardless of sampling temperature configurations, tend to converge toward similar implementations. Notably, DeQoG-Claude-Sonnet-3.7 achieves an SDP of 0.895, indicating approximately 90% probability that

Table 2: Code Diversity Comparison Across Different Methods and Datasets ($N = 5$).

Method	Model	MBPP+		HumanEval+		ClassEval		LCBD		MIPD	
		LS↓	SDP↑	LS↓	SDP↑	LS↓	SDP↑	LS↓	SDP↑	LS↓	SDP↑
Vanilla	OpenAI-o1-mini	0.473	0.616	0.447	0.567	0.517	0.569	0.483	0.566	0.504	0.540
	Claude-Sonnet-3.7	0.300	0.714	0.369	0.725	0.364	0.678	0.420	0.656	0.430	0.662
	DeepSeek-coder-6.7B	0.627	0.455	0.619	0.476	0.664	0.457	0.665	0.416	0.694	0.389
	CodeLLaMA-7B	0.499	0.600	0.546	0.549	0.547	0.532	0.557	0.555	0.544	0.566
BoN	OpenAI-o1-mini	0.400	0.643	0.460	0.610	0.451	0.578	0.475	0.614	0.463	0.590
	Claude-Sonnet-3.7	0.330	0.714	0.342	0.700	0.395	0.705	0.360	0.679	0.297	0.762
	DeepSeek-coder-6.7B	0.625	0.516	0.595	0.479	0.597	0.468	0.630	0.511	0.601	0.500
	CodeLLaMA-7B	0.462	0.573	0.508	0.585	0.524	0.596	0.462	0.566	0.431	0.628
DeQoG	OpenAI-o1-mini	0.381	0.663	0.325	0.679	0.348	0.732	0.287	0.752	0.171	0.843
	Claude-Sonnet-3.7	**0.238**	**0.792**	**0.257**	**0.808**	**0.174**	**0.791**	**0.150**	**0.836**	**0.096**	**0.895**
	DeepSeek-coder-6.7B	0.441	0.666	0.398	0.655	0.408	0.624	0.353	0.691	0.214	0.795
	CodeLLaMA-7B	0.265	0.719	0.301	0.742	0.253	0.799	0.236	0.782	0.307	0.740

Note: **LS**: Levenshtein Similarity (lower is better); **SDP**: Solution Difference Probability (higher is better).

any two randomly selected code variants employ fundamentally different implementation strategies. Meanwhile, DeQoG-DeepSeek-Coder-6.7B achieves 0.795 SDP, demonstrating a remarkable 104.37% improvement over Vanilla-DeepSeek-Coder-6.7B (0.389) on MIPD dataset, whereas the improvement on MBPP+ reaches only 46.4%—a disparity attributable to the inherent diversity ceiling imposed by each dataset's solution space. These results demonstrate that DeQoG transcends superficial syntactic variation by structurally enforcing exploration of the long-tail regions of the solution space. The robust negative correlation (-0.94) between LS and SDP further substantiates this finding: as algorithmic differentiation increases, LS decreases by 44.9% and 40.48% overall, respectively, confirming that DeQoG's diversity gains—anchored in the HILE and IRQN methodologies—stem from deep logical exploration rather than superficial syntactic restructuring.

Diversity Result #1:

DeQoG achieves **35.65%** and **26.06%** improvements in SDP over Vanilla and BoN baselines, respectively, while reducing LS by **44.9%** and **40.48%**, demonstrating that the DeQOG can produce not only diverse code texts but also more methodological differentiation.

While the overall improvements validate DeQoG's effectiveness, a deeper analysis across models of varying parameter scales, as shown in Figure 5 reveals a surprising and theoretically significant finding: DeQoG's diversity enhancement more benefits smaller language models. Experimental data demonstrate that DeQoG achieves overall average improvements of 46.78% and 33.57% over Vanilla and BoN baselines, respectively, for small-parameter models (SPMs, here refers to CodeLLaMA and DeepSeek-Coder), compared to 24.67% and 18.56% for large-parameter models (LPMs, here refers to Claude-Sonnet-3.7 and OpenAi-o1-mini). More remarkably, DeQoG-CodeLLaMA-8B attains an SDP of 0.74 on MIPD, surpassing Vanilla-OpenAi-o1-mini (0.54) by 37.04% and the SDP value of the SPM group under the DeQoG method exceeded that of the Vanilla LPM group by 14.62%. This indicates that the bottleneck in diverse code generation often resides not in the model's raw reasoning capacity but in the absence of structured guidance.

Why do SPMs benefit more from DeQoG? We propose two complementary explanations: (1) SOTA LPMs already possess substantial baseline diversity exploration capabilities (e.g., Claude-Sonnet-3.7's Vanilla SDP of 0.662 on MIPD). DeQoG's gains, while significant (35.2% improvement to 0.895), are constrained by the inherent diversity ceiling of the task. (2) SPMs with limited self-guided exploration benefit dramatically from external scaffolding. DeQoG's hierarchical isolation and adversarial questioning provide the structured reasoning path that SPMs lack internally, effectively compensating for their capacity limitations. The DeepSeek-coder-6.7B case exemplifies this: starting from a low baseline (0.389), DeQoG amplifies diversity

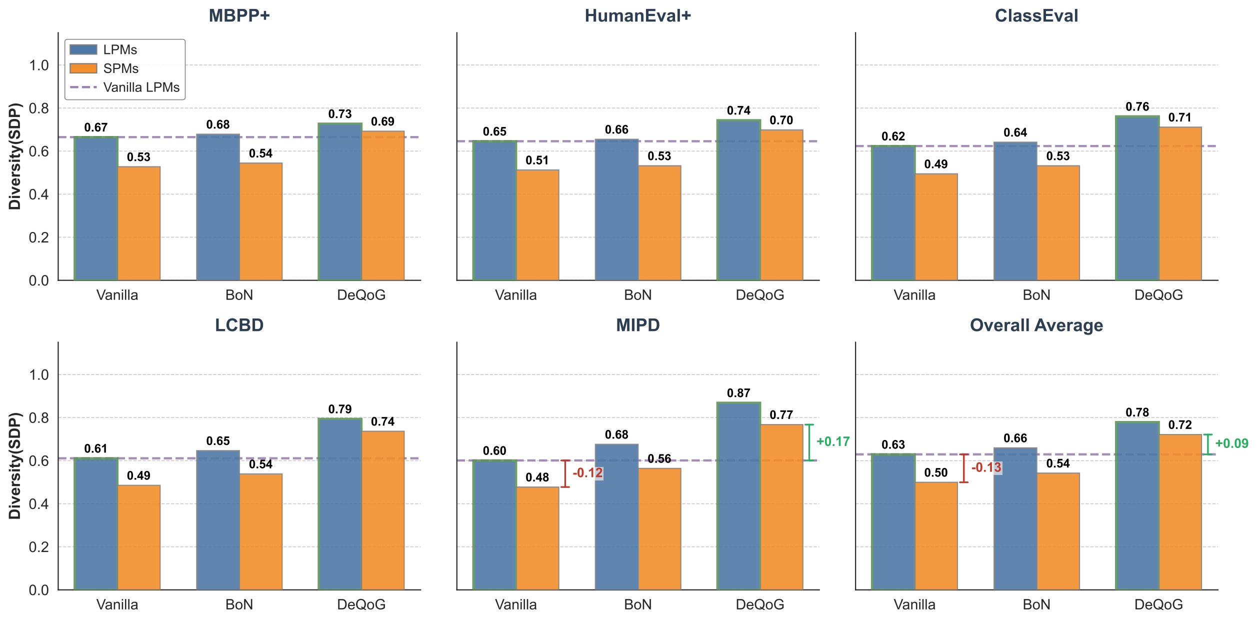


Figure 5: **Evaluation of DeQoG's effectiveness in bridging the diversity gap between Small Parameter Models (SPMs) and Large Parameter Models (LPMs).** The bar chart compares the Solution Difference Probability (SDP) across five datasets and their overall average. The dashed purple line represents the baseline performance of Vanilla LPMs. **Red values** indicate the initial performance gap of Vanilla SPMs, while **green values** highlight where DeQoG-enhanced SPMs surpass the Vanilla LPM baseline, particularly in complex tasks (MIPD) and overall average performance.

by 104.37% to reach 0.795—approaching the performance of much LPMs. This finding demonstrates that locally deployable SPMs can leverage DeQoG to generate high-quality N-version code cost-effectively, significantly advancing fault-tolerant software engineering practices.

DeQoG achieves overall average SDP improvements of 21.19% and 17.15% over Vanilla and BoN baselines, respectively, on the MBPP+ dataset, while improvements on LCBD and MIPD reach 41.83% and 29.75%, and 56.60% and 34.29%, respectively. This phenomenon applies not only to DeQoG's diversity strategies but also to BoN sampling approaches. This is because simpler datasets (e.g., MBPP+) impose lower diversity ceilings due to constrained solution spaces, which also proves the hypothesis we proposed in Section 3.2. However, on complex datasets such as LCBD and MIPD, DeQoG fully exploits potential heterogeneous solution and implementation paths, achieving substantially greater diversity improvements than on simpler datasets. This demonstrates the framework's distinct advantage in handling complex, real-world programming scenarios.

Diversity Result #2:

DeQoG exhibits **46.78%** and **33.57%** diversity improvements for SPMs, respectively, compared to **24.67%** and **18.56%** for LPMs. More remarkably, DeQoG-enhanced SPMs achieve a **14.62%** higher diversity performance than vanilla LPMs (0.721 vs 0.629), with the gap reaching **27.70%** on complex tasks (MIPD), demonstrating the effective ability of DeQoG to enhance diversity in LLMs.

Code Quality. We systematically evaluated each method under $k \in \{1, 3, 5\}$ configurations to assess the overall quality(using Pass@k) of N-version code sets. A key distinction exists between our Pass@k metric and that of traditional code generation domains: in this study, Pass@k is defined as the probability that, across k independent generation trials, at least one trial produces an N-version set where all N versions pass the complete test suite. This definition better aligns with the practical deployment requirements of N-version fault-tolerant systems—demanding collective usability of the version set rather than correctness of individual versions alone.

Figure 6 presents comprehensive code quality assessment results under the $N = 5$ configuration. De-

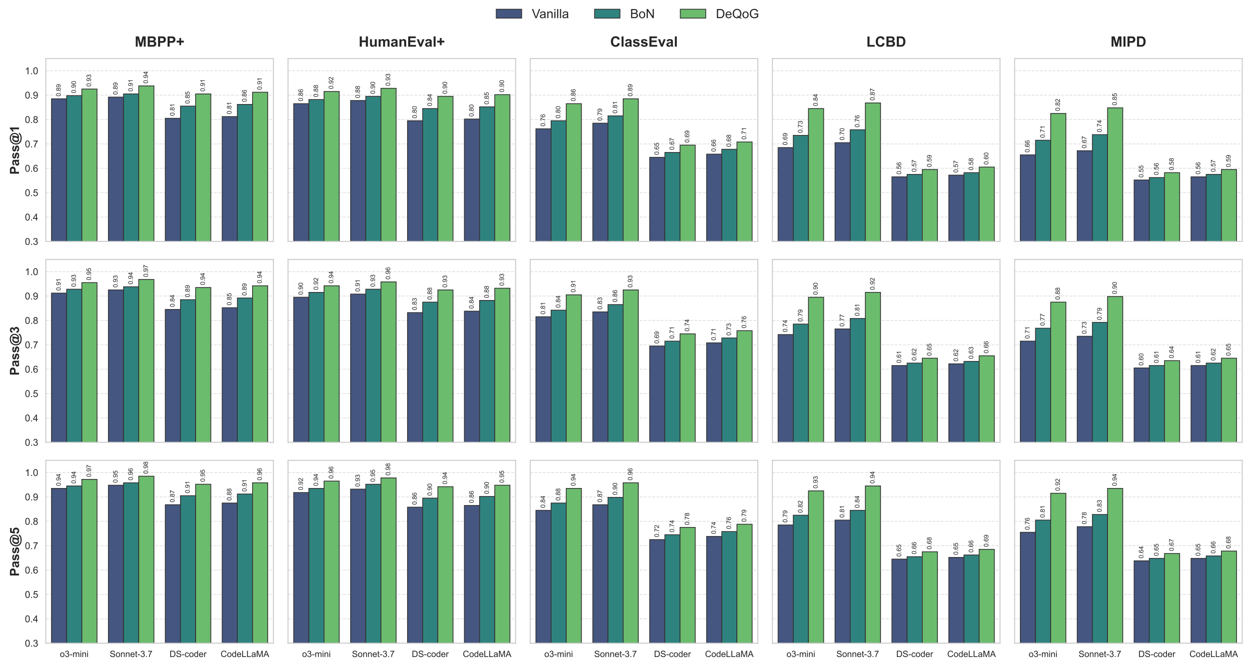


Figure 6: Pass@k Performance Comparison for Vanilla, BoN, and DeQoG Across Five Benchmarks

QoG demonstrates significant and consistent improvements in code correctness: across all 60 experimental configurations (5 datasets × 4 models × 3 k values), DeQoG's performance metrics uniformly surpass both Vanilla and BoN baselines. Compared to Vanilla and BoN baselines, DeQoG achieves significant improvements of 18.7% and 12.4% respectively in overall average Pass@1; Pass@3 improvements of 10.8% and 5.2%; and Pass@5 improvements of 8.1% and 4.5%.

More importantly, these quality improvements are achieved while simultaneously maintaining high diversity. Synthesizing the diversity metrics from Table 2 with the quality data in this section, taking DeQoG-Sonnet-3.7 as an example: this configuration achieves the highest algorithmic diversity (SDP=0.895) while also attaining the highest code correctness (Pass@1=0.86). Pearson correlation analysis reveals that among code sets generated by DeQoG, SDP and Pass@1 exhibit significant positive correlation ($\rho = 0.534$, $p < 0.01$); in contrast, Vanilla and BoN methods show weak negative correlation ($\rho = -0.14$, $p > 0.05$) and weak positive correlation ($\rho = 0.08$, $p > 0.05$) respectively, neither reaching statistical significance. These findings validate DeQoG's core design hypothesis: with appropriate systematic methodological support, diversity enhancement and quality assurance can form positive synergy rather than zero-sum trade-offs, jointly improving the overall efficacy of N-version fault-tolerant systems.

Quality Result #1:

DeQoG achieves 100% performance advantage coverage across all 60 test configurations. Compared to the Vanilla baseline, overall average improvements are: Pass@1 by **15.9%**, Pass@3 by **10.8%**, and Pass@5 by **8.1%**; compared to the BoN baseline: Pass@1 by **7.0%**, Pass@3 by **5.2%**, and Pass@5 by **4.5%**.

Further analysis reveals a noteworthy phenomenon: in contrast to the "SPMs benefit more significantly" pattern observed in the diversity analysis part, in the code quality dimension, LPMs exhibit higher improvement magnitudes on complex tasks. This contrasting pattern reveals the differential dependencies of DeQoG components on model capabilities—diversity enhancement (HILE+IRQN) relies more on exploration capability, while quality assurance (FBIR) depends more on reasoning capability. This component-level distinction will be deeply analyzed in the ablation study of Section 5.3. The following focuses on the specific impact of this phenomenon on DeQoG's code quality and its direct causes.

From the Pass@1 row subplots in Figure 6, large parameter models demonstrate higher absolute improvement magnitudes on complex datasets. Taking the MIPD dataset as an example, Claude-Sonnet-3.7 improves from the Vanilla baseline of Pass@1=0.67 to DeQoG's 0.86, achieving a relative improvement magnitude of 28.4%. Across all complex datasets (ClassEval, LCBD, MIPD), the LPM group's average improvement magnitude is 24.9%, while the SPM group's is 20.7%, with LPMs leading by 4.2 percentage points.

Table 3: FBIR Repair Efficiency of Several Typical Error in Different Models.

Error Type	Avg. Iterations		Success Rate		Rollback Rate	
	LPM	SPM	LPM	SPM	LPM	SPM
Syntax Error	1.1 ± 0.3	1.4 ± 0.5	96.7%	91.2%	1.2%	3.8%
Runtime Error	1.6 ± 0.6	2.2 ± 1.0	92.8%	82.4%	3.1%	8.6%
Logic Error	2.3 ± 0.9	3.4 ± 1.6	88.3%	71.4%	5.2%	16.8%
Boundary Error	1.8 ± 0.7	2.6 ± 1.2	90.5%	79.1%	4.3%	11.7%
Total	**1.7 ± 0.7**	**2.4 ± 1.1**	**92.1%**	**81.0%**	**3.5%**	**10.2%**

To reveal the underlying mechanism of performance differences, this study conducted systematic analysis of LLM generation logs during the FBIR phase, collecting statistics on repair iteration counts, success rates, and rollback frequencies for different error types. Table 3 presents detailed stratified statistical results. The data reveals specific manifestations of model capability gaps in the error repair process: in the most challenging logical error repair tasks, SPMs' success rate (71.4%) is significantly lower than LPMs' (88.3%), with a gap of 16.9 percentage points; simultaneously, SPMs' rollback rate (16.8%) is 3.2 times that of LPMs (5.2%).

This comparison indicates that the FBIR mechanism has high dependency on model reasoning capabilities: understanding semantic information from test failures, inferring causal roots of errors, and executing localized precise repairs all require strong code comprehension and logical reasoning abilities—precisely where small parameter models face capability bottlenecks. Conversely, this finding also confirms the rationality of DeQoG's design: through structured test feedback loops, FBIR can fully activate and leverage LLMs' existing reasoning capabilities to achieve systematic code quality assurance.

Quality Result #2:

In the code quality dimension, LPMs' improvement magnitude (**24.9%**) in complex N-version fault-tolerant code generation tasks significantly exceeds SPMs' (**20.7%**), with a gap of **4.2** percentage points.

From the above four experimental results on diversity and code quality, we can derive our answer to RQ1.

Summary for RQ1

DeQoG demonstrates performance advantage across all 60 test configurations in 5 benchmarks, achieving **35.72%** improvement in algorithmic diversity (SDP) and **18.7%** in code correctness (Pass@1) compared to baselines. In addition DeQoG establishes significant positive correlation between diversity and quality (ρ = 0.534, p < 0.01), validating that systematic software engineering frameworks can collaborate with diversity and quality to improve the fault tolerance of N-version code.

5.2. Answer to RQ2: Reliability

To comprehensively evaluate the fault tolerance of DeQoG-generated N-version code, we conducted two complementary fault injection experiments: code-level (syntactic/semantic defects) and algorithm-level (methodological errors). The framework operates on 100 MIPD tasks, constructing majority-voting-based fault-tolerant systems. Our experiments follow three core principles: (1) Progressive injection: systematically increasing fault density from fault-free baselines to examine robustness degradation; (2) Hierarchical faults: distinguishing implementation-layer errors from algorithmic design flaws to validate diversity's value at different abstraction levels; (3) Dataset-adaptive selection: prioritizing fault types effectively injectable into sample code for ecological validity. The fault-tolerant system employs standard majority voting: for each test case, the system selects the most frequently occurring output from N versions as the final result, with failure declared when more than half of the same majority versions do not exist or when they exist but the output is wrong.

Code-Level Fault Injection Experiments. To rigorously evaluate the fault tolerance capabilities of N-version systems generated by different methods against syntactic and semantic defects, we constructed a comprehensive fault pattern library based on empirical studies of LLM code generation errors [44]. This library encompasses eight categories of high-frequency syntactic and semantic defects, including index out-of-bounds errors, type mismatches, improper boundary condition handling, and other prevalent failure modes. We manually adapted these fault patterns to the sample code collection to ensure effective injection mechanisms, then systematically injected valid fault combinations into the N-version fault-tolerant systems according to varying fault density patterns. We designed five fault injection patterns (Pat-CL 0 through Pat-CL 4) representing pro-

Table 4: Comparison of System Failure Rate (FR) under Code-Level Fault Injection ($N = 5$, MIPD Dataset). Lower is better.

Pattern	**Vanilla**				**BoN**				**DeQoG (Ours)**			
	o1-mini	Sonnet	DS-Coder	LLaMA	o1-mini	Sonnet	DS-Coder	LLaMA	o1-mini	Sonnet	DS-Coder	LLaMA
Pat-CL 0	0.34	0.33	0.45	0.44	0.29	0.26	0.44	0.43	**0.18**	**0.15**	**0.42**	**0.41**
Pat-CL 1	0.41	0.40	0.53	0.52	0.35	0.32	0.51	0.50	**0.25**	**0.22**	**0.49**	**0.48**
Pat-CL 2	0.52	0.50	0.64	0.62	0.45	0.41	0.61	0.59	**0.34**	**0.30**	**0.58**	**0.56**
Pat-CL 3	0.87	0.85	0.93	0.91	0.84	0.81	0.91	0.89	**0.76**	**0.73**	**0.89**	**0.87**
Pat-CL 4	0.98	0.97	0.99	0.99	0.97	0.96	0.99	0.98	**0.94**	**0.93**	**0.98**	**0.95**
Average	0.55	0.54	0.65	0.64	0.52	0.47	0.62	0.60	**0.38**	**0.34**	**0.57**	**0.56**

Table 5: Fault Tolerance Performance Comparison under Code-Level Fault Injection (Averaged across models, $N = 5$, MIPD Dataset).

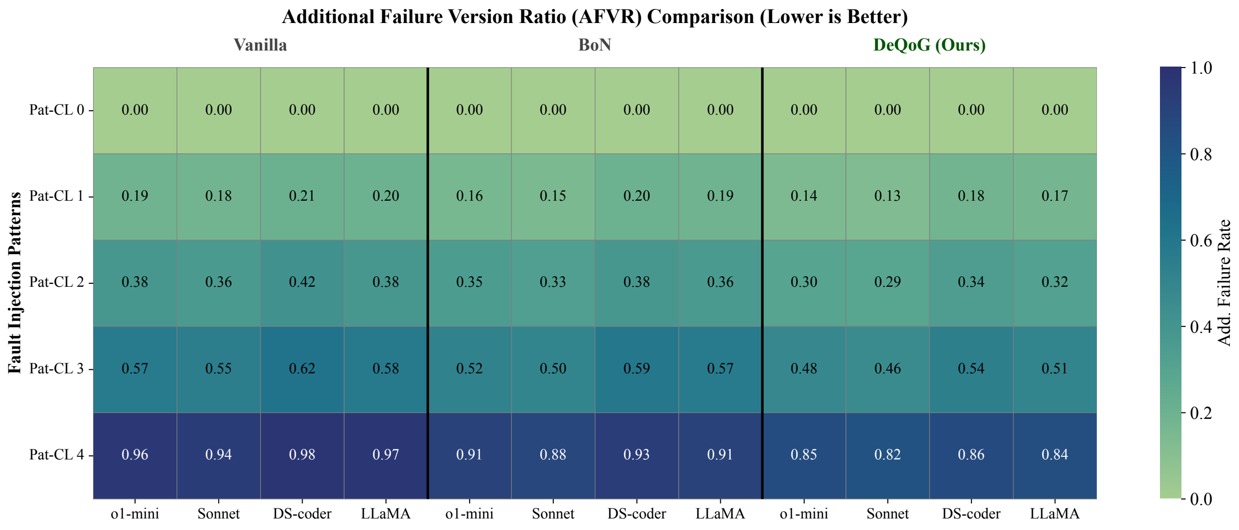
Pattern	**Vanilla**		**BoN**		**DeQoG**		**Improvement of Vanilla**		**Improvement of BoN**	
	FR ↓	AFVR ↓	FR ↓	AFVR ↓	**FR ↓**	**AFVR ↓**	ΔFR	ΔAFVR	ΔFR	ΔAFVR
Pat-CL 0	0.390	0.000	0.355	0.000	**0.290**	**0.000**	25.64%	-	18.31%	-
Pat-CL 1	0.465	0.195	0.420	0.175	**0.360**	**0.155**	22.58%	20.51%	14.29%	11.43%
Pat-CL 2	0.565	0.385	0.515	0.354	**0.445**	**0.312**	21.24%	18.96%	13.59%	11.86%
Pat-CL 3	0.890	0.578	0.863	0.545	**0.813**	**0.497**	8.65%	14.01%	5.79%	8.81%
Pat-CL 4	0.980	0.963	0.975	0.907	**0.950**	**0.843**	3.06%	12.46%	2.56%	7.06%
Overall	0.658	0.424	0.626	0.396	**0.572**	**0.361**	13.13%	14.80%	8.63%	8.78%

gressively severe scenarios: from fault-free configurations to cases where all versions contain defects.

1. **Pat-CL 0:** No faults in any version
2. **Pat-CL 1:** Exactly one version contains a fault
3. **Pat-CL 2:** $\lfloor (N-1)/2 \rfloor$ versions contain faults
4. **Pat-CL 3:** $\lfloor (N+1)/2 \rfloor$ versions contain faults
5. **Pat-CL 4:** All N versions contain faults

Table 4 presents the system Failure Rate (FR) across all fault injection patterns, where lower values indicate superior fault tolerance. DeQoG demonstrates consistent and substantial advantages across all experimental configurations. Under the most challenging Pat-CL 4 scenario (all versions containing faults), DeQoG-Sonnet achieves an FR of 0.93, significantly outperforming Vanilla-Sonnet (0.97) and BoN-Sonnet (0.96). This means a 3.06% relative reduce in system failures rate. The cross-model averaged results in Table 5 reveal DeQoG's systematic superiority. Compared to Vanilla and BoN baselines, DeQoG achieves overall FR reductions of 13.13% and 8.63%, respectively. Notably, the improvement magnitude exhibits a clear pattern across fault injection severity levels. Under minimal fault conditions (Pat-CL 0, single-version failures), DeQoG reduces FR by 25.64% relative to Vanilla. As fault density increases to moderate levels (Pat-CL 2, approximately 40% of versions failing), the improvement stabilizes at 21.24%. DeQoG's relative improvement is constrained by the inherent threshold characteristics of majority voting. With N=5 versions, Pat-CL 2 injects faults into 2 versions while preserving 3 correct ones—sufficient for majority consensus. However, Pat-CL 3 crosses the critical threshold by corrupting 3 versions, leaving only 2 correct—insufficient for majority voting to produce correct outputs. This mathematical constraint explains the precipitous FR increase observed across all methods and why DeQoG's relative improvement over Vanilla declines from over 20% to 8.65%. Despite this fundamental limitation, under extreme conditions (Pat-CL 4), the improvement narrows to 3.06% as the voting mechanism's effectiveness becomes constrained by overwhelming fault prevalence.

Beyond overall system failures, we measured the Additional Failure Version Ratio (AFVR)—quantifying how many additional versions fail post-injection relative to pre-injection baselines. This metric captures system degradation: lower AFVR indicates that the generated N-version set maintains its redundancy capacity even under fault injection. Figure 7 visualizes the AFVR distribution across models and patterns through a heatmap, where DeQoG (rightmost columns) consistently exhibits lighter colors (lower values) compared to Vanilla and BoN counterparts. Table 5 quantifies this advantage: DeQoG achieves an overall AFVR of 0.361, representing 14.80% and 8.78% improvements over Vanilla (0.424) and BoN (0.396), respectively. The dual improvements in both FR and AFVR validate that DeQoG can generate diverse syntactic and semantic variants. For instance, under Pat-CL 1 conditions,



Fault Injection Patterns	Vanilla o1-mini	Vanilla Sonnet	Vanilla DS-coder	Vanilla LLaMA	BoN o1-mini	BoN Sonnet	BoN DS-coder	BoN LLaMA	DeQoG (Ours) o1-mini	DeQoG (Ours) Sonnet	DeQoG (Ours) DS-coder	DeQoG (Ours) LLaMA
Pat-CL 0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Pat-CL 1	0.19	0.18	0.21	0.20	0.16	0.15	0.20	0.19	0.14	0.13	0.18	0.17
Pat-CL 2	0.38	0.36	0.42	0.38	0.35	0.33	0.38	0.36	0.30	0.29	0.34	0.32
Pat-CL 3	0.57	0.55	0.62	0.58	0.52	0.50	0.59	0.57	0.48	0.46	0.54	0.51
Pat-CL 4	0.96	0.94	0.98	0.97	0.91	0.88	0.93	0.91	0.85	0.82	0.86	0.84

Figure 7: CL AFVR Across Models

while Vanilla systems experience 19.5% additional failures on average, DeQoG limits this degradation to only 15.5%—a 20.51% relative improvement that directly translates to enhanced system survivability.

Code-Level Result:

DeQoG-generated N-version systems demonstrate superior fault tolerance across all code-level fault injection scenarios, achieving **13.13%** and **8.63%** reductions in FR compared to Vanilla and BoN baselines, respectively. The framework simultaneously reduces AFVR by **14.80%** and **8.78%**, confirming that HILE and IRQN methodologies produce more diverse syntactic and semantic variants.

Algorithm-Level Fault Injection Experiments. Algorithm-level faults represent deeper methodological defects that differ fundamentally from code-level errors. Unlike syntactic or semantic bugs, algorithmic faults exhibit strong task dependency—their correctness must be judged within specific task requirements and constraints, such as time complexity violations or incorrect algorithmic paradigm selection. These faults cannot be eliminated through local repairs and can only be tolerated through genuine algorithmic diversity. To validate the generalizability of DeQoG's fault tolerance capabilities, we constructed a comprehensive predefined Common Mode Failure (CMF) library based on MIPD's task distribution (Table in Appendix). The library covers 20 typical scenarios across 5 major task categories, with 2 CMF injection schemes designed for each scenario targeting both sample code and mainstream solution algorithms collected manually. We employed three distribution patterns: Pat-AL 0 (no CMF in any version), Pat-AL 1 (all versions contain exactly 1 CMF), and Pat-AL 2 (all versions contain all 2 CMFs).

Table 6 presents the system FR under algorithm-level fault injection. DeQoG demonstrates remarkably superior performance across all patterns. Under the baseline Pat-AL 0 scenario, DeQoG-Sonnet achieves an FR of 0.15, substantially outperforming Vanilla-Sonnet (0.33) and BoN-Sonnet (0.26)—representing 54.5% and 42.3% relative improvements, respectively. This advantage persists as fault severity escalates: under Pat-AL 2 where all versions contain all CMFs, DeQoG-Sonnet maintains an FR of 0.26 compared to Vanilla-Sonnet's 0.51, demonstrating 49.0% relative improvement.

The cross-model averaged results in Table 7 reveal DeQoG's systematic superiority at the algorithmic level. Compared to Vanilla and BoN baselines, DeQoG achieves overall FR reductions of 30.62% and 23.76%, respectively—substantially exceeding the code-level improvements (13.13% and 8.63%). This amplified advantage validates a critical hypothesis: algorithmic diversity, rather than superficial syntactic variations, constitutes the primary driver of fault tolerance in N-version systems.

Figure 8 visualizes the AFVR distribution through a heatmap, where DeQoG (rightmost columns) consistently exhibits the lightest colors across all models and patterns. Table 7 quantifies this advantage: DeQoG achieves an overall AFVR of 0.185, representing

Table 6: Comparison of System Failure Rate (FR) under Code-Level Fault Injection (N = 5, MIPD Dataset). Lower is better.

Pattern	**Vanilla**				**BoN**				**DeQoG (Ours)**			
	o1-mini	Sonnet	DS-Coder	LLaMA	o1-mini	Sonnet	DS-Coder	LLaMA	o1-mini	Sonnet	DS-Coder	LLaMA
Pat-AL 0	0.34	0.33	0.45	0.44	0.29	0.26	0.44	0.43	**0.18**	**0.15**	**0.42**	**0.41**
Pat-AL 1	0.37	0.36	0.48	0.47	0.32	0.30	0.47	0.45	**0.20**	**0.17**	**0.44**	**0.43**
Pat-AL 2	0.53	0.51	0.67	0.65	0.47	0.44	0.63	0.60	**0.29**	**0.26**	**0.48**	**0.46**
Average	0.41	0.40	0.53	0.52	0.36	0.33	0.51	0.49	**0.22**	**0.19**	**0.45**	**0.43**

Table 7: Fault Tolerance Performance Comparison under Code-Level Fault Injection (Averaged across models, N = 5, MIPD Dataset).

Pattern	**Vanilla**		**BoN**		**DeQoG**		**Improvement of Vanilla**		**Improvement of BoN**	
	FR ↓	AFVR ↓	FR ↓	AFVR ↓	**FR ↓**	**AFVR ↓**	ΔFR	ΔAFVR	ΔFR	ΔAFVR
Pat-AL 0	0.390	0.000	0.355	0.000	**0.290**	0.000	25.64%	-	18.31%	-
Pat-AL 1	0.420	0.255	0.385	0.225	**0.310**	**0.170**	26.19%	33.33%	19.48%	24.44%
Pat-AL 2	0.590	0.578	0.535	0.538	**0.373**	**0.385**	36.78%	33.39%	30.28%	28.44%
Overall	**0.467**	**0.278**	**0.425**	**0.254**	**0.324**	**0.185**	**30.62%**	**33.45%**	**23.76%**	**27.17%**

33.45% and 27.17% improvements over Vanilla (0.278) and BoN (0.254), respectively. Notably, under Pat-AL 1 conditions, DeQoG's AFVR of 0.170 demonstrates 33.33% improvement over Vanilla's 0.255, confirming that HILE and IRQN methodologies successfully generate implementations with genuinely independent algorithmic logic rather than mere syntactic variants.

A distinctive pattern emerges: DeQoG's relative improvement increases with fault severity. Under Pat-AL 0, FR improvement is 25.64% over Vanilla; this advantage expands to 26.19% under Pat-AL 1 and reaches 36.78% under Pat-AL 2. This counter-intuitive trend—contrasting sharply with code-level results where improvement diminished under extreme faults—demonstrates that algorithmic diversity becomes increasingly valuable as methodological faults accumulate. When all versions contain all CMFs, only genuinely diverse algorithmic approaches can maintain system functionality through majority voting.

Algorithm-Level Result:

DeQoG-generated N-version systems exhibit superior fault tolerance under algorithm-level fault injection, achieving **30.62%** and **23.76%** reductions in system Failure Rate compared to Vanilla and BoN baselines—more than double the improvements observed under code-level faults. The framework simultaneously reduces AFVR by **33.45%** and **27.17%**.

With improvement magnitude increasing (not decreasing) as fault severity escalates, the results confirm that HILE and IRQN produce genuine algorithmic diversity with independent failure modes—the cornerstone of reliable fault-tolerant systems.

Summary for RQ2

DeQoG-generated N-version systems demonstrate superior fault tolerance across both hierarchies. At the code level, DeQoG achieves 13.13% and 8.63% reductions in FR compared to Vanilla and BoN baselines, with corresponding AFVR improvements of 14.80% and 8.78%. At the algorithm level, these advantages amplify dramatically: DeQoG achieves 30.62% and 23.76% FR reductions—more than double the code-level improvements—with AFVR reductions of 33.45% and 27.17%.

5.3. Answer to RQ3: Component Contribution

To quantify the independent contributions and synergistic effects of each component, we designed three system variants: (1) **DeQoG-Full**, the complete framework; (2) **DeQoG-NoDiv**, removing HILE and IRQN while adopting simple end-to-end generation and retaining FBIR; (3) **DeQoG-NoQA**, removing FBIR's iterative repair mechanism while preserving HILE and IRQN. Experiments were conducted on the MIPD and MBPP+ datasets (N=3, 100 tasks each), using Claude-Sonnet-3.7 and DeepSeek-Coder-6.7B as representatives of large parameter models (LPMs) and small parameter models (SPMs), respectively, with comparisons against the Vanilla baseline.

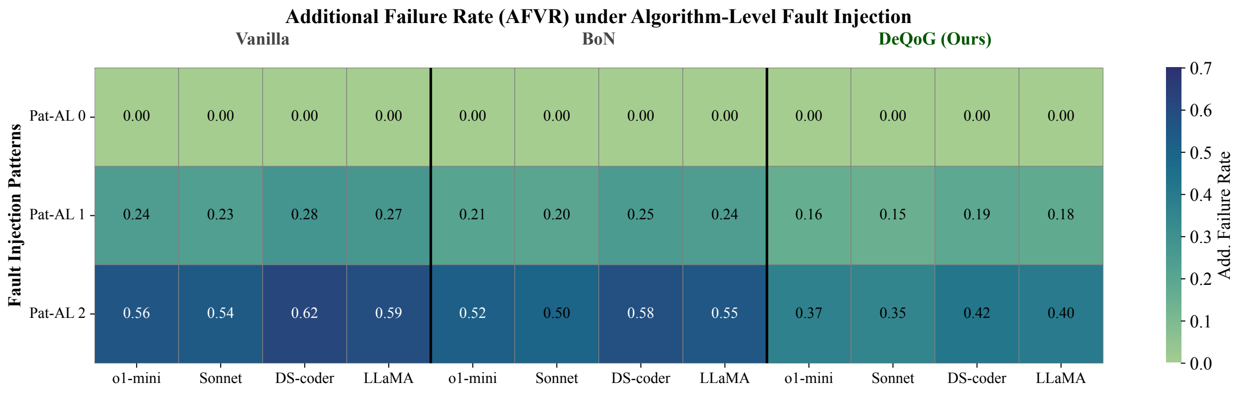


Figure 8: AL AFVR Across Models

Figure 9 presents the diversity (SDP) and correctness (Pass@1) comparison results across both datasets. DeQoG-Full achieves optimal performance under all configurations, with component contributions exhibiting differentiated patterns according to task complexity. On the MIPD dataset, the complete DeQoG framework significantly narrows the performance gap between SPMs (orange bars) and LPMs (blue bars); on the MBPP+ dataset, despite the generally high baseline levels across all variants, the Full configuration maintains its leading advantage.

Comparing the NoDiv and Full configurations, HILE+IRQN significantly enhances diversity across both datasets, though improvement magnitudes vary with task complexity. On the MIPD dataset, Sonnet-3.7's SDP improves from 0.689 to 0.895 (+29.9%), and DeepSeek-Coder from 0.425 to 0.795 (+87.1%); on the MBPP+ dataset, corresponding improvements are from 0.726 to 0.792 (+9.1%) and from 0.489 to 0.666 (+36.2%), respectively. SPMs exhibit average improvement magnitudes 3.4 times of those of LPMs across both datasets (MIPD: 87.1% vs. 29.9%; MBPP+: 36.2% vs. 9.1%), validating the result from Section 5.1 that smaller models rely more heavily on external structured guidance to explore solution spaces.

Comparing the NoQA and Full configurations, FBIR drives quality improvements across both datasets, with more pronounced effects on SPMs. On the MIPD dataset, Sonnet-3.7's Pass@1 improves from 0.74 to 0.86 (+16.2%), and DeepSeek-Coder from 0.62 to 0.78 (+25.8%); on the MBPP+ dataset, corresponding improvements are from 0.90 to 0.93 (+3.3%) and from 0.83 to 0.90 (+8.4%), respectively. The higher baseline quality in MBPP+ (Base Pass@1 reaching 0.89 and 0.81) constrains the improvement headroom, reflecting the influence of task difficulty on quality enhancement potential. Nevertheless, FBIR's relative advantage for SPMs remains consistent across both datasets, averaging 1.6× that of LPMs (MIPD: 25.8% vs. 16.2%; MBPP+: 8.4% vs. 3.3%), corroborating the repair efficiency data presented in Table 3 of Section 5.1.

The overall patterns in Figure 9 reveal positive synergistic effects—DeQoG-Full's aggregate improvements exceed the arithmetic sum of individual component contributions. On the MIPD dataset, Sonnet achieves SDP improvement of 35.2% and Pass@1 improvement of 28.4%; on the MBPP+ dataset, corresponding improvements are 10.9% and 4.5%. This demonstrates that the diverse algorithmic approaches generated by HILE provide more robust starting points for FBIR repairs, while FBIR's test feedback effectively validates and filters IRQN's adversarially generated results. This finding highlights the application value of DeQoG's systematic framework in scenarios characterized by high complexity, high diversity requirements, and resource constraints.

Summary for RQ3

On complex tasks (MIPD), HILE+IRQN achieves diversity improvements of 29.9%–87.1%, while FBIR achieves quality improvements of 16.2%–25.8%; on simpler tasks (MBPP+), corresponding improvements are 9.1%–36.2% and 3.3%–8.4%. Concurrently, SPMs benefit significantly more from both components than LPMs.

6. Threats to Validity

Internal validity threats include data contamination and prompt engineering bias. Since LLMs are trained

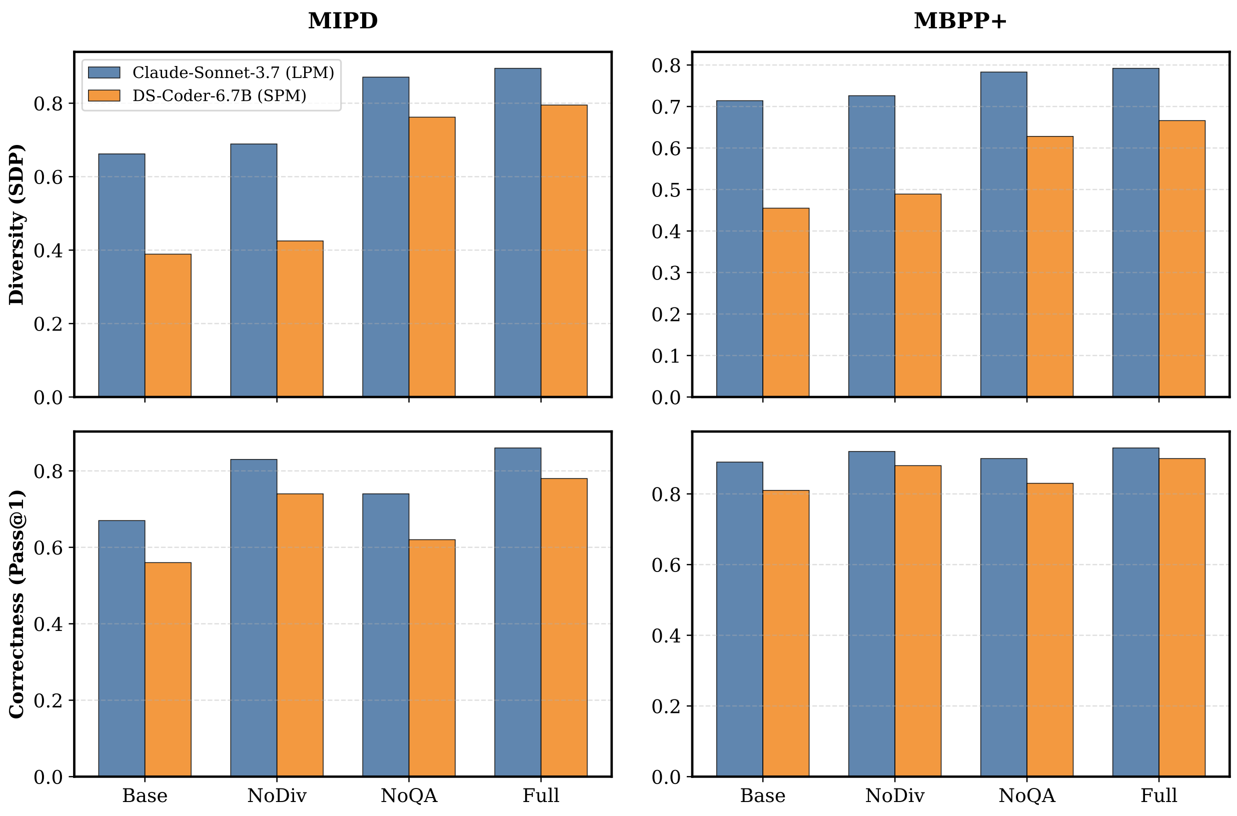


Figure 9: Pass@k Performance Comparison for Vanilla, BoN, and DeQoG Across Five Benchmarks

on vast corpora, they might memorize standard solutions. To mitigate this, we constructed the MIPD dataset with manually crafted, complex tasks specifically designed to be unseen. Additionally, to ensure fair comparison, we applied rigorous prompt engineering best practices to all baseline methods (Vanilla, Best-of-N) rather than deliberately weakening them.

External threats concern the generalizability of our findings. First, our evaluation is limited to Python, which may not reflect performance in statically typed or low-level languages. Second, while we included class-level tasks (ClassEval), the benchmarks primarily focus on algorithmic complexity rather than large-scale, multi-module industrial software systems, which remains a limitation for future work.

The primary construct validity threat lies in the reliability of our metrics, particularly Solution Difference Probability (SDP), which relies on an LLM-as-a-judge approach and may introduce subjectivity. We mitigated this by triangulating SDP with syntactic Levenshtein Similarity (LS) and verifying correctness using expanded test suites (e.g., HumanEval+, MIPD) to reduce the risk of false positives due to insufficient test coverage.

7. Realted Work

N-Version Programming and Software Fault Tolerance. N-version programming was pioneered by Chen and Avizienis[10] as a design diversity approach to enhance software reliability through redundant implementations. Traditional research has extensively studied the independence assumption[11], diversity assurance methodologies[14], and optimization techniques for version selection[15]. However, these approaches rely on manual development by independent teams, incurring prohibitive costs that scale linearly with version count[12, 13]. Early automation attempts through rule-based program synthesis[17] and neural program synthesis[18] reduced development costs but inherited the single-solution focus of conventional code generation, failing to address the fundamental diversity-cost dilemma. Recent work by Ron et al.[19] explored LLM-based N-version generation but relied on simple prompt variations without systematic diversity enforcement, resulting in superficial syntactic differences rather than genuine algorithmic diversity.

LLM-Based Code Generation. Large language models have demonstrated remarkable code generation capabilities across various programming tasks[20, 21, 22, 23]. However, most LLMs are optimized during training to maximize single-solution correctness through techniques like RLHF[24, 44], exhibiting strong bias toward methodologically similar implementations despite superficial variations[25, 37, 38]. Recent studies have identified critical limitations including hallucinations[26, 32, 33, 34], training data bias[35], and brittleness of natural language prompts[41, 40, 42]. While techniques such as self-debugging[31] and Best-of-N sampling improve individual solution quality, they do not systematically address diversity generation. Temperature tuning[40] and instruction tuning[38] have been explored for output diversification, but these inference-time strategies produce shallow variants that preserve core algorithmic structures—insufficient for fault-tolerant systems requiring independent failure modes.

8. Conclusion and Future Work

This study addresses the critical bottleneck of algorithmic homogeneity and nondeterminism in LLM-based N-version programming by introducing DeQoG, a diversity-enhanced and quality-assured framework. Unlike conventional prompt engineering approaches that often yield superficial syntactic variations, DeQoG enforces deep methodological diversity through the synergistic application of Hierarchical Isolation and Local Expansion (HILE) and the Iterative Retention, Questioning, and Negation (IRQN) mechanism. Furthermore, by embedding Large Language Models within a structured workflow orchestration orchestration and utilizing Feedback-Based Iterative Repair (FBIR), the framework successfully transforms probabilistic model outputs into controllable, reliable system components. Empirical evaluations across five benchmarks, including the newly constructed Multi-Implementation Programming Dataset (MIPD), confirm that DeQoG significantly outperforms standard LLM and Best-of-N baselines. These results validate that integrating systematic software engineering principles with GenAI is essential for making automated, low-cost fault-tolerant software development a viable reality.

Future Directions. Building upon the established efficacy of DeQoG, future research will focus on expanding the framework's applicability and rigor. First, we aim to extend the HILE and IRQN methodologies beyond Python to support multi-lingual and system-level software generation, addressing the complexities of heterogeneous computing environments. Second, to further bolster the reliability of the generated N-versions, we plan to integrate formal verification techniques into the structured workflow, providing mathematical guarantees that complement the current testing-based quality assurance. Finally, considering the computational costs associated with iterative reasoning, future work will investigate lightweight model optimization and adaptive token allocation strategies to enhance the cost-effectiveness of large-scale fault-tolerant code synthesis.

References

[1] John W Bennett, Glynn J Atkinson, Barrie C Mecrow, and David J Atkinson. Fault-tolerant design considerations and control strategies for aerospace drives. *IEEE Transactions on Industrial Electronics*, 59(5):2049–2058, 2011.

[2] Afef Fekih. Fault diagnosis and fault tolerant control design for aerospace systems: A bibliographical review. In *2014 American Control Conference*, pages 1286–1291. IEEE, 2014.

[3] Ahmed Shihab Albahri, AA Zaidan, Osamah Shihab Albahri, BB Zaidan, and MA Alsalem. Real-time fault-tolerant mhealth system: Comprehensive review of healthcare services, opens issues, challenges and methodological aspects. *Journal of medical systems*, 42(8):137, 2018.

[4] Benjamin Lussier, Alexandre Lampe, Raja Chatila, Jérémie Guiochet, Félix Ingrand, Marc-Olivier Killijian, and David Powell. Fault tolerance in autonomous systems: How and how much? In *4th IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments (DRHE)*, 2005.

[5] Zishen Wan, Karthik Swaminathan, Pin-Yu Chen, Nandhini Chandramoorthy, and Arijit Raychowdhury. Analyzing and improving resilience and robustness of autonomous systems. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[6] Francesco Flammini, Cristina Alcaraz, Emanuele Bellini, Stefano Marrone, Javier Lopez, and Andrea Bondavalli. Towards trustworthy autonomous systems: Taxonomies and future perspectives. *IEEE Transactions on Emerging Topics in Computing*, 12(2):601–614, 2022.

[7] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang

Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of systems and software*, 86(8):1978–2001, 2013.

[8] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936, 2024.

[9] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. Software engineering for ai-based systems: a survey. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–59, 2022.

[10] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, volume 1, pages 3–9, 1978.

[11] JC Knight and NG Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.

[12] Algirdas Avizienis. The methodology of n-version programming. *Software fault tolerance*, 3:23–46, 1995.

[13] Les Hatton. N-version design versus one good version. *IEEE Software*, 14(6):71–76, 1997.

[14] Michael R Lyu and Algirdas Avižienis. Assuring design diversity in n-version software: a design paradigm for n-version programming. In *Dependable Computing for Critical Applications 2*, pages 197–218. Springer, 1992.

[15] Hidemi Yamachi, Yasuhiro Tsujimura, Yasushi Kambayashi, and Hisashi Yamamoto. Multi-objective genetic algorithm for solving n-version program design problem. *Reliability Engineering & System Safety*, 91(9):1083–1094, 2006.

[16] John PJ Kelly, David E Eckhardt, Mladen A Vouk, David F McAllister, and Alper Caglayan. A large scale second generation experiment in multi-version software: description and early results. In *1988 The Eighteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 9–10. IEEE Computer Society, 1988.

[17] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[18] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems*, 31, 2018.

[19] Javier Ron, Diogo Gaspar, Javier Cabrera-Arteaga, Benoit Baudry, and Martin Monperrus. Galapagos: Automated n-version programming with llms. *arXiv preprint arXiv:2408.09536*, 2024.

[20] Jianxun Wang and Yixiang Chen. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289. IEEE, 2023.

[21] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[22] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[24] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[25] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. On the evaluation of neural code summarization. In *Proceedings of the 44th international conference on software engineering*, pages 1597–1608, 2022.

[26] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM computing surveys*, 55(12):1–38, 2023.

[27] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

[28] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

[29] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927*, 2024.

[30] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[31] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[32] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024.

[33] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):481–503, 2025.

[34] Vibhor Agarwal, Yulong Pei, Salwa Alamir, and Xiaomo Liu. Codemirage: Hallucinations in code generated by large language models. *arXiv preprint arXiv:2408.08333*, 2024.

[35] Lukas Twist, Jie M Zhang, Mark Harman, Don Syme, Joost Noppen, and Detlef Nauck. Llms love python: A study of llms' bias for programming languages and libraries. *arXiv preprint arXiv:2503.17181*, 2025.

[36] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 2002.

[37] Yuqi Zhu, Jia Li, Ge Li, YunFei Zhao, Jia Li, Zhi Jin, and Hong Mei. Hot or cold? adaptive temperature sampling for code generation with large language models. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, pages 437–445, 2024.

[38] Alexander Shypula, Sicheng Li, Bolun Zhang, Vishakh Padmakumar, Kensen Yin, and Osbert Bastani. Does instruction tuning reduce diversity? a case study using code generation. In *International Conference on Learning Representations (ICLR)*, 2025.

[39] Fernando Vallecillos-Ruiz, Max Hort, and Leon Moonen. Wisdom and delusion of llm ensembles for code generation and repair. *arXiv preprint arXiv:2510.21513*, 2025.

[40] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. An empirical study of the non-determinism of chatgpt in code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–28, 2025.

[41] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. *arXiv preprint arXiv:2310.11324*, 2023.

[42] Tanise Ceron, Neele Falk, Ana Barić, Dmitry Nikolaev, and Sebastian Padó. Beyond prompt brittleness: Evaluating the reliability and consistency of political worldviews in llms. *Transactions of the Association for Computational Linguistics*, 12:1378–1400, 2024.

[43] Xiang Chen, Chaoyang Gao, Chunyang Chen, Guangbei Zhang, and Yong Liu. An empirical study on challenges for llm application developers. *ACM Transactions on Software Engineering and Methodology*, 2025.

[44] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.