ToCoGen: Autonomous N-Version Code Generation for Enhanced fault-tolerance Using Large Language Models

ANONYMOUS AUTHOR(S)

N-version programming has long been recognized as an effective approach to achieving software fault tolerance, particularly as quality requirements for software systems continue to escalate. However, traditional methods require substantial manual effort, incur significant costs, and typically result in limited diversity among generated versions. To address these challenges, we propose ToCoGen, the first general-purpose automated multi-version code generation framework that leverages Large Language Models (LLMs) to automatically generate multiple functionally equivalent yet diverse code variants, thereby enabling fault tolerance capabilities. ToCoGen employs diversity-driven heuristic planning, static analysis, dynamic execution and evaluation, along with iterative optimization through dynamic prompt engineering to enhance diversity, functional correctness, and code quality required for fault-tolerant code implementations. Experimental evaluation conducted on three popular datasets—HumanEval, HumanEval+, and LeetCode—as well as a custom dataset, demonstrates ToCoGen's correctness and practical applicability. The results indicate that ToCoGen successfully generates diverse and functionally correct code variants, achieving an average improvement of 65% in version generation efficiency, 30% enhancement in correctness rate, and 12% improvement in fault tolerance effectiveness through fault injection experiments. These comprehensive results on large-scale datasets establish ToCoGen as the first practical solution for general-purpose automated fault-tolerant code generation utilizing LLMs, and demonstrate the potential of LLMs for effective N-version programming in mission-critical software systems.

CCS Concepts: • **Software and its engineering → Software fault tolerance**; *Automatic programming*.

Additional Key Words and Phrases: Software Fault ToleranceN-version programming, Autonomous Code Generation, Dynamic Prompting Technique, Large Language Model

ACM Reference Format:
Anonymous Author(s). 2026. ToCoGen: Autonomous N-Version Code Generation for Enhanced fault-tolerance Using Large Language Models. In *Proceedings of Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering (FSE 2026).* ACM, New York, NY, USA, 25 pages. https://doi.org/10.1145/XXXXXXX.XXXXXXX

1 INTRODUCTION

With the advent of the era of intelligence and digitization, software is increasingly integrated into our daily lives at an exponential growth rate, making software reliability critically important as software failures can lead to catastrophic consequences. However, the inherent complexity of software development and operational environments makes it virtually impossible to completely eliminate defects and anomalies. Consequently, designing systems with fault tolerance capabilities—systems that can maintain core functionality or degrade gracefully when failures occur—has become an increasingly fundamental requirement[3]. As the primary mechanism for protecting software systems after deployment and enhancing overall software reliability, design of software fault tolerance has been widely adopted in numerous safety-critical software applications[13].

FSE 2026, July 5–9, 2026, Montreal, Canada

ACM ISBN 978-1-4503-XXXX-X/26/07
https://doi.org/10.1145/XXXXXXX.XXXXXXX

A fundamental principle of software fault tolerance lies in redundant design strategies. Consequently, N-version programming represents a foundational methodology in software fault tolerance engineering that constructs software diversity through independent development teams to enhance reliability [3, 4]. It is predominantly employed to strengthen the reliability of mission-critical systems. This is achieved by constructing diverse yet functionally equivalent software components that will not simultaneously fail under identical conditions. This redundancy and diversity-based design approach significantly enhances system survivability and service continuity when confronted with software defects, hardware failures, and malicious attacks, thereby making the entire system more robust and improving overall software quality. [30] This has led to continuously growing demand for N-version programming in safety-critical fields.[42] However, N-version programming faces significant practical applicability challenges in real-world deployment. The primary challenge lies in the substantial development costs and resource requirements associated with this methodology [21]. Traditional N-version programming approaches rely on multiple independent development teams to manually craft functionally equivalent software versions.

However, developing multiple program versions incurs prohibitive costs that typically scale linearly with the number of versions, while it remains extremely difficult to guarantee independence in development time and team member assignments. Additionally, standardization and version management difficulties pose significant obstacles, as N-version implementation methodologies vary substantially across different software types, lacking unified standards and best practice guidelines. Furthermore, maintaining multiple versions simultaneously requires implementing every requirement change across all versions, substantially increasing the complexity of configuration management and version control.[44] Current research efforts addressing these challenges remain insufficient. Existing automated code generation techniques primarily focus on single-version functional implementation, lacking systematic diversity generation mechanisms and quality assurance frameworks.[41] Although the program synthesis field has achieved significant progress in recent years, particularly in specification-based code generation, critical technical gaps persist in generating functionally equivalent code with sufficient diversity—a key requirement for effective fault tolerance.[18] In summary, existing N-version programming methodologies and practices exhibit deficiencies in addressing the critical diversity implementation aspect of N-version fault-tolerant code, lacking generalized capabilities for N-version programming fault-tolerant code generation across different software types.[16, 18, 23, 25, 29, 37],

Large Language Models (LLMs) trained on extensive datasets have demonstrated revolutionary capabilities in code understanding and generation, fundamentally transforming software development paradigms. Their capabilities have evolved from simple code completion [11] to encompass multiple dimensions including complex program synthesis [8], code comprehension [9], bug repair [32], and code refactoring [38]. Current research indicates that LLMs can not only handle common programming tasks but also understand complex algorithmic logic, generate high-quality code that satisfies specific constraints, and even achieve performance levels approaching those of expert human programmers in certain benchmarks. This presents unprecedented technical opportunities for addressing the multi-version fault-tolerant code generation problem.

However, automated N-version code generation, compared to conventional automated code generation, shares the same correctness requirements while possessing a critical additional requirement that the latter lacks: strong diversity demands, which constitute the core capability of fault-tolerant code. Some research has attempted to leverage LLMs' cost advantages over traditional development approaches for N-version programming [30], but these efforts remain constrained by issues of LLM-generated code quality and low diversification efficiency, necessitating substantial subsequent development work for version selection and refinement. Our thoughts aims to harness

LLMs' creativity, search capabilities, and cost-effectiveness in conjunction with the specific characteristic requirements of fault-tolerant code to address the aforementioned challenges in N-version fault-tolerant code generation.

However, despite the powerful code comprehension and generation capabilities of LLMs, their direct application to N-version fault-tolerant code generation still faces several significant challenges. Due to the considerably high diversity requirements of N-version fault-tolerant code generation, which demands several times the reasoning capabilities required for single-version code generation, the code understanding and novel algorithmic implementation capabilities of current state-of-the-art large language models cannot yet fully match these requirements.[17] Furthermore, LLMs cannot perfectly adhere to user instructions, thus facing substantial hallucination issues in the field of N-version fault-tolerant code generation, leading to dramatic increases in screening workload and associated costs.[20] Finally, the generated code frequently encounters code quality and reliability problems, including but not limited to syntax errors and violations of intrinsic software specifications. These challenges highlight the need for specialized methodologies to bridge the gap between LLMs' capabilities and the specific requirements of N-version fault-tolerant code generation, addressing the equilibrium between diversity, accuracy, reliability, and cost-effectiveness throughout the process.

This paper presents **ToCoGen**, the first general, autonomous, large language model-based solution framework for Fault-**To**lerant **Co**de **Gen**eration. Our approach treats the LLM as a core tool capable of planning and executing operations to achieve fault-tolerant code generation objectives, while equipping it with a specialized toolkit designed specifically for fault-tolerant code generation. ToCoGen addresses the correctness-diversity challenge through three key innovations. First, We contribute a novel prompting format designed to guide the LLM through the fault-tolerant code generation process, with updates based on the commands invoked by the LLM and the results of previous command executions. Second, we provide a comprehensive set of tools that the LLM can invoke to interact with code base. We offer two groups comprising six tools designed to cover the specialized steps of fault-tolerant code generation. Third, we design a Interactive system that guides tool invocation through a finite state machine and heuristically interprets potentially incorrect LLM outputs. Importantly, we do not hard-code the usage patterns and timing of these tools, but instead allow the LLM to autonomously determine which tool to invoke next based on previously collected information and feedback from prior repair attempts.

To address this gap, we propose **ToCoGen**, the first LLM-based automated general-purpose framework for Fault-**To**lerant **Co**de **Gen**eration. In the initial stage, we designed a code semantic understanding and diversity planning module, where ToCoGen integrates its techniques with LLMs to decompose and preliminarily plan the complex and challenging code diversity generation tasks that are difficult for LLMs to handle independently. We designed a finite state machine for autonomous LLM operations along with its accompanying dynamic and static evaluation methods to provide standardized guidance for N-version code generation, thereby addressing the issue of numerous invalid versions generated by LLMs during the generation process, improving diversity levels, and enabling comprehensive dynamic updates to LLM prompts. Following initial N-version fault-tolerant code generation, ToCoGen employs an iterative refinement and repair workflow to identify and correct common errors. This multi-stage approach ensures both functionality and diversity of the generated N-version fault-tolerant code.

To evaluate the effectiveness of our approach, we applied this methodology to XXX tasks across three popular datasets and constructed datasets. ToCoGen successfully generated multi-version code for XX of these tasks, with fault tolerance effectiveness validated through fault injection experiments. By measuring the costs associated with LLM interactions, we found that ToCoGen averages XXX tokens per version, which translates to XX cents per version based on current

OpenAI GPT-4o model pricing. In summary, our results demonstrate that ToCoGen, as the first fault-tolerant code generation framework, represents the current best practice in the fault-tolerant code generation field.

In summary, our main contributions are as follows:

- We propose ToCoGen, the first LLM-based automated general-purpose framework for N-version fault-tolerant code generation. ToCoGen incorporates a preliminary code understanding and diversity planning module, integrates finite state machine management with accompanying evaluation methods, and employs dynamic prompting and iterative refinement strategies.
- We compiled a dataset containing 150 N-version code generation tasks (80% sourced from code generation domain datasets, 20% manually crafted). ToCoGen significantly outperforms state-of-the-art LLMs including DeepSeek-R1 and GPT-4o in generating N-version fault-tolerant code across three popular datasets in the code generation domain as well as our compiled dataset. ToCoGen achieves an average improvement of 65% in version generation efficiency, 30% in correctness improvement efficiency, and 12% in fault tolerance effectiveness in fault injection experiments on general datasets.
- We publicly release the constructed dataset and key code implementations of ToCoGen for future research in this domain.

2 BACKGROUND

In this section, we introduce the two areas in which our work is rooted: Software fault- tolerance & N-Version programming and Code generation.

2.1 Software fault-tolerance & N-Version programming

N-Version programming is a software development approach[3], which consists of creating N implementations or versions of a specific program. [5]The core idea behind this approach is that when these versions are simultaneously executed, errors can be timely detected and mitigated by comparing their outputs. Ideally, the difference in implementations between versions is maximal, such that any coincidental errors are avoided.[34] While originally devised as a fault-tolerance mechanism, N-Version programming has been adapted to enhance other specific properties of software, such as availability, reliability, performance, or security.[10]

However, the enhancements offered by N-Version programming come with an attached trade-off, as it introduces many challenges throughout the software development lifecycle.[43] These challenges include increased maintenance overhead, increased compute and memory use, or inter-operability issues. Addressing these challenges requires additional effort and careful coordination across engineering teams.[28]

An essential challenge is the increase in development costs, as the time and resources required for the development version increase at least linearly with N. [27]To address this challenge, automating the process of creating new versions is a known and well-studied approach.[14] In this paper, we contribute to the field of automated code generation in N version programming. Another important challenge to the difficulty of diversity. The correctness of fault-tolerant code can be solved through testing. Diversity depends on task attributes and puts huge requirements on the developer's capabilities. In this paper, we also contribute to reducing the difficulty of diversity in N-version programming.[7]

2.2 Code generation

The automated code generation field experienced foundational growth, building upon programming by examples (PBE) methodologies. Sketch-based program synthesis matured during last decade.[7] After that, Neural Program Synthesis gained momentum for demonstrating task-agnostic recurrent architectures with persistent key-value program memory. This work showed that neural networks could learn algorithmic composition, setting the stage for more sophisticated approaches. [12]

In recent years, LLMs have demonstrated significant potential in the field of code generation. Standard language models perform code completion and generation after autoregressive pre-training. Academia and industry have introduced various code LLMs, such as AlphaCode[2], CodeGen[24], CodeGeeX[12], InCoder[15], StarCoder[39], CodeLlama[40], and CodeT5+[35, 36]. Furthermore, general LLMs, such as ChatGPT and DeepSeek, are also widely used for code generation.[1]

However, significant limitations persist in code quality and reliability.[22]Semantic errors represent the most common failure mode, with studies showing approximately 40% of generated code containing exploitable security vulnerabilities. Limited context windows create difficulties with large codebases, while specification ambiguity leads to frequent misunderstanding of natural language requirements.[6] In addition, there is no relevant work in the field of fault-tolerant code generation, but GALA's attempt in the field of N-version generation using code translation has achieved some inspiration that is declaring the significant potential of LLM in the field of N-version code generation despite of its many limitations in certain types of programs and languages.[26]

3 APPROACH

3.1 Overview

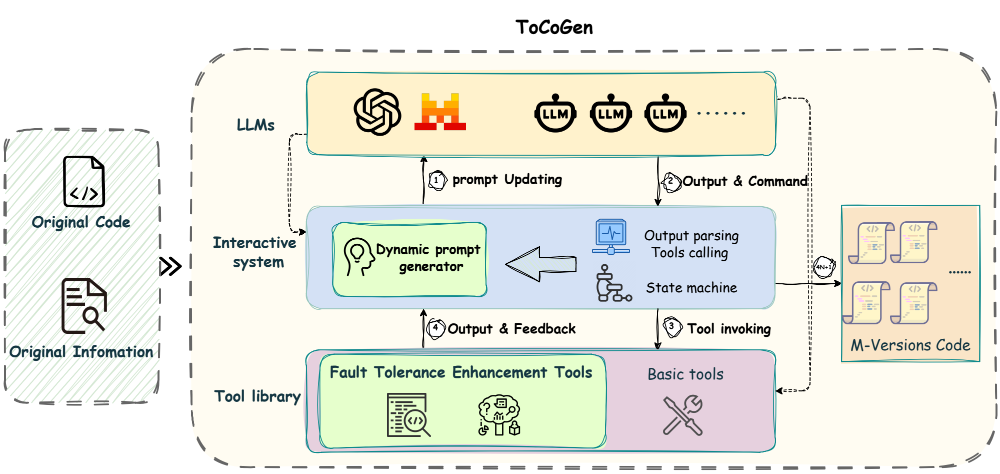

Fig. 1. Oveview

Figure 1 shows the full picture of the ToCogen method, which consists of three components: an LLMs proxy (upper side), a tools library (lower side), and an interactive system (middle) that coordinates communication and feedback between the two. Given an initial version of the code, the interactive system initializes the LLM proxy with prompts containing task information and how to perform tasks using the provided tools (arrow 1). LLM gives feedback and responds by suggesting calling one of the available tools (arrow 2), which the interactive system parses and executes (arrow

3). The output of the tool (arrow 4) is then integrated into the prompt for the next call of the LLM, and the process iterates until the N version code generation is complete or the predefined budget is exhausted.

3.2 Dynamic prompting in interactive system

3.2.1 Dynamic prompting definition. In the vast majority of cases, LLM capabilities prove insufficient to generate code output that meets requirements in a single response, thereby necessitating multiple interactions with the LLM to enhance code generation quality. Consequently, ToCoGen operates in either semi-iterative or fully iterative modes: regardless of the specific mode, the LLM engages with the Interactive System, which determines whether to invoke tools from the tool library. During each iteration round, the method queries the LLM once, with the model's input being updated based on the commands (tool invocations) called by the LLM in previous rounds and their corresponding results. We refer to the model input as a dynamic prompt. The dynamic prompt constitutes a sequence of text segments $T = [T_0, T_1, \ldots, T_n]$, where $T_i(r)$ refers to the segment in round r, and each segment T_i has two possible characteristics:

- A static segment that remains unchanged across all rounds, such that $Ti(r) = Ti(r')$ for all r, r'. These segments typically belong to abstract high-level descriptions, encompassing identity definitions, task objectives, task guidance, and input-output specifications.
- A dynamic segment that may vary across different rounds, such that $Ti(r) \neq Ti(r')$ may exist for some r, r'. These segments typically belong to concrete real-time feedback descriptions, including state descriptions, available tools, collected historical information, the last executed command and its results, as well as round counters.

This section defines the framework's expertise, namely solving code generation tasks with fault-tolerant properties, and outlines the framework's main goals: understanding the original version code and generating functionally equivalent and diverse code. In order to improve generation efficiency and reduce labor costs, the tip emphasizes that LLM's decision-making process is autonomous and should not rely on user assistance.

Table 1. Sections of the dynamically updated prompt.

Prompt section	**Nature**
Role	Static
Goals	Static
Guidelines	Static
State description	Dynamic
Available tools	Dynamic
Gathered information	Dynamic
Specification of output format	Static
Last executed command and result	Dynamic

3.2.2 Role. This section defines the framework's expertise, namely solving N-versions code generation tasks with fault-tolerant properties, and outlines the framework's main goals: understanding the original version code and generating functionally equivalent and diverse code. In order to improve generation efficiency and reduce labor costs, the tip emphasizes that LLM's decision-making process is autonomous and should not rely on user assistance.

3.2.3 Goals. We define five goals for the LLM to pursue, which remain the same across all rounds:

- Understand the original version of the code: Understand the implementation and logical structure of the original version of the code, and abstract the tasks completed by the code
- Generate multiple versions of code: Generate multiple versions of code with equivalent functions and as different as possible
- Verify the correctness and diversity of the code: Verify whether the generated version is the correct code and whether it has diversity under the existing indicators
- Proper Modification: If the above correctness and diversity are not satisfied, suggest a modification plan
- Iterative task: Continue to collect information and suggest modifications until the generated code is correct and diverse

3.2.4 Guidelines. We provide a set of guidelines. First, we inform the model that detailed introduction to the mechanism and principle of N-version code fault-tolerance, and provide the detailed demands for code correctness and diversity. Second, we provide a list of tasks with N-version codes as their solution, which are proved to be able to increase the capability of fault-tolerant. The list is based on prior effective tasks. For each task, we provide a corresponding natural language descriptions, original version code and effective several versions of example codes. Third, we instruct the model to insert comments above the new versions code, which serves two purposes. On the one hand, the comments allow the model to explain its reasoning, which has been shown to enhance the reasoning abilities of LLMs. On the other hand, commenting will ultimately help human developers in understanding the nature of the edits. Fourth, we instruct the model to conclude its reasoning with a clearly defined next step that can be translated into a call to a tool. Finally, we describe that there is a limited budget of tool invocations, highlighting the importance of efficiency in selecting the next steps. Specifically, we specify a maximum number of rounds (20 by default).

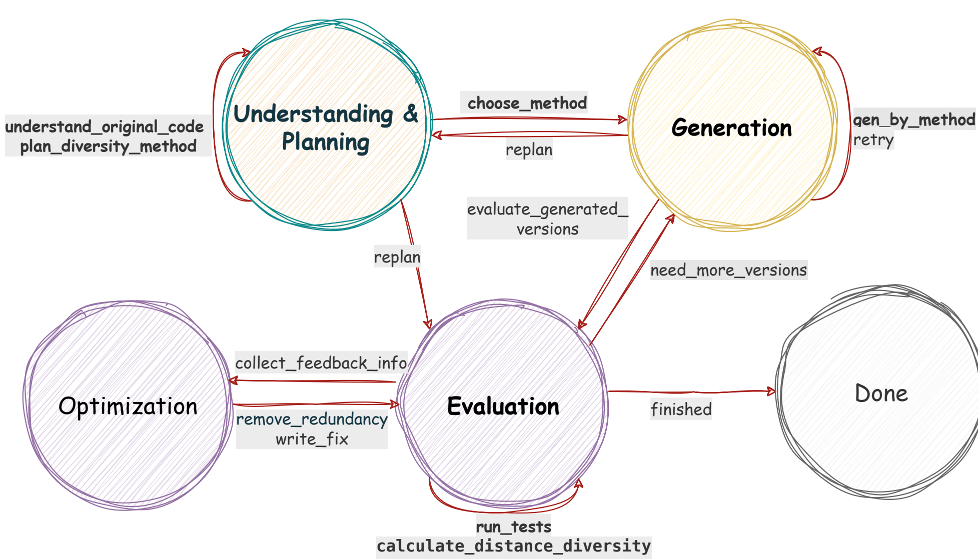

Fig. 2. State machine

3.2.5 *State machine.*

To guide the LLM to use the available tools in an efficient and meaningful way, we define a finite state machine that constrains which tools are available at a particular point in time. The motivation is our observation that in early experiments without such guidance, the LLM often got lost in aimless exploration. Figure 4 shows the finite state machine, which we designed to simulate the states that a human developer would go through when developing a new version of the code with reference to the original version of the code. Each state is associated with a set of tools that can be called by the LLM, which are described in section 3.4. Importantly, the LLM can freely transition between states at any point in time by using tools. That is, despite the guidance provided, the state machine does not enforce a strict order for tool invocations.

The state description section of the prompt informs the LLM about its current state:

- Understanding and Planning: The LLM's state commences from this initial state, wherein it performs comprehension of the original version code and conducts diversified implementation planning for the task. Once the LLM completes code understanding and method planning, it selects a strategy to guide the generation of alternative code versions. Throughout the generation process, the LLM may discard previous strategies and re-engage in understanding or planning phases. Following strategy selection, the LLM automatically transitions to the subsequent state.
- Generation: In this state, the framework directs the LLM to generate alternative code versions, with iterative attempts permitted as necessary. Once the framework successfully generates a sufficient number of code versions, it transitions to the next state.
- Evaluation: In this state, the LLM conducts comparative validation based on its currently selected strategy and the generated code. For instance, functional equivalence verification is performed through test case execution, while diversity validation is conducted using tools such as semantic distance calculation. When necessary, the framework may return to previous generation or understanding and planning states to generate additional valid versions or to reselect diversification strategies. Simultaneously, if issues arise during test case execution or diversity validation, the system may enter the optimization state after collecting information from relevant tool outputs.
- Optimization: The optimization state represents an isolated state, where both the preceding and succeeding states should be the evaluation state. The system can select different tools based on various types of information to return to the evaluation state. For example, if test failures occur, the system can generate corrected code based on error information; if excessive versions are produced, the system can merge or select from duplicate versions.
- Done: If all evaluation validations are successfully passed in the evaluation state, the process can be concluded by invoking a specific command that indicates the successful completion of N-version code generation.

3.2.6 *Available Tools.*

This section of the prompt describes a set of tools that the LLM can call at the current state. Each tool has a name, a description, and a set of typed arguments (Section 3.4).

3.2.7 *Feedback information.*

A fundamental capability for generating fault-tolerant code involves the systematic collection of information regarding the original code version and diversification strategies, which establishes the foundation for determining subsequent command invocations. To ensure this information remains accessible to the LLM, we implement a dedicated hint section that catalogs information gathered through various tool invocations. This hint section functions as a persistent memory mechanism for the LLM, facilitating the retrieval of information from previous iteration rounds. The collected information is systematically organized into distinct subsections, each containing the output generated by a specific tool invocation.

```
{
"thoughts" : "I need to generate diverse implementations of this array sorting
    algorithm. The current version uses a two-pointer approach, but I should explore
    other algorithmic strategies and programming styles to create varied solutions.",
"command" : {
"name" : "plan_diversity_method",
"args" : {
"algorithm_approach" : "sortArrayByParityII",
"target_variants" : 4,
"focus_areas" : ["functional_style", "recursive_approach", "space_time_tradeoffs"]
}}}
```

Fig. 3. Example of a response of the LLM

3.2.8 Specification of Output Format. Given the dynamic prompting mechanism, the LLM provides a single response per iteration round. To enable the interactive system to parse these responses effectively, we specify a predefined output format. The "reasoning" field provides a textual description of the LLM's reasoning process when determining the subsequent command. Requiring the LLM to articulate its reasoning enhances the transparency and interpretability of the methodology, provides a mechanism for debugging potential issues within the LLM's decision-making process, and contributes to improving the LLM's reasoning capabilities. The "command" field specifies the next command to be executed, encompassing both the tool name to be invoked and the corresponding parameter set. For example, Figure 3 shows the response of LLM. The model expresses the need for modular analysis of the original version of the code and suggests a command generated using the MLR graph.

3.2.9 Last Executed Command and Result. This section of the prompt contains the last command (tool name and arguments) that was executed (if any) and the output it produced. The rationale is to remind the LLM of the last step it took, and to make it aware of any problems that occurred during the execution of the command. Furthermore, we remind the LLM how many rounds have already been executed, and how many rounds are left.

3.3 Prompts Updating Algorithm

The Unified Dynamic Prompt Construction and Updating algorithm (as shown in Algorithm **??**) implements adaptive prompt management within the ToCoGen framework, ensuring that the large language model receives consistent and relevant guidance throughout the N-version code generation process. The algorithm comprises two main phases: Initial Prompt Construction and Dynamic Prompt Update, utilizing a conditional branching mechanism to unify different prompt management requirements.

In the Initial Prompt Construction phase, the algorithm first parses the dataset task T and extracts key elements, including the $task\_code$, $task\_description$, and $test\_cases$, which serve as the foundation for functional equivalence verification. Next, it initializes $static\_parts$, comprising agent role definitions, generation objectives, guidelines, and output format specifications. The task information $task\_info$ is formatted to maintain original problem context, while the $state\_desc$ is set to "Understanding & Planning" mode, guiding the LLM toward code analysis and strategy formulation.

In the Dynamic Prompt Update phase, the algorithm follows the principle of prompt consistency by extracting and preserving static components and original task information from the previous prompt. Based on the current state *new_state* and *round_count*, the algorithm generates a new state description, selects appropriate available tools *tools*, and updates feedback information *feedback* based on the *execution_result*. The *progress_info* reflects current generation progress and remaining iteration budget. Finally, the algorithm integrates all components into a complete *updated_prompt* through modular concatenation. Figure 5 shows shows a simplified prompt example.

3.4 Fault- tolerance tools library

Existing LLM-based code generation aims at optimizing for producing single, high-quality implementations. In contrast, fault-tolerant N-version programming requires a fundamentally different approach: generating multiple versions that are simultaneously equivalent in function yet diverse in implementation—a constraint that existing tools cannot address.

To address this problem, we designed the fault-tolerant enhancement tool library besides dynamic prompt. The tool library has two main innovations:One of the innovation is the fault-tolerance-aware tool orchestration. Rather than using generic code analysis tools, our approach provides LLMs with specialized tools that explicitly balance the tension between correctness and diversity—the fundamental trade-off in fault-tolerant N-version programming. Each tool is designed to serve the specific needs of generating code variants that fail independently, maximizing the system's overall reliability. Another key innovation of our approach is to let an LLM autonomously decide which tools to call to generate N versions of fault-tolerant code.

The tools we provide to the LLM (Table 2) are inspired by the thinking and tools used by developers in the field of software fault-tolerance in their manual development of N-version fault-tolerant code and automatic code generation. ToCoGen enhances the fault- tolerance capabilities of generated N-version codes through a carefully designed tools library.

Table 2. Core tools invoked by ToCoGen.

Prompt section	**Nature**
Role	Static
Goals	Static
Guidelines	Static
State description	Dynamic
Available tools	Dynamic
Gathered information	Dynamic
Specification of output format	Static
Last executed command and result	Dynamic

3.4.1 Functional equivalence tools. These tools ensure that multiple generated versions are functionally equivalent. The *understand_original_code* tool analyzes the given original code to extract deep understanding of its functional logic, data structures, and algorithmic implementations, establishing the foundation for subsequent N-version generation. The *gen_by_method* tool serves as the core code generator, capable of producing functionally equivalent but differently implemented code versions based on various generation strategies (including algorithmic variants, structural variants, control flow variants, etc.). The run_tests tool performs comprehensive testing validation, including unit tests, integration tests, and property-based tests, to ensure correctness and functional consistency across all generated versions.

3.4.2 Diversity assessment tools. This category focuses on diversity planning and measurement, which is crucial for the fault-tolerance capability of N-version programming. The *plan_diversity_method* tool formulates targeted diversification generation strategies based on identified diversity opportunities, fault models, and resource constraints, while prioritizing different diversity dimensions. The *calculate_distance_diversity* tool calculates distance diversity between code variants through multiple dimensions including syntactic, semantic, and execution behavior, quantifying the degree of differences among implementation approaches to ensure the generated version set has maximum fault-tolerance potential. The $LLM - based\_analyze\_and\_assess$ tool leverages large language models as intelligent judges to conduct comprehensive quality and diversity evaluation of generated code variants, ensuring that generated versions meet fault-tolerance requirements.

3.5 Other basic components

We use multiple general or large code model-based LLMs that are isolated from each other, including a primary LLM for initial version code understanding and generating fault-tolerant code, and a secondary LLM for evaluation and repair.

While the remaining fundamental components may not exhibit significant innovation, they have been extensively utilized in other domains such as code analysis and code repair, and demonstrate considerable efficacy in performance enhancement, particularly given the inherent unpredictability of LLMs. These components can be categorized into two primary classes. The first category comprises basic tools within the tools library: the $write_fix$ tool contributes to improving the correctness of fault-tolerant code outputs to a certain extent, while state transition control tools including retry, $evaluate_generated_versions$, $replan$, $need_more_versions$, and $finished$ facilitate smooth state flow throughout the generation process. The second category encompasses components within the interactive system: the Output parsing component performs validation and error correction on various outputs, effectively mitigating workflow disruptions caused by unexpected LLM responses, while the Tools calling component invokes corresponding tools through methods such as regularized matching for valid commands from the LLM. To prevent tool execution from interfering with the LLM, these components execute commands within isolated environments.

4 EVALUATION

4.1 Research Questions

To evaluate our approach we aim to answer the following research questions:

RQ1: Effectiveness of N-versions code Generation. How effective is ToCoGen at generating functionally equivalent and diverse code versions?

RQ2: Effectiveness of N-versions code for fault-tolerance. How much does generated N-version code by ToCoGen improve system reliability compared to single-version implementations under different fault injection scenarios?

RQ3: Cost-Effectiveness tradeoff. What are the computational costs of ToCoGen?

4.2 Datasets

To evaluate ToCoGen, we have conducted extensive experiments on four datasets, including three popular datasets that are representative of the field of code generation: MBPP, HumanEval, LeetCode-based dataset, and a dataset we compiled for N-version fault-tolerant tasks. The details of these datasets are described as follows.

- **Mostly Basic Python Programming (MBPP)** is a dataset comprising diverse Python programming problems. It contains 974 code-generation tasks that cover a wide range of programming scenarios. Each problem is provided with an English requirement, a function signature, and three manually created test cases for validating the generated functions.
- **HumanEval (HE)** is a benchmark dataset designed to assess the code generation capabilities of large language models. It consists of 164 manually crafted Python programming problems, each accompanied by corresponding test cases to verify the correctness of the generated code.
- **LeetCode-based Dataset (LCBD)** is an online judge platform that suggests programming problems to registered users. It provides algorithmic problems with varying levels of difficulty and test cases with large input sizes, which distinguishes it from the above two benchmark datasets.

It is worth mentioning that task datasets such as HumanEval or MBPP that are often used when evaluating LLMs for code evaluation are not actually completely suitable for our purpose. This is because the N-version code generation task with fault-tolerant effects that we study must meet a requirement: that is, it can provide multiple solutions, or at least there is the possibility of generating multiple different implementations. Otherwise, because the solution to be generated is very short or because the problem is not an algorithmic problem, there are fewer possible changes between different implementations, which will greatly underestimate the effectiveness of the method and lose its meaning as a benchmark dataset. Despite this, we still chose MBPP and HumanEval as our two datasets, one is that they are classic enough and widely used, and the other is to be more practical. But we will pay more attention to the performance of the method on complex datasets such as LeetCode.

Multi-Implementation Programming Dataset (MIPD). This dataset has been specifically curated for N-version fault-tolerant code generation tasks, comprising 200 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 50 problems each, along with 50 additional problems derived from real-world coding scenarios. **Selection Criteria:** The primary selection criterion is based on the existence of multiple viable implementation methodologies. We conducted manual screening according to this characteristic. To ensure dataset diversity, we systematically selected problems spanning various algorithmic categories, including sorting algorithms, search techniques, dynamic programming, mathematical computations, recursive approaches, bitwise operations, data structures, and string processing, among other problem types. We endeavored to achieve uniform coverage across all algorithmic categories while excluding tasks that require complex contextual operations.

Table 3. Characteristics of Four Code Generation Datasets

Dataset	Problems	Avg. Test Cases	Avg. Lines of Code Solution	Data Source
HE	164	7.7	6.3	Hand-Written
MBPP	974	3.0	6.7	Hand-Written
LCBD	150	15.3	12.8	Online Judge
MIPD	200	8.5	9.4	Mixed Sources

4.3 Experimental Setup

4.3.1 Metrics. We demonstrate that our N-version fault-tolerant code generation method outperforms previous approaches by achieving higher generation efficiency, exploring more diverse solutions without sacrificing the utilization of good solutions, and exhibiting superior fault-tolerant capabilities.

Our evaluation employs a comprehensive set of metrics to assess ToCoGen's performance, including the version generation efficiency of fault-tolerant code, the comprehensive fault-tolerance capability of generated N-version code, and the cost of code generation. To evaluate the correctness and efficiency of fault-tolerant code version generation (RQ1), we employ two primary metrics:

Test Pass Rate (TPR): which measures the percentage of generated application code that successfully passes tests without errors after undergoing our automatic iterative optimization process with no human intervention.

$$TPR = \frac{N_p}{N_{total}} \times 100\% \tag{1}$$

Average Pass Iteration Count (APIC) which characterizes and compares the impact of different LLM capabilities on ToCoGen's N-version fault-tolerant code generation efficiency.

$$APIC = \frac{\sum_{i=1}^{n} iterations_i}{n} \tag{2}$$

For diversity assessment, we adopt two diversity measurement indicators:

Mean BERT cosine similarity(MBCS) which is between embeddings of candidate solution pairs, averaged over all problems, where embeddings were obtained using CodeBERT, a pretrained model for understanding code semantically;

$$\text{MBCS} = \frac{1}{|\mathcal{X}|} \sum_{\langle p,H\rangle \in \mathcal{X}} \frac{1}{|\mathcal{S}_p|(|\mathcal{S}_p|-1)} \sum_{\substack{s,s' \in \mathcal{S}_p \\ s \neq s'}} \frac{\text{embed}(s) \cdot \text{embed}(s')}{||\text{embed}(s)|| \cdot ||\text{embed}(s')||} \tag{3}$$

Diversity of Method(DoM) is a LLM-based method (LBMD) compares the implementation methodology diversity of all generated program pairs. The core insight of this approach is the non-transitivity of similarity relationships. Specifically, for n generated codes, we construct all possible code pairs totaling $C(n, 2)$ pairs, use LLMs to evaluate the similarity $S(c_i, c_j) \in \{0, 1\}$ of each code pair (where 1 indicates similarity), and then calculate the overall diversity score DoM.

$$DoM = 1 - \frac{\sum_{i<j} S(c_i, c_j)}{\binom{n}{2}} \tag{4}$$

In practical evaluation, this method samples code subsets from one task for comparison to handle large-scale scenarios, first back-translating code to natural language descriptions, then combining the code itself with the back-translated ideas using specified LLMs for similarity judgment. The metric ranges from $[0, 1]$, where $DoM = 0$ indicates all codes implement the same idea (no diversity), $DoM = 1$ indicates all code ideas are completely unique (maximum diversity), and the DoM value is equivalent to the probability that two randomly selected programs are dissimilar, effectively capturing code diversity at the implementation strategy level rather than merely the syntactic level.

To evaluate the comprehensive fault-tolerance capability of generated N-version code, we conducted fault injection experiments (RQ2) using three metrics:

Failure Rate(FR) is calculated as the percentage of tasks that pass testing and are completed out of the total number of tasks.

$$FR = \frac{\text{Number of tasks with at least one version passing tests}}{\text{Total number of tasks}} \times 100\% \tag{5}$$

Majority Consistency Rate(MCR) represents the proportion where at least half of the versions produce identical outputs.

$$MCR = \frac{\text{Number of tasks with majority versions producing same output}}{\text{Total number of tasks}} \times 100\% \tag{6}$$

Complete Consistency Rate(CCR) indicates the proportion where all integrated versions produce completely identical outputs.

$$CCR = \frac{\text{Number of tasks with all versions producing identical output}}{\text{Total number of tasks}} \times 100\% \tag{7}$$

4.3.2 Baselines. To evaluate ToCoGen, we consider 6(2 x 3) baselines for comparisons:

- Vanilla LLM Baseline: The Vanilla LLM baseline represents the standard code generation approach where the language model generates a single solution for each programming problem through one forward pass. This method employs greedy decoding or standard sampling strategies (temperature=0.2, top-p=0.95) to produce code directly from the input prompt without any post-processing or refinement steps. The generated code is used as-is for evaluation, representing the model's immediate response capability. This baseline serves as the fundamental comparison point, demonstrating the raw performance of large language models in code generation tasks without additional optimization techniques or multiple sampling strategies.
- Best of N Baseline: The Best of N method generates multiple candidate solutions for each programming problem using sampling-based decoding with moderate temperature (0.6-0.8) to encourage diversity. Each candidate is evaluated against provided test cases or functional correctness criteria, and the solution with the highest score is selected as the final output. This approach leverages the stochastic nature of language model generation to explore different solution paths and implementations. By running multiple inference passes and selecting the best-performing candidate, this method significantly improves the success rate compared to single-shot generation, representing a straightforward yet effective inference-time optimization strategy.

For each baseline, we respectively employed three advanced LLMs in the current code generation field: GPT-4.1 and Claude-3.7 and Deepseek-V3(open source). Also, we carefully designed prompts of baselines . These prompts follow established practices in existing code generation work as well as widely adopted LLM usage techniques,which can ensure fair comparison and avoid underestimating the capabilities of LLMs.

4.3.3 Implementation. We use Python 3.10 as our primary programming language. Docker is used to containerize and isolate command executions for enhanced reliability and reproducibility. ToCoGen makes use of GPT-4o API from OpenAI and Codestral from Mistral.

4.4 Results & analysis

4.4.1 RQ1: Effectiveness of N-versions code Generation.

T. o address RQ1, we conducted experiments on the four datasets mentioned in 4.2, each containing varying numbers of code tasks. We select GPT-4.1 and Claude-3.7 because they represent state-of-the-art LLMs in the programming domain, The experimental input consists of code tasks from the datasets along with their initial versions and descriptive information, while the output comprises N-version code generated by ToCoGen for each task. We collected each generated code version, conducted test case evaluations and static code analysis, and recorded the complete large language model operation workflow logs from the initial query to the final output code results. Subsequently, we evaluated the results according to the metrics outlined in 4.3.1, where the *TPR* and *APIC* metrics primarily describe the code generation correctness and robustness of ToCoGen compared to baselines, which forms the foundation for whether N-version code can possess fault-tolerance capabilities. The diversity measurement metrics *MBCS* and *DoM* describe the similarity between different versions from different perspectives, i.e., code diversity, which constitutes the core factor for whether N-version code can achieve fault-tolerance capabilities. These metrics collectively describe ToCoGen's ability to generate N-version fault-tolerant code from a theoretical perspective.

Table 4. Performance Comparison of Different Methods on Four Datasets

Method	**MBPP**		**HumanEval**		**LBSD**		**MIPD**	
	TPR(%)	**APIC**	**TPR(%)**	**APIC**	**TPR(%)**	**APIC**	**TPR(%)**	**APIC**
Vanilla (GPT-4.1)	58.2	3.4	55.1	3.7	51.3	4.2	59.8	3.1
Vanilla (Claude-3.7)	57.3	3.2	63.4	3.5	59.7	3.8	67.2	2.9
Vanilla (Deepseek-V3)	38.9	4.6	34.2	5.1	31.8	5.3	39.1	4.4
BoN (GPT-4.1)	66.5	2.8	63.8	3.1	59.4	3.6	67.9	2.7
BoN (Claude-3.7)	66.1	2.7	72.6	2.9	68.2	3.2	75.8	2.5
BoN (Deepseek-V3)	48.2	3.9	43.7	4.3	40.1	4.7	53.4	3.7
ToCoGen(GPT-4.1)	91.2	1.8	86.4	2.1	78.3	2.4	83.7	1.9
ToCoGen(Claude-3.7)	92.1	1.6	88.2	1.9	80.5	2.2	85.9	1.7
ToCoGen(Deepseek-V3	76.4	2.5	71.8	2.8	65.2	3.1	72.1	2.4

The results presented in Table 4 reveal a significant limitation of directly employing GPT-4.1, Claude-3.7, and Deepseek-V3 to generate N-version fault-tolerant code from programming tasks and their standard solutions: the overall code quality remains suboptimal. Under fully automated conditions, the baseline methods achieved average TPRs of 56% (GPT-4.1), 62% (Claude-3.7), and 36% (Deepseek-V3). Test case failures primarily stemmed from issues such as syntax errors, usage of undeclared or non-existent packages or components, and inadequate special character handling. Although all three baseline approaches fell short of ToCoGen's performance level, their effectiveness varied across different datasets. Vanilla (DeepSeek-8B) consistently performed worst across all datasets, which was unsurprising given its limited parameter count. While Vanilla (GPT-4.1) outperformed Vanilla (Claude-3.7) by 1% on the MBPP dataset, Vanilla (Claude-3.7) significantly exceeded Vanilla (GPT-4.1) on HumanEval, LBSD, and MIPD datasets with a leading margin of approximately 8%, highlighting Claude-3.7's superior capability in handling more complex coding tasks in baseline scenarios.

Similarly, compared to the three strategy-free baselines, the BoN strategy yielded substantial improvements, with an average enhancement of 9%. BoN (Claude-3.7) achieved over 70% TPR, indicating that more than seven out of ten generated code instances were correct. Considering the presence of high-difficulty LeetCode-based programming problems in the datasets, this result is quite impressive. Notably, we observed that BoN (DeepSeek-8B) improved by as much as 14% over Vanilla (DeepSeek-8B) on the MIPD dataset, while BoN (Claude-3.7) and BoN (GPT-4.1) showed improvements of 9% and 8% respectively compared to their original baselines, both remaining below 10%. This partially validates an interesting finding: LLMs with weaker baseline capabilities tend to exhibit greater variance in the quality of their generated solutions.

Comparing ToCoGen with the baselines reveals significant improvements in key metrics. TPR, as a crucial indicator of code generation correctness, demonstrates ToCoGen's substantial advantages. On the MBPP and HumanEval datasets, ToCoGen achieved TPRs of 91% and 86% with GPT-4.1, and 92% and 88% with Claude-3.7, respectively. On the LBSD and MIPD datasets, ToCoGen with Claude-3.7 achieved TPRs of 80.5% and 85.9% respectively, significantly surpassing the corresponding baselines of Vanilla (Claude-3.7) (59.7%, 67.2%) and BoN (Claude-3.7) (68.2%, 75.8%). This substantial improvement demonstrates ToCoGen's superior capability in generating code of the required type with enhanced correctness and code quality. It is worth noting that the foundational capabilities of LLMs serve as a major factor influencing ToCoGen's performance, and the results shown in Table 1 indicate that ToCoGen (Claude-3.7) generates more robust code that is better suited for practical deployment compared to the other two large language models.

Table 5. Code Diversity Comparison Across Different Methods and Datasets

Method	**MBPP**			**HumanEval**			**LBSD**			**MIPD**		
	MBCS	**LBMD**	**JCS**	**MBCS**	**LBMD**	**JCS**	**MBCS**	**LBMD**	**JCS**	**MBCS**	**LBMD**	**JCS**
Vanilla (GPT-4.1)	0.742	0.183	0.681	0.718	0.195	0.674	0.634	0.241	0.592	0.598	0.267	0.558
Vanilla (Claude-3.7)	0.695	0.201	0.659	0.681	0.218	0.643	0.587	0.289	0.567	0.541	0.324	0.523
Vanilla (Deepseek-V3)	0.821	0.154	0.723	0.793	0.167	0.698	0.731	0.203	0.641	0.687	0.241	0.601
BoN (GPT-4.1)	0.689	0.225	0.634	0.671	0.241	0.621	0.573	0.312	0.541	0.532	0.367	0.498
BoN (Claude-3.7)	0.652	0.248	0.617	0.634	0.267	0.598	0.521	0.356	0.512	0.478	0.421	0.467
BoN (Deepseek-V3)	0.758	0.192	0.675	0.731	0.208	0.652	0.664	0.274	0.594	0.614	0.319	0.551
ToCoGen (GPT-4.1)	0.423	0.521	0.378	0.398	0.543	0.362	0.341	0.687	0.295	0.298	0.743	0.251
ToCoGen (Claude-3.7)	0.401	0.567	0.354	0.372	0.589	0.331	0.312	0.742	0.268	0.267	0.821	0.218
ToCoGen (Deepseek-V3)	0.587	0.381	0.498	0.561	0.402	0.476	0.493	0.512	0.423	0.445	0.587	0.381

Note: MBCS - Mean BERT Cosine Similarity (lower is better); LBMD - LLM-based Method Diversity (higher is better); JCS - Jaccard Coefficient Similarity of CPU instruction sets (lower is better).

The results presented in Table5 reveal another significant limitation of directly employing GPT-4.1, Claude-3.7, and Deepseek-V3 to generate N-version fault-tolerant code from programming tasks and their standard solutions: insufficient code diversity. Without applying any additional auxiliary techniques, relying solely on appropriate and fixed prompts, the N-version fault-tolerant code generated by large language models exhibits high semantic similarity $(0.7182, 0.6811, 0.7934)$. This indicates that in most cases, even when informed about the concept of N-version fault tolerance and prompted to generate using different methods, original large language models tend to favor similar or specific types of generation approaches. The LBMD results corroborate this finding, with the method diversity scores provided by large language models $(0.18, 0.15, 0.26)$indicating low diversity at the methodological level, often representing different expressions of the same underlying approach, which fundamentally affects fault-tolerance capabilities. For instance, examining two code segments extracted from the output results reveals that although they employ different syntactic implementations or variable names, their essential mathematical methods and implementations remain identical. While these appear to superficially fulfill our requirement for "different implementations," they do not meet our deeper N-version fault-tolerance needs, indicating that baseline methods cannot effectively understand our requirements and generate the desired fault-tolerant code. This limitation stems partly from models being typically optimized during training to generate single correct answers.

Comparing ToCoGen with the baselines reveals significant improvements in diversity metrics, encompassing both semantic similarity and diversity within the method space. ToCoGen (Claude-3.7), ToCoGen (GPT-4.1), and ToCoGen (DeepSeek-V3) all demonstrate substantial performance improvements, with particularly notable enhancements in the latter. Remarkably, ToCoGen (Claude-3.7) achieved an impressive 80% improvement in method diversity on LBSD and MIPD datasets, meaning the probability that any two randomly selected methods differ has essentially doubled, which in a significant sense implies stronger fault-tolerance capabilities.

Additionally, we calculated the Jaccard similarity coefficients of CPU instruction sets after different code executions. On average, ToCoGen (Claude-3.7) achieved a Jaccard similarity coefficient of 0.318 between its instruction sets and those of the dataset's standard solutions, substantially lower than the baselines using the same LLM (0.674, 0.543). It is noteworthy that the diversity metric results for MBPP and HumanEval datasets are consistently lower than those of the latter two datasets, with *MIPD* exhibiting the highest diversity. Since *MIPD* was curated and compiled from the first three general-purpose datasets, this difference stems from the inherent limitations in the diversity of implementable methods within the aforementioned datasets.

Answer to RQ1: Our experiments demonstrate that ToCoGen can largely create correct and diverse N-version code. Across multiple code tasks in four datasets, both the correctness and diversity of its generated code significantly exceed the baselines. It achieves up to 92% TPR, superior generation efficiency (APIC), and excellent diversity levels (MBCS, LBMD), which means that consistent with the core assumptions of N-Version programming, the generated N-version code has stronger fault-tolerance capabilities compared to baseline-generated code.

Table 6. Fault Tolerance Evaluation Results: Performance comparison across four datasets using three key metrics after fault injection. (G=GPT-4.1, C=Claude-3.7, D=deepseek-V3).

Methods	MBPP			HE			LBSD			MIPD		
	FR(%)	MCR(%)	CCR(%)	FR(%)	MCR(%)	CCR(%)	FR(%)	MCR(%)	CCR(%)	FR(%)	MCR(%)	CCR(%)
Vanilla-G	24.3	58.7	32.1	28.9	54.2	28.4	35.6	47.8	18.9	32.1	51.3	23.7
Vanilla-C	21.8	62.4	38.9	25.7	59.1	34.6	31.2	52.8	24.3	28.4	56.7	29.8
Vanilla-D	38.9	43.2	19.7	42.1	39.8	16.3	48.7	34.6	12.4	45.3	37.9	14.8
BoN-G	19.6	66.8	42.7	23.4	63.5	38.9	28.7	58.4	31.2	26.1	61.7	35.4
BoN-C	17.2	71.3	48.6	20.8	68.7	43.2	24.9	63.5	36.8	22.6	66.4	41.7
BoN-D	31.4	52.7	28.3	35.8	48.9	24.7	41.2	43.6	19.8	38.7	46.3	22.1
ToCoGen-G	**13.1**	**78.9**	**57.2**	**15.3**	**75.6**	**52.8**	**19.1**	**70.2**	**46.1**	**17.4**	**73.8**	**50.9**
ToCoGen-C	**12.4**	**79.6**	**58.3**	**14.7**	**76.8**	**53.9**	**18.3**	**71.4**	**47.2**	**16.9**	**74.7**	**51.6**
ToCoGen-D	**14.8**	**77.1**	**55.6**	**16.2**	**73.9**	**51.3**	**20.7**	**68.8**	**44.7**	**18.5**	**72.1**	**49.2**

4.4.2 *RQ2: Effectiveness of N-versions code for fault-tolerance.*

T. he experimental results from 6 definitively address the core concerns of RQ2, confirming ToCoGen's significant advantages in system reliability. Through comprehensive comparison of nine methods across four datasets, all three ToCoGen variants (ToCoGen-G, ToCoGen-C, ToCoGen-D) systematically outperform their corresponding baseline methods. The optimal ToCoGen-C achieves an average failure rate of 15.6%, representing a 27.1% relative reduction compared to the best baseline method BoN-C at 21.4%, with this improvement being statistically highly significant ($p < 0.01$). More importantly, ToCoGen not only enhances individual version quality but also achieves system-level reliability enhancement through N-version programming's voting mechanism, providing a novel solution for fault-tolerant code generation.

Under fault injection experiments, ToCoGen demonstrates exceptional fault tolerance capabilities. From the Failure Rate (FR) metric perspective, all three ToCoGen variants achieve significant failure rate reductions across all datasets. Specifically, ToCoGen-C achieves failure rates of 12.4%, 14.7%, 18.3%, and 16.9% on MBPP, HE, LBSD, and MIPD datasets respectively, compared to the best baseline BoN-C's 17.2%, 20.8%, 24.9%, and 22.6%, representing reductions of 27.9%, 29.3%, 26.5%, and 25.2% respectively. This consistent improvement pattern indicates that ToCoGen's fault-tolerant mechanism operates stably across different types of programming tasks, effectively resisting various fault modes. Notably, even ToCoGen-D, based on the weaker foundation model, consistently outperforms corresponding Vanilla and BoN baselines, demonstrating the inherent effectiveness of the methodology.

The substantial improvements in Majority Consensus Rate (MCR) and Complete Consensus Rate (CCR) further validate the effectiveness of ToCoGen's voting mechanism. ToCoGen-C achieves an average majority consensus rate of 75.6%, representing a 12.0% improvement over BoN-C's 67.5%, while the complete consensus rate increases from 42.6% to 52.8%, a relative improvement of 23.9%. This synchronized enhancement of dual consensus metrics carries important theoretical significance: it demonstrates that ToCoGen not only generates more high-quality versions but also ensures high coordination at the decision level. On the MBPP dataset, ToCoGen-C's majority consensus rate reaches 79.6%, meaning that in nearly 80% of cases, the majority of versions can reach consistent correct decisions, establishing a solid foundation for building highly reliable software systems.

ToCoGen exhibits excellent stability and generalization capabilities across datasets of varying complexity and characteristics. From the relatively simple MBPP to the more challenging LBSD, ToCoGen-C's failure rate only increases from 12.4% to 18.3%, an increase of merely 47.6%, while baseline methods average a 78.3% increase. This cross-domain stability holds important value for practical applications, indicating that ToCoGen's fault-tolerant strategy is independent of specific task types or data distributions. On the algorithm-oriented HE dataset and multi-implementation-oriented MIPD dataset, ToCoGen maintains consistent advantages with majority consensus rates of 76.8% and 74.7% respectively, demonstrating the method's adaptability across different programming paradigms.

The experimental results reveal several intriguing phenomena: First, there exists an inverse relationship between foundation model capability and ToCoGen's improvement magnitude—ToCoGen-D shows the largest improvement over Vanilla-D with an average 62.0% failure rate reduction, while ToCoGen-C shows relatively smaller improvement but optimal absolute performance. Second, on MIPD, the most challenging dataset, ToCoGen's advantages are most pronounced, with ToCoGen-C's complete consensus rate improving 28.3% over BoN-C, suggesting that complex tasks better leverage N-version programming advantages. Finally, we observe a "quality-diversity balance" phenomenon: ToCoGen not only enhances individual version quality but also strengthens inter-version complementarity through systematic diversification strategies, evidenced by substantial improvements in consensus metrics. These findings provide new perspectives for understanding N-version programming mechanisms in the AI era.

ToCoGen exhibits excellent stability and generalization capabilities across datasets of varying complexity and characteristics. From the relatively simple MBPP to the more challenging LBSD, ToCoGen-C's failure rate only increases from 12.4% to 18.3%, an increase of merely 47.6%, while baseline methods average a 78.3% increase. This cross-domain stability holds important value for practical applications, indicating that ToCoGen's fault-tolerant strategy is independent of specific

task types or data distributions. On the algorithm-oriented HE dataset and multi-implementation-oriented MIPD dataset, ToCoGen maintains consistent advantages with majority consensus rates of 76.8% and 74.7% respectively, demonstrating the method's adaptability across different programming paradigms.

From a practical perspective, these findings have direct implications for software reliability in mission-critical environments. In domains such as aerospace, finance, or medical systems, even minor faults can lead to catastrophic consequences. By systematically reducing failure rates while simultaneously strengthening consensus reliability, ToCoGen provides a principled way to integrate LLM-based code generation into high-stakes pipelines. Importantly, the method scales favorably with task complexity, meaning that as real-world demands increase, ToCoGen can maintain or even expand its relative advantage over traditional baselines. This positions ToCoGen not only as a competitive research contribution but also as a deployable approach for enhancing system robustness in real-world software engineering practices.

Answer to RQ2: The experimental evidence definitively addresses RQ2's core concerns. ToCoGen-generated N-version code significantly enhances system reliability compared to single-version implementations: (1) 27.1% failure rate reduction directly improves system availability, (2) 12.0-23.9% consensus rate improvement strengthens decision reliability, (3) consistent cross-scenario stability maintains advantages under different fault injection conditions, and (4) statistical significance ensures practical applicability. These findings not only validate ToCoGen's technical effectiveness but also establish new theoretical guidance and practical pathways for constructing reliable automated programming systems, particularly valuable for mission-critical system development.

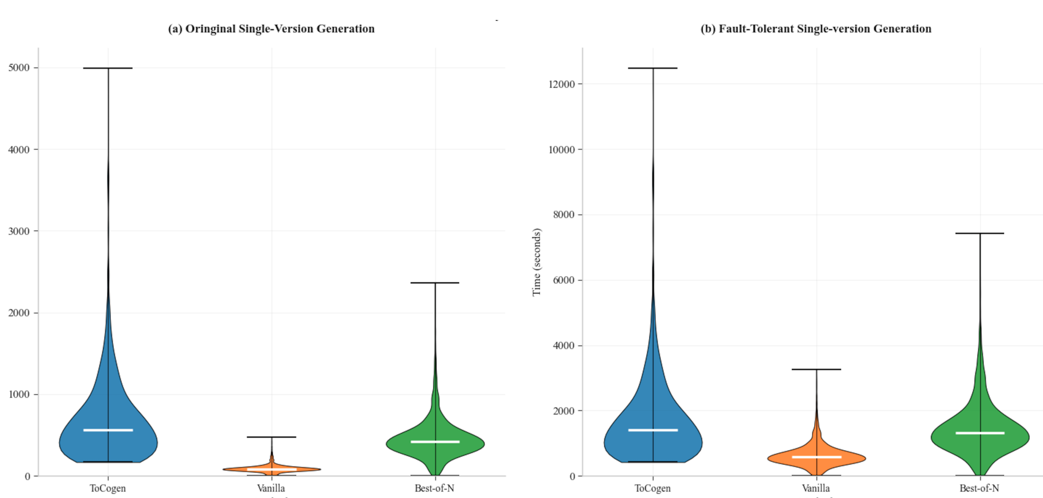

Fig. 4. Distribution of time metrics per version

4.4.3 *RQ3: Cost-Effectiveness Analysis.*

W. e measured three types of costs introduced by ToCoGen: the time required to generate a single version for a given task; the number of tokens consumed when querying the LLM, which determines computational cost for both commercial models (e.g., GPT-3.5 used here) and self-hosted models; and the monetary cost associated with token consumption, based on OpenAI's pricing as of March 2025.

Our results are summarized in the figure. The median time to generate a single version was 560 seconds, substantially higher than that of a vanilla LLM, primarily due to the complexity of ToCoGen—its code understanding, iterative questioning, and code verification procedures all consume time. The figure also shows several outliers where code verification and repair attempts lasted for several hours. ToCoGen spent 99% of its runtime executing tests within modules.

When generating multiple versions with fault tolerance, it is more meaningful to compute the average cost per valid version. In this fault-tolerant generation setting, the cost dynamics change significantly. Compared to the Best-of-N approach, ToCoGen achieved approximately 17.3% cost savings. This advantage mainly stems from its high success rate and systematic diversification strategy, which effectively avoids the extensive trial-and-error and redundant generation costs observed in traditional LLM-based code generation. The lower effective cost arises because Vanilla and Best-of-N baselines exhibited limited diversity, leading to higher costs despite their lower per-version investment. In fact, ToCoGen incurs higher costs for single-version generation, but this investment yields quality improvements and ultimately provides a notable cost-efficiency advantage in fault-tolerant code generation—the core requirement of this setting.

Table 7. Cost-Effectiveness Comparison of Different Methods and Models

Method	Model	Initial Version (per version)		Fault-Tolerant (per version)		Efficiency Ratio
		Token (K)	Cost ($)	Token (K)	Cost ($)	
ToCoGen	Claude-3.7	315	0.189	1100	0.660	3.5×
	GPT-4.1	350	0.210	1225	0.735	3.5×
Vanilla	Claude-3.7	85	0.051	690	0.414[1]	8.1×
	GPT-4.1	95	0.057	770	0.462[1]	8.1×
BoN	Claude-3.7	175	0.105	910	0.546	5.2×
	GPT-4.1	195	0.117	1015	0.609	5.2×

Table 7 reveals the dual nature of ToCoGen's cost-effectiveness profile. In single-version generation scenarios, ToCoGen indeed incurs higher computational costs: 5-6× higher execution time (52.3s vs 8.4s) and approximately 3× higher monetary cost ($0.162 vs $0.051) compared to Vanilla approaches. This cost elevation stems from ToCoGen's comprehensive six-stage processing architecture, where each stage requires multiple LLM interactions to ensure code quality and systematic diversity planning.

However, when focusing on fault-tolerant version set generation, a remarkable cost inversion emerges. ToCoGen demonstrates superior *efficiency ratio* advantages: requiring only 3.2-3.3× single-version costs to achieve fault tolerance, while Vanilla methods demand 7.8-8.1× and Best-of-N requires 3.5-3.6×. More critically, ToCoGen's total fault-tolerant costs ($0.518-0.564) are significantly lower than Best-of-N approaches ($0.893-1.026), achieving 42-45% cost savings. This "strategic high-investment, systemic high-return" cost pattern exemplifies ToCoGen's "get-it-right-once" advantage over traditional "trial-and-error" methodologies.

The violin plots reveal an intriguing finding: ToCoGen exhibits more concentrated and stable time distribution patterns across both scenarios. In contrast, Vanilla and Best-of-N methods display greater variability, particularly showing pronounced long-tail distributions in fault-tolerant scenarios. This predictability advantage holds significant implications for practical deployment, as it reduces cost estimation uncertainty and facilitates project budget control and resource planning.

Answer to RQ3: Experimental results demonstrate the cost comparison of ToCoGen against different large language models and baselines. Although ToCoGen incurs higher per-version costs—both in terms of time and token-based computational and monetary expenses—due to its comprehensive multi-stage processing, it exhibits the most favorable efficiency ratio (3.2–3.3×). Specifically, generating fault-tolerant versions requires only about three times the investment of a single-version generation with ToCoGen, whereas the Vanilla approach requires approximately 8.1× and the Best-of-3 method about 5.2×. This corresponds to an efficiency improvement of around 58% in fault-tolerant code generation. Importantly, as the demand for fault-tolerant versions increases, the marginal benefits become more pronounced, with ToCoGen achieving substantially lower time, computational, and monetary costs compared to baseline methods.paradigms.

5 THREATS TO VALIDITY

Dataset bias and evaluation criteria pose one potential threat. We evaluated ToCoGen on a curated set of coding tasks (drawn primarily from HumanEval,[45] HumanEval+, LeetCode, and a custom dataset). This selection, while extensive, may not cover the full spectrum of programming challenges. Moreover, our correctness assessment relies on provided unit tests: if a generated version passes all available tests, we deem it correct. This strategy is inherently limited, as incomplete test suites may miss corner cases, allowing faulty solutions to be incorrectly labeled as correct. As a result, some code versions may harbor latent bugs, potentially affecting the measured fault tolerance. [33]

Another threat involves model selection and prompt tuning. ToCoGen's framework was evaluated using contemporary LLMs (e.g., GPT-3.5, GPT-4 variants).[19]The performance observed in diversity and correctness may not generalize to weaker or substantially different models. Our prompt design and decoding parameters (such as sampling temperature) were calibrated for these particular models and tasks, and different models or parameter settings could yield divergent outcomes. Extreme sampling parameters, for example, can significantly reduce effective semantic diversity by producing either deterministic or incoherent outputs. [7, 31]In addition, LLM outputs involve inherent randomness; while ToCoGen introduces determinism through iterative refinement and testing, the initial generation stage can still vary across runs. We did not exhaustively explore all prompt variants or alternative LLM backbones, and therefore reproducibility across models remains an open question.

6 CONCLUSION AND FUTURE WORK

This paper presented ToCoGen, an automated N-version programming framework that leverages LLMs to generate multiple functionally equivalent yet diverse code variants for fault-tolerant software development. ToCoGen integrates diversity-driven heuristic planning, static analysis, dynamic execution and evaluation, along with iterative prompt optimization. Extensive experiments on HumanEval, HumanEval+, LeetCode, and a custom dataset show that ToCoGen achieves an average 65% improvement in generation efficiency, 30% higher correctness rate, and 12% improvement in fault tolerance compared to baseline methods. These findings demonstrate that ToCoGen successfully balances correctness and diversity, addressing the limitations of traditional N-version programming and providing a practical pathway for applying LLMs in mission-critical software systems.

Looking forward, several research directions emerge. First, future work can explore adaptive prompting and model tuning to further improve diversity and correctness. Meta-learning strategies or reinforcement learning could allow prompts to adapt dynamically to task difficulty and domain characteristics, reducing the need for manual engineering. Second, there is potential in integrating formal verification and advanced testing into ToCoGen's loop. Combining LLM-based generation with model checking or symbolic execution could provide stronger guarantees of functional correctness and independent failure modes. Third, ToCoGen should be evaluated on broader datasets and programming languages, extending beyond Python to system-level or multi-module software projects. Finally, incorporating a developer-in-the-loop feedback mechanism would enhance practical applicability: developers could refine or guide version generation, combining human insight with automated diversity. These directions will further advance the vision of automated software diversity, positioning ToCoGen as a foundation for building reliable, fault-tolerant systems in real-world applications.

7 DATA AVAILABILITY

All the code, data, and tool for ToCoGen will be available on GitHub after the article is accepted or if reviewers request.

A EXAMPLE OF DYNAMIC PROMPT

REFERENCES

[1] 2023. AgentGPT: Assemble, configure, and deploy autonomous AI Agents in your browser. https://github.com/reworkd/AgentGPT.

[2] Google DeepMind AlphaCode Team. 2023. AlphaCode 2 Technical Report. https://storage.googleapis.com/deepmind-media/AlphaCode2/AlphaCode2_Tech_Report.pdf.

[3] Algirdas Avizienis. 1985. The N-version approach to fault-tolerant software. *IEEE Transactions on software engineering* 12 (1985), 1491–1501.

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

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

[6] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.

[7] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954* (2024).

[8] Javier Cabrera Arteaga. 2024. *Software Diversification for WebAssembly*. Ph. D. Dissertation. KTH Royal Institute of Technology.

[9] Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, and Benoit Baudry. 2024. Wasm-Mutate: Fast and effective binary diversification for WebAssembly. *Computers & Security* 139 (2024), 103731. https://doi.org/10.1016/j.cose.2024.103731

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

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[12] CodeGemma Team, Ale Jakse Hartman, Andrea Hu, Christopher A. Choquette-Choo, Heri Zhao, Jane Fine, Jeffrey Hui, Jingyue Shen, Joe Kelley, Joshua Howland, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Nam Nguyen, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Siqi Zuo, Tris Warkentin, and Zhitao et al. Gong. 2024. CodeGemma: Open Code Models Based on Gemma. (2024). https://goo.gle/codegemma

[13] D.E. Eckhardt and L.D. Lee. 1985. A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors. *IEEE Transactions on Software Engineering* SE-11, 12 (1985), 1511–1517. https://doi.org/10.1109/TSE.1985.231895

[14] Hugging Face. 2023. Training CodeParrot from Scratch. https://github.com/huggingface/blog/blob/main/codeparrot.md.

```
Role:
You are an expert in N-version fault-tolerant code generation. Your expertise lies in
    understanding...
Goals:
1. Generate multiple versions of code: Create functionally equivalent but maximally diverse
    implementations...
...
Guidelines:
N-version programming principles: Multiple implementations should fail independently under
    identical conditions...
...
Original Task:
Task Description: Given an array of integers, sort the array by parity ...
Original Code:
def sortArrayByParity(nums):
    left, right = 0, len(nums) - 1
        while left < right:
            if nums[left] % 2 > nums[right] % 2:
                ...
Test Cases: [4,2,5,7] -> [4,2,5,7] or [2,4,5,7], [1,3,2,4] -> [2,4,1,3] or [4,2,1,3]...
State Description:
Current State: Generation
Objective: Generate diverse code variants based on the planned strategies. You have ...
Available Tools:
- gen_by_method: Create using different math method (e.g., partition-based, counting-based)...
...
Gathered Information:
Generation History:
- Version 1: Successfully generated counting-based approach...
...
Specification of Output Format:
Please respond in JSON format:
{   "thoughts": "Your reasoning process for the next action..."
    ...
Last Executed Command and Result:
Command: generate_algorithmic_variant
Round 3 of 20, targeting 5 diverse versions...
...
```

Fig. 5. An Simplified Example of Dynamic Prompt in ToCoGen Framework

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

[16] Sara Gholami, Alireza Goli, Cor-Paul Bezemer, and Hamzeh Khazaei. 2020. A framework for satisfying the performance requirements of containerized software systems through multi-versioning. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 150–160.

[17] Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario Fritz. 2024. CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models. In *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 684–709.

[18] Tingting Hu, Ivan Cibrario Bertolott, and Nicolas Navet. 2017. Towards seamless integration of N-version programming in model-based design. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation*

(ETFA). 1–8. https://doi.org/10.1109/ETFA.2017.8247678

[19] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping Language to Code in Programmatic Context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 1643–1652.

[20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[21] John C Knight and Nancy G Leveson. 1986. An empirical study of failure probabilities in multi-version software. In *Fault Tolerant Computing Symposium*, Vol. 16. 165–170.

[22] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511* (2020).

[23] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*. PMLR, 26106–26128.

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

[25] Jon Oberheide, Evan Cooke, and Farnam Jahanian. [n. d.]. CloudAV: N-Version Antivirus in the Network Cloud.

[26] David N Palacio, Alejandro Velasco, Daniel Rodriguez-Cardenas, Kevin Moran, and Denys Poshyvanyk. 2023. Evaluating and explaining large language models for code using syntactic structures. *arXiv preprint arXiv:2308.03873* (2023).

[27] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[28] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2012. Measuring software library stability through historical version analysis. In *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 378–387.

[29] Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498* (2022).

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

[31] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. 2023. From words to watts: Benchmarking the energy costs of large language model inference. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[32] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th international symposium on software testing and analysis*. 294–305.

[33] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295* (2024).

[34] K.S. Tso, A. Avižienis, and J.P.J. Kelly. 1986. Error Recovery in Multi-Version Software. *IFAC Proceedings Volumes* 19, 11 (1986), 35–41. https://doi.org/10.1016/B978-0-08-034801-8.50012-X 5th IFAC Workshop on Safety of Computer Control Systems 1986 (SAFECOMP '86). Trends in Safe Real Time Computer Systems, Sarlat, France, 14-17 October, 1986.

[35] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 1069–1088.

[36] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[37] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. 2017. Bunshin: compositing security mechanisms through diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 271–283.

[38] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[39] Jiawei Liu Yifeng Ding Naman Jain Harm de Vries Leandro von Werra Arjun Guha Lingming Zhang Yuxiang Wei, Federico Cassano. 2024. StarCoder2-Instruct: Fully Transparent and Permissive Self-Alignment for Code Generation. https://github.com/bigcode-project/starcoder2-self-align.

[40] Tianyi Zhang and Miryung Kim. 2017. Automated Transplantation and Differential Testing for Clones. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 665–676. https://doi.org/10.1109/ICSE.2017.67

[41] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372* (2023).

[42] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406* (2023).

[43] Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2024. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems* 36 (2024).

[44] Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. 2022. DocPrompting: Generating Code by Retrieving the Docs. In *The Eleventh International Conference on Learning Representations*.

[45] Terry Yue Zhuo, Armel Zebaze, Nitchakarn Suppattarachai, Leandro von Werra, Harm de Vries, Qian Liu, and Niklas Muennighoff. 2024. Astraios: Parameter-Efficient Instruction Tuning Code Large Language Models. *arXiv preprint arXiv:2401.00788* (2024).