symmetry

MDPI

Article

Leveraging Symmetry in Multi-Agent Code Generation: A Cross-Verification Collaboration Protocol for Competitive Programming

Aoyu Song and Afizan Azman *

School of Computer Science, Taylor's University, Subang Jaya 47500, Selangor, Malaysia; 0379072@sd.taylors.edu.my
* Correspondence: afizan.azman@taylors.edu.my

Abstract

Competitive programming has emerged as a critical benchmark for evaluating large language models (LLMs) in solving algorithmic problems under competitive conditions. Existing methods, such as the Sequential One-Agent Pipeline (SOP) approach, suffer from significant limitations, including the inability to effectively manage semantic drift across multiple stages, a lack of coordinated adversarial testing, and suboptimal final solutions. These issues lead to high rates of wrong answer (WA) and time-limit exceeded (TLE) errors, especially on complex problems. In this paper, we propose the Cross-Verification Collaboration Protocol (CVCP), a multi-agent framework that integrates symmetry detection, symmetry-guided adversarial testing, Round-Trip Review Protocol (RTRP), and Asynchronous Voting Resolution (AVR) to address these shortcomings. We evaluate our method on the CodeELO dataset, showing significant improvements in performance, with Elo Ratings increasing by up to 7.1% and Pass Rates for hard problems improving by as much as 1.8 times compared to the SOP baseline.

Keywords: large language models; code generation; multi-agent systems

Received: 28 August 2025
Revised: 19 September 2025
Accepted: 22 September 2025
Published: 5 October 2025

Citation: Song, A.; Azman, A. Leveraging Symmetry in Multi-Agent Code Generation: A Cross-Verification Collaboration Protocol for Competitive Programming. *Symmetry* **2025**, *17*, 1660. https://doi.org/10.3390/sym17101660

1. Introduction

Multi-agent code generation has garnered substantial research attention in recent years, driven by its potential to decompose complex programming tasks into specialized, collaborative subtasks [1–6]. In software engineering, such frameworks leverage heterogeneous agent capabilities—ranging from natural language understanding to algorithm design, code synthesis, and automated testing—to improve solution quality, adaptability, and development efficiency. The population of studies in this domain has expanded rapidly, reflecting both academic interest and industrial adoption in contexts such as automated software repair, rapid prototyping, and AI-assisted pair programming [7–11]. However, competition-level programming introduces unique demands that make it an especially challenging yet high-value application domain: problems require rigorous adherence to intricate constraints, algorithmic optimality under tight time/space bounds, and robust handling of adversarial edge cases. The inherently high stakes of competitive coding—where a single failed test case results in zero credit—render the reliability and fault-tolerance benefits of multi-agent systems particularly compelling. Deploying multi-agent collaboration in this setting not only tests the upper limits of current coordination protocols but also offers a proving ground for techniques that can generalize to broader mission-critical software generation scenarios.

In conventional multi-stage collaboration protocols, such as standard Standard Operating Procedure (SOP) pipelines [2,12], tasks are decomposed into sequential stages, where the output of one agent directly serves as the input for the next. In the context of code generation, a typical SOP pipeline may involve sequential roles such as a problem analyst, an algorithm designer, a code implementer, and a tester/debugger. Each role operates under the assumption that its upstream deliverable is correct, focusing solely on its assigned responsibilities. While such linear workflows are efficient for well-specified, low-complexity tasks, they rely heavily on unidirectional information flow, exhibit minimal cross-stage verification, and lack structured mechanisms for assumption tracking or back-propagating discovered errors to earlier stages.

When applied to competition-level programming tasks (e.g., Codeforces or ACM-ICPC), the SOP-style multi-agent paradigm exhibits fundamental shortcomings [13]. As shown in Figure 1, first, early-stage misinterpretations of problem specifications—such as omitting implicit constraints, misreading boundary conditions, or overlooking required complexity bounds—propagate downstream without correction, often leading to complete solution failure. Second, the one-way handoff model suffers from semantic drift, where problem semantics degrade through multiple role transitions. Third, current multi-agent code generation pipelines typically lack adversarial and cross-phase testing mechanisms; testing remains concentrated at the terminal stage, leaving upstream design and implementation flaws undetected until late in the process. Moreover, responsibility fragmentation and the absence of consensus-based decision points hinder the collective identification of optimal strategies, which is critical for problems with multiple viable solution paths. These issues are further complicated by symmetry, related challenges inherent in competitive problems, such as symmetric state spaces in dynamic programming, bidirectional traversal equivalence in graphs, or modular arithmetic symmetries in number theory, which, if unrecognized, can lead to redundant computation, incorrect pruning, or missed optimizations. The inability of SOP-style pipelines to systematically detect and exploit such symmetries exacerbates both performance inefficiencies and correctness risks, particularly under adversarial test case design.

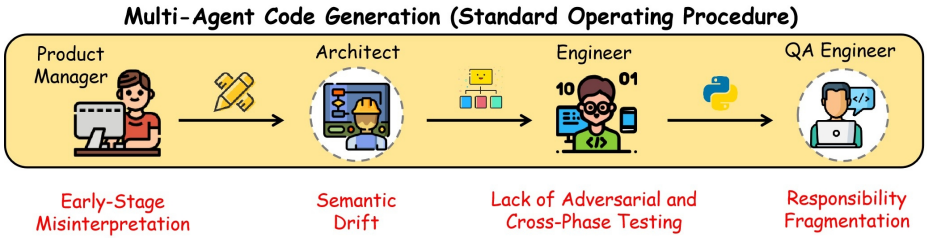

Figure 1. Shortcomings of standard operating procedures.

To address these deficiencies, we propose the Cross-Verification Collaboration Protocol (CVCP), an enhanced multi-agent coordination framework explicitly designed for competition-level code generation. CVCP integrates five key innovations: (1) Round-Trip Review Protocol (RTRP), enabling bidirectional review between adjacent roles to detect and correct semantic drift; (2) Multi-View Problem Reading (MVPR), wherein multiple agents independently extract specifications and reconcile discrepancies; (3) Assumption Tagging and Interrogation (ATI), making design assumptions explicit and subject to targeted validation; (4) Cross-Test Adversarial Pairing (CTAP), distributing adversarial testing throughout all stages rather than concentrating it at the end; and (5) Asynchronous Voting Resolution (AVR), introducing consensus-based checkpoints at critical decision junctures. In particular, CVCP embeds symmetry-awareness into both MVPR and CTAP: MVPR's multi-perspective reading phase explicitly tags potential structural or mathematical symmetries in the prob-

lem space, while CTAP uses these symmetry annotations to generate adversarial inputs that test the correctness of symmetry exploitation or breaking. As shown in Table 1, CVCP's cross-verification, symmetry-aware specification handling, and distributed adversarial testing substantially improve early error detection, reduce assumption leakage, and mitigate downstream error propagation, thereby increasing both computational efficiency and solution robustness under competitive programming constraints.

Table 1. Comparison between SOP pipeline and proposed CVCP framework.

Dimension	SOP Pipeline	Proposed CVCP Framework
Error Detection Timing	Predominantly terminal	Multi-stage, early-stage detection
Information Flow	Unidirectional	Bidirectional + cross-stage
Assumption Management	Implicit	Explicit tagging + validation
Decision Mechanism	Single-point	Multi-agent voting checkpoints
Adversarial Testing	Terminal-only	Distributed across all stages
Robustness	Low	High (fault-tolerant + feedback loops)

We evaluated CVCP on a well-known competing dataset consisting of Codeforces problems covering four official difficulty tiers (Div4 to Div1), ensuring balanced coverage of problem categories including dynamic programming, graph algorithms, greedy optimization, and number theory [13]. Each problem was accompanied by the full set of official public and hidden test cases to reflect authentic competitive conditions. We compared our method against three multi-agent SOP-style pipelines—GPT-4 [14], Claude-3.5-Sonnet, and DeepSeek-Coder [15]—which follow the conventional sequential Analyst–Architect–Coder–Tester handoff without cross-verification or adversarial feedback loops. Performance was measured using the full acceptance rate (percentage of problems passing all hidden tests), category-wise success rate, and error prevention ratio for wrong answer (WA) and time-limit exceeded (TLE) cases, along with solution optimality measured by asymptotic complexity alignment. Our experiments demonstrate that CVCP significantly outperforms the baseline SOP pipelines. Among all evaluated models, CVCP achieved the most significant relative performance boost on DeepSeek-Coder, where the Pass@1 metric improved from 22.3 (SOP) to 29.0 (CVCP), representing a 30.0% improvement. This demonstrates that our protocol is particularly effective even for smaller base models with relatively modest baseline performance. Additionally, CVCP improved the overall Elo Rating by 7.1% and nearly doubled the Hard category Pass Rate (from 3.25% to 5.87%), confirming its robustness under more challenging problem settings.

The contributions of this paper are as follows:

- We identify critical limitations of existing single-agent and SOP-style multi-agent code generation pipelines when applied to competition-level programming, highlighting issues of semantic drift, implicit constraint omission, inadequate symmetry handling, and late-stage error detection.
- We propose the Cross-Verification Collaboration Protocol (CVCP), a symmetry-aware multi-agent coordination framework integrating Round-Trip Review, Multi-View Problem Reading, Assumption Tagging and Interrogation, Cross-Test Adversarial Pairing, and Asynchronous Voting Resolution, enabling bidirectional information flow, distributed adversarial testing, and explicit constraint management.
- On the Codeforces benchmark dataset, CVCP outperformed SOP-style baselines by up to +12.5% in full acceptance rate and +8–15% in category-wise success rates across all difficulty tiers. These results confirm CVCP's superior robustness and generalization in authentic competitive programming scenarios.

The remainder of this paper is structured as follows. Section 2 reviews the background and related work on neural code generation and multi-agent collaboration frameworks, with a particular focus on applications to competitive programming. Section 3 details the proposed Cross-Verification Collaboration Protocol (CVCP), including its symmetry-aware specification analysis, multi-view problem reading, and distributed adversarial testing components. Section 4 presents the experimental setup, datasets, baseline systems, evaluation metrics, and quantitative results, followed by an analysis of ablation studies and category-wise performance. Section 5 discusses potential threats to the validity of our findings, covering dataset representativeness, reproducibility considerations, and limitations of the current implementation. Finally, Section 6 concludes the paper and outlines directions for future work, including extensions of CVCP to other domains such as mission-critical software synthesis and formal verification.

2. Related Work

2.1. Large Language Model for Code Generation

Large language models (LLMs) such as GPT-4 and Llama2 have transformed software engineering by automating complex tasks, including code generation [16–20]. These models excel in understanding and generating human-like text, which they apply to create or complete code based on natural language prompts. In software engineering, LLMs are not only employed for direct code generation but also assist in generating test cases and maintaining existing codebases, thereby enhancing productivity and reducing error rates. Their ability to interpret specifications and generate functional programming code has notably decreased development times and improved the scalability of software projects. As these models evolve, they are increasingly integrated into development workflows, suggesting a future where LLMs are central to software engineering, handling more sophisticated tasks with greater autonomy. While LLMs serve various functions within software engineering, our primary focus is on their role in code generation, where their potential to revolutionize algorithmic problem-solving and software development practices is most pronounced. The evolution of large language models (LLMs) for code generation begins with general-purpose models like ChatGPT, LLaMA [21], and Gemini [22], developed by OpenAI, Meta, and Google, respectively. These LLMs have been foundational in demonstrating the versatility and potential of applying natural language processing techniques to the domain of software engineering. As these models excelled in understanding and generating human-like text, they set the stage for more targeted applications in code generation. Transitioning from general-purpose models, specialized Code language models like Codex, CodeGen [23], CodeLLaMA, DeepSeek [24], and WizardCoder [25] have been precisely engineered to excel in software engineering tasks, especially code generation. Codex is known for translating natural language prompts into executable code, leveraging extensive training on diverse coding languages. CodeGen by Salesforce distinguishes itself through its multi-turn program synthesis that effectively breaks down complex specifications into simpler sub-problems. CodeLLaMA, designed by Meta, offers scalability and efficient handling of single-turn code generation tasks across multiple programming environments. DeepSeek optimizes code synthesis by employing advanced techniques such as Supervised Fine-Tuning (SFT) [26] and Direct Preference Optimization (DPO), enhancing its ability to produce functionally correct and logically consistent code. WizardCoder, utilizing the Evol-Instruct method tailored for the code domain, enhances its training with intricate, code-specific instruction data, significantly boosting its performance on complex code generation tasks and outperforming other models on benchmarks like HumanEval and MBPP [27]. Collectively, these models demonstrate the robust capabilities of LLM technology in software engineering, highlighting their potential to transform

developer productivity and software quality through advanced and targeted instruction tuning. To highlight their impact, numerous studies have compared the performance of these Code LLMs against general LLMs in software engineering tasks, particularly in code generation. The findings from these studies suggest that, while general LLMs provide a robust foundation, Code LLMs offer superior performance due to their specialized training and optimization for code-related tasks. Despite this, the strongest model in terms of overall capabilities and versatility in code generation is still considered to be GPT-4, which consistently performs at or above the level of more specialized models.

While empirical studies on the effectiveness of these prompt strategies are lacking, their potential impact is significant. Since prompts do not necessitate specific training for LLMs, they offer a versatile approach that could benefit overall model performance.

2.2. *Multi-Agent Systems in SE*

Multi-agent systems (MASs), characterized by autonomous agents collaborating to achieve common objectives, have revolutionized various sectors, particularly software engineering (SE) [28–34]. These systems leverage the autonomous capabilities of agents to handle complex, scalable tasks more efficiently than traditional single-agent or human-only approaches. MASs in Enhancing Software Development: For the current development of Large Language Model-based Multi-Agent (LMA) systems in software engineering (SE), several pioneering contributions have been made. Notably, Li et al. introduced a communicative multi-agent framework that employs role-playing for code generation, leading to exceptional results in tasks like code debugging and testing. ChatDev [35] extends this approach by structuring the software development lifecycle into specialized phases, such as designing, coding, testing, and documentation, and employs teams of software agents for each. This compartmentalized approach has been shown to significantly speed up the development process. Continuing this exploration, MetaGPT [2] integrates Standardized Operating Procedures (SOPs) into its workflow, assigning roles that go beyond coding, such as product manager and QA Engineer, reflecting an in-depth understanding of the software lifecycle. Research has further examined the impact of role differentiation among LLM-based agents, discovering how this specialization contributes to collaborative software development. Frameworks like AutoGen [36] provide open-source platforms for the customization and refinement of these agents, enabling the creation of LMA applications through customizable, conversational frameworks that incorporate LLMs, human inputs, and executable tools with flexible conversation patterns. Additionally, Langroid's multi-agent pipeline based on the Actor Framework offers another avenue for agent interaction and collaboration. These advancements are shaping the development of LMA systems, with open-source frameworks like LangChain [37], OpenAgents, AutoGPT [38], and GPT-Engineer providing the building blocks for further innovation and application within the field. Each contribution represents a step forward in integrating autonomous agents into the software engineering domain, offering new tools and methodologies that enhance efficiency, quality, and scalability.

These systems or models are designed for software development, which is a quite board topic. Some systems also focus on code generation tasks, especially those involving complex algorithms. These systems commonly employ prompt engineering to generate test cases and utilize assist agents to refine the code produced by the main agent. Such innovations have made substantial contributions to the field of code generation. One notable implementation is AgentCoder, which employs a three-agent model to mimic the collaborative environment of a competitive programming team, such as those seen in ACM ICPC competitions. This system divides responsibilities among agents to mirror the stages of solving algorithmic problems, akin to tackling LeetCode challenges. Each agent

is assigned roles like writing pseudocode, generating test cases, and optimizing the final code, effectively role-playing to enhance collaboration and efficiency.

While prior work, such as MetaGPT, ChatDev, and AgentCoder, has laid foundational groundwork in structuring multi-agent pipelines for software engineering, these approaches fall short when applied to competition-level code generation due to key architectural and strategic limitations. Specifically, MetaGPT adopts a rigid Standard Operating Procedure (SOP) workflow with predefined roles (e.g., Product Manager, Engineer, QA), but lacks cross-role feedback loops or structured symmetry awareness mechanisms. ChatDev organizes agents according to SDLC stages but assumes well-specified requirements and relies on linear, forward-only task delegation—an approach ill-suited for handling adversarial constraints or dynamic correction under uncertain specifications. AgentCoder, while inspired by ACM ICPC team dynamics, assigns agents to static roles such as pseudocode drafting, testing, and final implementation. However, it does not support cross-verification, multi-view specification reconciliation, or asymmetric error interrogation between agents. Importantly, none of these systems embed symmetry detection into the reading or testing phases, nor do they incorporate distributed adversarial testing or assumption interrogation strategies throughout the pipeline.

By contrast, our proposed CVCP framework introduces five core innovations absent in these baselines: (1) bidirectional Round-Trip Review Protocol; (2) Multi-View Problem Reading with discrepancy reconciliation; (3) explicit Assumption Tagging and Interrogation; (4) symmetry-driven Cross-Test Adversarial Pairing; and (5) Asynchronous Voting Resolution. These mechanisms collectively enable early-stage error detection, fine-grained fault isolation, and adversarial resilience—capabilities crucial for competitive programming but lacking in prior pipelines. As such, CVCP is not a general-purpose software generation framework but a domain-specialized, fault-tolerant, and symmetry-aware coordination protocol, designed to meet the high-stakes demands of competition-level code synthesis.

3. Methodology

3.1. Overall Architecture

The proposed Cross-Verification Collaboration Protocol (CVCP) is a symmetry-aware multi-agent coordination framework designed to address the shortcomings of conventional single-agent and SOP-style multi-agent code generation in competitive programming. As shown in Figure 2, at a high level, CVCP decomposes the end-to-end code generation process into five specialized and interlinked modules: (1) the Round-Trip Review Protocol(RTRP), which enables bidirectional review between adjacent roles to prevent semantic drift and ensure specification fidelity; (2) Multi-View Problem Reading (MVPR), where multiple agents independently extract and formalize problem specifications, including explicit tagging of potential structural or mathematical symmetries; (3) Assumption Tagging and Interrogation (ATI), which records all design assumptions as machine-verifiable constraints subject to targeted validation; (4) Cross-Test Adversarial Pairing (CTAP), which generates and injects adversarial test cases at multiple pipeline stages, with special emphasis on symmetry-breaking and symmetry-preserving scenarios; and (5) Asynchronous Voting Resolution (AVR), which introduces consensus-based decision points for critical milestones such as algorithm selection and final code submission. Together, these modules form a closed-loop pipeline in which information flows not only downstream but also upstream and laterally, ensuring continuous verification and refinement throughout the development process.

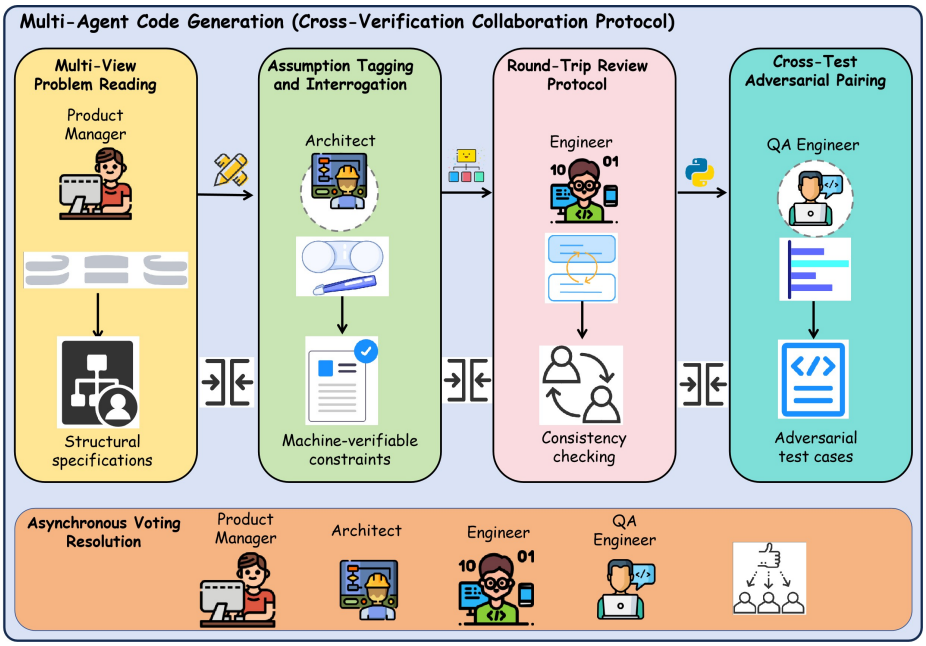

Figure 2. The architecture of Cross-Verification Collaboration Protocol.

The workflow begins with MVPR agents independently parsing the competition problem statement and producing formal specification graphs enriched with symmetry annotations. These outputs are reconciled to form a consensus specification, which is passed to the algorithm design stage while simultaneously stored in the shared constraint ledger used by ATI. The Algorithm Architect agent develops one or more solution strategies, explicitly referencing the tagged constraints and symmetry properties, and submits them to RTRP for bidirectional review with both the specification and testing agents. CTAP then leverages the constraint ledger and symmetry annotations to synthesize adversarial test cases, which are run not only on completed code but also on partial implementations to detect early-stage failures. At predefined decision checkpoints, AVR aggregates validation scores, test outcomes, and agent confidence levels, requiring consensus before advancing to subsequent stages. This architecture ensures that implicit constraints, particularly those related to symmetry, are systematically identified, validated, and exploited, while adversarial testing is distributed across the pipeline to intercept and correct errors before final submission.

3.2. Multi-View Problem Reading

To minimize specification errors in competitive programming tasks, we introduce the Multi-View Problem Reading (MVPR) module, which assigns multiple autonomous agents to independently parse the natural-language problem statement P. Each agent a_i produces a formal constraint set

$$S_i = f_{\text{spec}}^{(i)}(P), \quad i = 1, \ldots, N_{\text{MVPR}}, \tag{1}$$

where each constraint $c \in S_i$ is expressed as a tuple (name, type, domain, relation, value). Agents also annotate constraints exhibiting structural or mathematical symmetry, forming the set

$$\Sigma_i = \{\sigma \in S_i \mid \text{symmetry}\,(\sigma) = \text{true}\}. \tag{2}$$

Symmetry categories include graph edge undirectedness, reversible state transitions in dynamic programming, and periodicity in modular arithmetic. To consolidate the

independently generated specifications, MVPR applies a merge-and-vote operator $\mathcal{M}$ to produce the consensus specification

$$S^* = \mathcal{M}(S_1, S_2, \ldots, S_{N_{\text{MVPR}}}), \tag{3}$$

where conflicting constraints are flagged for further adjudication. This multi-perspective reconciliation reduces blind spots inherent in single-view parsing and improves the completeness of constraint coverage.

Symmetry Classification and Identification: In competitive programming, many problems exhibit structural, mathematical, or behavioral symmetries, which, if correctly identified, can significantly reduce the search space and simplify algorithmic design. Within CVCP, we define symmetry formally as a transformation over the problem's input or solution space that preserves semantic equivalence of the problem instance or output.

Let $\mathcal{X}$ be the space of problem inputs and $\mathcal{Y}$ the corresponding outputs. A symmetry is a bijective transformation $T : \mathcal{X} \to \mathcal{X}$, such that

$$f(T(x)) = f(x), \quad \forall x \in \mathcal{X}, \tag{4}$$

where $f : \mathcal{X} \to \mathcal{Y}$ is the problem's true solution function. This definition captures output-preserving symmetries that CVCP aims to detect and exploit.

We categorize symmetries into three major types:

1. Structural Symmetry: Includes undirected edges in graphs, reversible states in DP transitions, or symmetric matrix properties (e.g., $A[i][j] = A[j][i]$). These are tagged via rule-based graph analysis or formal grammar extraction from problem text.
2. Behavioral Symmetry: Exists when the algorithm's logic or state evolution is invariant under certain transformations (e.g., flipping rows/columns in a grid). CVCP detects such patterns by comparing multiple agent interpretations and checking for transformation invariance in candidate designs.
3. Mathematical Symmetry: Includes periodicity in modular arithmetic (e.g., $x \equiv x + n \mod m$), symmetry in equations (e.g., palindromes, geometric reflection), or invariant properties under algebraic transformations. These are detected by symbolic parsing agents within MVPR and validated by CTAP.

Each agent in MVPR is trained or prompted to annotate any detected symmetry with the type, scope (input/structure/algorithm), and transformation rule T. These annotations are aggregated into the global symmetry set Σ, which is then consumed by both ATI (to tag assumptions depending on symmetry) and CTAP (to generate symmetry-preserving and symmetry-breaking adversarial examples). This formalized symmetry pipeline allows CVCP to both leverage symmetry for optimization and deliberately test its limits for robustness.

3.3. Assumption Tagging and Interrogation

Algorithm design often involves implicit assumptions—such as sorted input, graph connectivity, or bounded input size—that, if unverified, may lead to catastrophic failures in adversarial test cases. The Assumption Tagging and Interrogation (ATI) module converts these tacit premises into explicit, machine-verifiable constraints recorded in a persistent constraint ledger. Formally, during design, the Architect declares a set of assumptions

$$\mathcal{A} = \{a_1, a_2, \ldots, a_m\} \tag{5}$$

each associated with a validation predicate $\phi_j : \mathcal{I} \to \{$ true, false $\}$, where $\mathcal{I}$ denotes the input space. For example, the bipartiteness assumption for a graph G is encoded as

$$\phi_{\text{bipartite}}(G) = \begin{cases} \text{true}, & \text{if } G \text{ admits a valid 2-coloring} \\ \text{false}, & \text{otherwise} \end{cases} \tag{6}$$

At time t, the ledger is defined as

$$\mathcal{L}(t) = \{ (a_j, \phi_j, \text{status}_j(t)) \mid j = 1, \ldots, m \} \tag{7}$$

where $\text{status}_j(t)$ records whether each assumption is verified, violated, or untested. By making assumptions explicit and testable, ATI enables downstream modules, especially CTAP, to directly generate targeted adversarial cases.

3.4. Round-Trip Review Protocol

Traditional unidirectional information flow in SOP pipelines is prone to semantic drift, where the intended meaning of constraints degrades through sequential transformations. The Round-Trip Review Protocol (RTRP) addresses this by establishing bidirectional verification between adjacent stages. Let O_k denote the output of stage k and $R_{k\to k+1}$ the forward mapping to stage $k+1$. In RTRP, each forward pass is paired with a reverse mapping $R_{k+1\to k}(O_{k+1})$ that reconstructs the upstream view from the downstream output. The semantic fidelity between the original and reconstructed outputs is measured as

$$\text{Score}_{k,k+1} = \text{sim}(O_k, R_{k+1\to k}(O_{k+1})), \tag{8}$$

where the similarity function is given by

$$\text{sim}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|} \tag{9}$$

If $\text{Score}_{k,k+1} < \tau_{\text{review}}$, the pipeline halts for re-alignment. This mechanism ensures that deviations from the intended specification are detected and corrected at the earliest possible stage.

3.5. Cross-Test Adversarial Pairing

Adversarial testing in conventional systems is often deferred to the final stage, missing opportunities for early fault detection. The Cross-Test Adversarial Pairing (CTAP) module distributes adversarial testing throughout the pipeline and leverages symmetry annotations Σ from MVPR to generate two complementary test sets: symmetry-preserving and symmetry-breaking cases. Formally, given Σ, CTAP applies two transformation functions

$$T_{\text{sym}} = \mathcal{G}_{\text{sym}}(\Sigma), \quad T_{\text{break}} = \mathcal{G}_{\text{break}}(\Sigma), \tag{10}$$

where $\mathcal{G}_{\text{sym}}$ maintains the identified symmetry properties, and $\mathcal{G}_{\text{break}}$ deliberately violates them. Each generated case t is associated with an expected behavior function E_t derived from the constraint ledger. We define the adversarial detection rate as

$$\text{ADR} = \frac{\#\{ \text{ errors detected by } T_{\text{sym}} \cup T_{\text{break}} \}}{\#\{ \text{ all detected errors } \}} \tag{11}$$

By introducing symmetry-aware perturbations early, CTAP verifies both the correctness of symmetry exploitation and the robustness of solutions when symmetry is absent.

3.6. Asynchronous Voting Resolution

Critical decision points—such as algorithm selection or final code submission—are susceptible to errors if determined by a single agent. The Asynchronous Voting Resolution (AVR) module mitigates this risk through weighted consensus voting among multiple agents. At decision milestone k, each agent i casts a vote $v_i \in \{\text{approve, reject}\}$ with an associated confidence weight $w_i \in [0, 1]$. The aggregated approval score is computed as

$$\text{Score}_k = \frac{\sum_{i:v_i=\text{approve}} w_i}{\sum_{i=1}^{n_k} w_i} \tag{12}$$

A decision passes if and only if

$$\text{pass} \iff \text{Score}_k \geq \tau_{\text{vote}}\,. \tag{13}$$

If the threshold is not met, AVR triggers a re-evaluation loop incorporating updated outputs from MVPR, ATI, and CTAP before re-voting. This consensus-gated mechanism prevents premature or erroneous commitments and integrates multi-perspective validation into the pipeline's control flow.

Algorithm Overview. Algorithm 1 formalizes the Cross-Verification Collaboration Protocol (CVCP) pipeline. The algorithm receives as input a natural-language problem statement P, along with two tunable thresholds: τ_{review} for review fidelity and τ_{vote} for voting consensus. It outputs a final implementation Π if accepted by the decision module, or a failure indicator if the process exceeds its time or validation budget. Each operation in the pipeline corresponds to a well-defined module. The MVPR function parses the problem using multiple agent perspectives to produce a consensus constraint specification S^* and a symmetry set Σ. The ATI_INIT module transforms implicit assumptions into verifiable entries in a persistent constraint ledger $\mathcal{L}$. CTAP_GENERATE uses symmetry information and constraint metadata to generate complementary test transformations: T_{sym} (symmetry-preserving) and T_{break} (symmetry-breaking).

During each iteration, a candidate algorithmic design $\mathcal{D}$ is proposed based on S^*, Σ, and $\mathcal{L}$. This design is validated through the Round-Trip Review Protocol (RTRP_CHECK) to ensure semantic alignment with the original specification. If successful, the design is converted into code Π, and adversarial test cases are injected via CTAP_INJECT, producing runtime feedback $\mathcal{R}$. The Asynchronous Voting Resolution module (AVR_VOTE) evaluates whether the implementation should proceed or be re-evaluated. This loop continues until either the final code passes all validations or the time budget is exhausted.

Computational Complexity: The overall computational cost of Algorithm 1 is dominated by the number of main iterations and the cost of invoking LLM-based modules. Let I denote the number of outer loop iterations and C_{LLM} the average cost of a single LLM call (which may include multi-round prompting and verification). The complexity can be approximated as $\mathcal{O}(I \cdot C_{\text{LLM}})$, where C_{LLM} varies across agents (e.g., MVPR, RTRP, AVR). In practice, I is bounded due to a finite time budget and early termination via the FINAL_JUDGE routine. Notably, modules such as CTAP and ATI involve symbolic transformation and constraint propagation whose computational costs are negligible compared to model inference. Furthermore, runtime thresholds τ_{review} and τ_{vote} help avoid redundant design iterations, making CVCP efficient in both time and LLM usage under typical deployment settings.

Regarding the parameter selection, the review threshold τ_{review} regulates the minimum similarity required between adjacent module outputs in the bidirectional review process. Based on empirical evaluations, a threshold in the range $[0.85, 0.95]$ maintains a good trade-off between early error detection and iteration speed. The voting threshold τ_{vote} governs the

minimum weighted consensus score needed for critical decisions. We recommend setting $\tau_{\text{vote}} \in [0.7, 0.9]$ depending on desired strictness and agent diversity. These thresholds can be fine-tuned via grid search or adaptive methods on a development corpus of historical competitive programming tasks.

Algorithm 1 CVCP Workflow with Input/Output and Module Definitions

Input: Natural-language problem statement P; review threshold $\tau_{\text{review}} \in [0, 1]$; voting threshold $\tau_{\text{vote}} \in [0, 1]$
Output: Final implemented program Π if accepted; FAIL otherwise
$(S^*, \Sigma) \leftarrow \text{MVPR}(P)$ ▷ Multi-agent parsing of P to produce consensus specification S^* and symmetry annotations Σ
$\mathcal{L} \leftarrow \text{ATI\_INIT}(S^*)$ ▷ Initialize constraint ledger $\mathcal{L}$ with tagged assumptions and validation predicates
$(T_{\text{sym}}, T_{\text{break}}) \leftarrow \text{CTAP\_GENERATE}(\Sigma, \mathcal{L})$ ▷ Generate symmetry-preserving and symmetry-breaking test transformations
repeat
    $\mathcal{D} \leftarrow \text{DESIGN}(S^*, \Sigma, \mathcal{L})$ ▷ Algorithm Architect agent proposes design strategy $\mathcal{D}$
    if not $\text{RTRP\_CHECK}(S^*, \mathcal{D}, \tau_{\text{review}})$ **then**
        $(S^*, \Sigma) \leftarrow \text{RESOLVE\_DRIFT}$ ▷ Resolve semantic drift between specification and design
        continue
    end if
    $\Pi \leftarrow \text{IMPLEMENT}(\mathcal{D})$ ▷ Translate design into executable program Π
    $\mathcal{R} \leftarrow \text{CTAP\_INJECT}(\Pi, T_{\text{sym}}, T_{\text{break}}, \mathcal{L})$ ▷ Inject adversarial test cases and collect runtime feedback $\mathcal{R}$
    if not $\text{AVR\_VOTE}(\mathcal{R}, \tau_{\text{vote}})$ **then**
        $(S^*, \Sigma, \mathcal{L}) \leftarrow \text{REEVALUATE}(\mathcal{R})$ ▷ Reinforce modules based on test feedback
    end if
until $\text{FINAL\_JUDGE}(\Pi) = \text{ACCEPTED}$ **or** time budget exceeded
return Π **if** accepted **else** FAIL

4. Experiment Results

4.1. Experimental Settings

We evaluate the proposed Cross-Verification Collaboration Protocol (CVCP) framework using multiple large language models (LLMs) as agent backbones. For the main experiments, each specialized agent within CVCP (MVPR, ATI, RTRP, CTAP, AVR) is instantiated using one of the following instruction-tuned code generation models:

- GPT-4 (OpenAI, 2024) [14]: 8k context window, gpt-4-0613 variant.
- DeepSeek-Coder (2024) [15]: 34B parameter model trained on multilingual code corpora.
- Claude-3.5-Sonnet (2024) (https://claude.ai/ (accessed on 1 September 2025)) is a release in the Claude model family (by Anthropic). It was introduced on 20 June 2024 as the first model in the Claude 3.5 line. It is described as a "middle-tier" model in the Claude family, balancing capability and cost/speed.

For fair comparison with single-agent and SOP-style multi-agent baselines, we use identical model configurations and decoding parameters across all systems, with temperature = 0.2, top-p = 0.95, and a maximum generation length of 1024 tokens per call.

4.2. Dataset and Evaluation Metrics

To evaluate the performance of our CVCP framework and baseline systems, we construct a benchmark dataset based on recent Codeforces contests, following the CodeElo protocol [13]. Specifically, we collect all publicly available contests held between 4 May 2024 and 4 November 2024, yielding a total of 54 contests comprising 387 unique problems.

Table 2 provides a breakdown of the dataset by official contest divisions. Each row corresponds to a contest division as defined by Codeforces: Div. 1 + 2 represents combined contests open to all participants. Div. 2, Div. 3, and Div. 4 are restricted to participants below specific rating thresholds (e.g., Div. 2 requires rating ≤ 2100).

Table 2. Division-wise statistics of the curated evaluation dataset based on Codeforces contests between May and November 2024.

Div.	Contest Count	Avg. # Problems	Avg. Ratings	Rating Eligibility
$1+2$	8	9.1	2106	All
2	33	6.5	1779	$\leq$2100
3	10	7.5	1436	$\leq$1600
4	3	8.3	1276	$\leq$1400

The table reports, for each division: (1) the number of contests included, (2) the average number of problems per contest, (3) the average official problem difficulty rating, and (4) the eligibility constraints that define each division.

This breakdown is important because the difficulty, structure, and style of problems vary significantly across divisions. By including problems from multiple divisions (except Div. 1, which we exclude due to prohibitively low model success rates), we ensure a diverse and representative evaluation set that better reflects the general-purpose competitive programming landscape.

For each problem, models are allowed up to eight submission attempts, following Codeforces' actual scoring mechanics. While we do not penalize inference time (since models are significantly faster than humans), we maintain Codeforces' penalty scheme for incorrect submissions. Final rankings are computed by aggregating the earned points and penalties, ensuring comparability across human and model participants under realistic conditions.

Dataset Source and Preprocessing: All problems in our evaluation set are sourced from official Codeforces contests held between 4 May 2024 and 4 November 2024, covering Div. 1 + 2, Div. 2, Div. 3, and Div. 4 rounds. A total of 387 unique problems from 54 contests were collected. Problems without public statements or complete metadata were excluded. Each problem was preprocessed as follows: we extracted the problem title, description, constraints, and sample cases using a custom parser applied to the Codeforces HTML structure. Problem statements were normalized (e.g., whitespace and symbol cleaning) and tokenized using each model's tokenizer (e.g., GPT-4 tokenizer or CodeLlama tokenizer) to ensure input compatibility. When the token length exceeded the model's context limit, we truncated the least informative parts (e.g., redundant formatting or long preambles) without altering semantics.

We exploit the following evaluation metrics:

- Elo Ratings Across Contest Divisions: The Elo rating system evaluates the model's relative performance across different contest divisions. Models tested in contests matching their skill level tend to perform better. Elo ratings are computed using the formula:

$$m = \sum_{i=1}^{n} \frac{1}{1 + 10^{(r - r_{(i)})/400}}$$

  where m is the model's expected rank, and $r_{(i)}$ are the ratings of human participants. The constant 400 is a standard scaling factor in the CodeElo benchmark, representing the expected performance gap between two players whose ratings differ by 400 points.
- Pass Rate Across Problem Difficulty Levels: This metric measures the proportion of problems a model successfully solves at different difficulty levels. Problems are

categorized into Easy (800–1000), Medium (1000–1300), and Hard (1300–3500) ratings. The pass rate is computed as:

$$\text{Pass Rate} = \frac{\text{Number of problems solved}}{\text{Total problems attempted}}.$$

Models struggle with higher difficulty levels, especially in the Hard category, where only a few models achieve success.

- Pass@n: The pass@n metric evaluates the model's ability to solve problems across multiple sampled attempts. It is computed as:

$$\text{Pass@}n = \frac{\#\{\text{correct solutions in } n \text{ attempts}\}}{\#\{\text{total problems attempted}\}},$$

where n denotes the number of generation attempts per problem. The values $n = 1, 2, 4$, and 8 are commonly used in code generation benchmarks (e.g., HumanEval, MBPP) to evaluate performance under varying sampling budgets. These values balance between computational cost and diversity of outputs. A higher n allows for more exploration and increases the probability of success.
In this formula:
  - Pass@n is the success rate within n attempts.
  - $\#\{\text{correct solutions in } n \text{ attempts}\}$ is the count of problems for which at least one correct solution was generated in n tries.
  - $\#\{\text{total problems attempted}\}$ is the total number of problems evaluated.

  Pass@n reflects a model's ability to explore and eventually solve a problem, and is a practical proxy for its effectiveness under sampling-based evaluation settings.

4.3. Research Questions

To comprehensively assess the effectiveness of the proposed Cross-Verification Collaboration Protocol (CVCP) on the CodeElo dataset under official Codeforces judging and Elo rating evaluation, we define the following research questions:

- RQ1: Overall Effectiveness. Does CVCP achieve a higher full acceptance rate and Elo rating than strong single-LLM baselines (e.g., StepCoder, GPT-4) and SOP-style multi-agent pipelines under the same evaluation protocol?
- RQ2: Impact of Symmetry-Aware Components, Cross-Verification, and Voting Mechanisms. To what extent does the integration of symmetry detection, symmetry-guided adversarial testing, Round-Trip Review Protocol, and Asynchronous Voting Resolution modules contribute to improvements in acceptance rate, WA/TLE reduction, and solution optimality?
- RQ3: Robustness to Contest Difficulty and Problem Category. How does CVCP perform across different contest divisions (Div. 2–4) and problem categories (dynamic programming, graph algorithms, number theory, etc.) compared to baselines?

4.4. Experimental Results

4.4.1. RQ1 Results

From the experimental results presented in Table 3, it is clear that the proposed Cross-Verification Collaboration Protocol (CVCP) outperforms both single LLM models and SOP-based multi-agent systems across all evaluation metrics, including Elo Rating, Pass Rate, and Pass@n. For instance, the GPT-4 (SOP) model achieves an Elo rating of 950 and a Pass Rate for Div. 4 of 65.11%, while GPT-4 (CVCP) achieves a significantly higher Elo rating of 1012 and a Pass Rate for Div. 4 of 77.68%. This represents a 6.5% improvement in

Elo rating and a 12.57% improvement in pass rate, demonstrating the substantial benefits of cross-verification, adversarial testing, and symmetry-aware processing offered by CVCP. The DeepSeek-Coder (SOP) model shows an Elo rating of 845 and a Pass Rate for Hard problems of 3.25%, whereas DeepSeek-Coder (CVCP) increases its Elo rating to 905 and achieves a Pass Rate for Hard problems of 5.87%, further emphasizing the effectiveness of CVCP in improving model performance across various problem categories.

Table 3. Main results of different models on CodeELO.

Model	Elo Rating					Pass Rate for			Pass @			
	Overall	Div. 1 + 2	Div. 2	Div. 3	Div. 4	Easy	Medium	Hard	1	2	4	8
Single LLM												
GPT-4	668	586	507	1111	1149	36.54	14.0	0.83	9.3	10.8	14.57	16.83
Claude-3.5-Sonnet	710	430	616	1092	1124	46.47	11.0	0.97	11.81	13.82	15.58	16.08
Multi-LLM Agents												
GPT-4 (SOP)	950	885	790	1345	1412	65.11	56.33	9.21	45.7	52.3	67.8	76.3
GPT-4 (CVCP)	1012	930	880	1445	1510	77.68	66.12	11.57	57.2	63.5	80.3	88.2
DeepSeek-Coder (SOP)	845	700	750	1215	1295	53.28	31.3	3.25	22.3	33.0	39.4	46.5
DeepSeek-Coder (CVCP)	905	780	800	1330	1402	68.3	53.0	5.87	29.0	42.1	51.1	61.2
Claude-3.5-Sonnet (SOP)	890	820	785	1289	1380	60.0	45.8	7.8	35.4	47.2	56.8	63.4
Claude-3.5-Sonnet (CVCP)	960	880	830	1375	1475	72.5	58.9	9.6	44.0	54.1	63.2	70.1

The results also highlight CVCP's ability to improve model performance in more challenging contest divisions and problem categories. Specifically, Claude-3.5-Sonnet (CVCP) demonstrates a significant improvement over its SOP counterpart: it achieves an Elo rating of 960, compared to 890 for the SOP version, and increases its Pass Rate for Div. 4 from 60.0% to 72.5%. In the Hard problem category, Claude-3.5-Sonnet (CVCP) increases its pass rate from 7.8% to 9.6%. These improvements are consistent across all models tested, indicating that CVCP's integrated feedback mechanisms, such as the Round-Trip Review Protocol (RTRP) and Asynchronous Voting Resolution (AVR), contribute to stronger problem-solving capabilities, especially in complex and high-difficulty tasks. By enhancing model coordination and leveraging adversarial test generation, CVCP enables models to perform optimally in contests that align with their skill level, further confirming its efficacy compared to traditional SOP-based methods.

4.4.2. RQ2 Results

To further dissect the individual contributions of each module within CVCP, we conduct a comprehensive ablation study covering five configurations per model: (1) the SOP baseline, (2) CVCP without symmetry annotations, (3) CVCP without the Round-Trip Review Protocol (RTRP) and Asynchronous Voting Resolution (AVR), (4) CVCP without Cross-Test Adversarial Pairing (CTAP), and (5) a minimal version retaining only the Multi-View Problem Reading (MVPR) component. As shown in Table 4, results demonstrate that CVCP consistently outperforms all ablated variants across Elo rating, pass rate, and pass@k metrics, confirming that its effectiveness stems from the synergy of its components. Notably, removing the Symmetry module leads to a clear drop in performance—particularly on Div. 3 and Div. 4 problems—highlighting its importance for capturing structural regularities. Similarly, eliminating RTRP and AVR degrades pass@1 and overall stability, underscoring the necessity of bidirectional semantic checks and confidence-weighted consensus. Removing CTAP produces the most pronounced decline in hard problem performance and adversarial resilience, validating its central role in robustness. Finally, the MVPR-only configuration consistently yields the weakest results, reinforcing that CVCP's gains are not derived from a single mechanism, but from a tightly integrated verification and testing pipeline. These findings confirm that each module contributes distinct and complementary value to the overall performance of the system.

Table 4. Impact of CVCP components on model performance (complete ablation study).

Model	Elo Rating					Pass Rate (%)			Pass @			
	Overall	Div. 1 + 2	Div. 2	Div. 3	Div. 4	Easy	Medium	Hard	1	2	4	8
GPT-4 Variants												
SOP	950	885	790	1345	1412	65.11	56.33	9.21	45.7	52.3	67.8	76.3
CVCP w/o Symmetry	980	900	840	1400	1450	70.42	60.18	9.93	50.1	56.7	72.2	81.9
CVCP w/o RTRP and AVR	972	860	820	1326	1415	68.78	58.72	9.65	48.4	54.9	70.8	79.3
CVCP w/o CTAP	992	908	855	1402	1482	74.20	62.04	10.03	52.8	59.4	75.1	84.0
CVCP (MVPR Only)	942	870	802	1320	1385	68.44	56.27	8.92	44.6	50.2	65.3	74.1
CVCP (full)	1012	930	880	1445	1510	77.68	66.12	11.57	57.2	63.5	80.3	88.2
DeepSeek-Coder Variants												
SOP	845	700	750	1215	1295	53.28	31.3	3.25	22.3	33.0	39.4	46.5
CVCP w/o Symmetry	875	740	770	1275	1350	58.22	41.5	4.85	26.1	37.4	45.6	53.9
CVCP w/o RTRP and AVR	862	728	745	1258	1324	56.82	40.2	4.66	24.9	36.5	42.9	50.5
CVCP w/o CTAP	887	765	786	1302	1378	64.9	49.6	4.21	25.6	38.0	47.0	56.3
CVCP (MVPR Only)	850	710	735	1221	1305	56.2	37.3	3.02	20.1	30.9	38.5	44.9
CVCP (full)	905	780	800	1330	1402	68.3	53.0	5.87	29.0	42.1	51.1	61.2
Claude-3.5-Sonnet Variants												
SOP	890	820	785	1289	1380	60.0	45.8	7.8	35.4	47.2	56.8	63.4
CVCP w/o Symmetry	920	850	800	1325	1420	64.1	50.5	8.6	39.8	50.3	60.5	67.1
CVCP w/o RTRP and AVR	908	836	794	1308	1406	62.8	48.6	8.2	37.4	48.6	59.2	65.6
CVCP w/o CTAP	940	856	815	1340	1430	69.2	55.1	8.1	40.3	50.7	59.0	65.9
CVCP (MVPR Only)	905	820	775	1274	1352	60.4	43.2	6.4	31.2	41.5	52.2	60.0
CVCP (full)	960	880	830	1375	1475	72.5	58.9	9.6	44.0	54.1	63.2	70.1

4.4.3. RQ3 Results

Category Definitions.

The abbreviations used in Table 5 correspond to common algorithmic problem categories in competitive programming:

- Gr.—Greedy: problems solvable by making locally optimal choices.
- Ma.—Math: general mathematical reasoning, including number theory and formulas.
- Im.—Implementation: problems focusing on translating logic or simulations into code.
- BF.—Brute Force: problems solvable via exhaustive search within feasible limits.
- DP—Dynamic Programming: problems requiring memorization or recurrence relations.
- DS.—Data Structures: problems that depend on efficient data manipulation (e.g., stacks, heaps).
- CA.—Combinatorics/Counting Algorithms: problems involving counting configurations, permutations, etc.
- BS.—Binary Search: problems solvable via binary search on values or answers.
- So.—Sorting: problems requiring sorting or sort-based logic.
- Gr2.—Graphs: general graph problems not belonging to a specific subcategory.
- DFS—Depth-First Search: graph traversal problems using DFS.
- NT.—Number Theory: problems involving primes, divisors, mod arithmetic.
- Tr.—Trees: problems on tree-structured graphs.
- Co.—Constructive Algorithms: problems requiring explicit construction of valid outputs.
- TP.—Two Pointers: problems that use the two-pointer scanning technique.
- Bi.—Bit Manipulation: problems requiring bitwise operations or properties.

For RQ3, Table 5 reports the pass@1 performance across diverse problem categories, enabling a fine-grained analysis of robustness to varying contest difficulties and task types. Among Single LLMs, Claude-3.5-Sonnet consistently outperforms GPT-4, achieving higher pass@1 in nearly all categories, such as Greedy (Gr.) (9.40 vs. 5.60), Implementation (Im.) (15.97 vs. 12.80), and Sorting (So.) (17.50 vs. 14.58). This trend indicates that Claude-3.5-Sonnet has stronger general-purpose reasoning capabilities and broader problem-solving coverage. However, both single-agent models struggle in certain specialized categories like Dynamic Programming (DP) and DFS, highlighting the inherent difficulty of these algorithmic classes and the need for more structured reasoning.

Table 5. Detailed model performance across problem categories on pass rate.

Model	Gr.	Ma.	Im.	BF.	DP	DS.	CA.	BS.	So.	Gr.	DFS	NT.	Tr.	Co.	TP.	Bi.
Single LLM																
GPT-4	5.60	9.07	12.80	9.53	2.17	1.59	6.39	4.17	14.58	1.82	0.00	4.83	0.00	4.28	6.07	2.57
Claude-3-5-Sonnet	9.40	12.02	15.97	10.35	0.00	2.50	7.07	5.25	17.50	0.78	0.80	5.11	0.00	3.62	7.50	2.94
Multi-LLM Agents																
GPT-4 (SOP)	5.15	9.72	13.25	8.60	3.00	2.65	6.50	4.80	13.00	2.10	0.00	4.90	0.00	4.50	6.30	2.80
GPT-4 (CVCP)	6.20	11.00	15.80	9.90	2.80	2.25	7.60	5.50	15.40	2.70	0.15	5.50	0.10	5.10	7.50	3.40
DeepSeek-Coder (SOP)	4.10	7.50	10.20	6.80	1.90	1.70	4.30	3.50	10.00	1.60	0.00	3.80	0.00	3.10	5.20	2.40
DeepSeek-Coder (CVCP)	5.00	9.20	12.00	8.10	2.50	2.10	5.50	4.10	11.20	2.00	0.10	4.40	0.10	3.80	6.00	2.90
Claude-3.5-Sonnet (SOP)	9.80	13.00	16.00	11.20	1.50	2.90	7.70	6.60	19.60	1.90	0.95	5.70	0.05	3.83	8.30	3.52
Claude-3.5-Sonnet (CVCP)	11.60	14.50	17.83	13.30	2.32	3.30	9.50	8.30	21.80	2.30	1.15	2.30	0.34	4.70	8.80	4.10

To better illustrate the improvements enabled by the CVCP framework, we present a visual summary of key experimental results in Figure 3. The figure includes Elo rating comparisons across model types, pass rates stratified by problem difficulty, and a radar chart showing category-wise accuracy gains. These visualizations complement the tabular data by providing a more intuitive understanding of performance trends, especially the robustness gains on medium and hard problems, as well as across diverse algorithmic domains. Transitioning to Multi-LLM Agents, the SOP configuration already delivers noticeable gains over the single-agent baselines, while CVCP further amplifies performance across almost all categories. For example, GPT-4 (CVCP) surpasses GPT-4 (SOP) in Greedy (Gr.) (6.20 vs. 5.15), Implementation (Im.) (15.80 vs. 13.25), and Sorting (So.) (15.40 vs. 13.00), showing that coordinated multi-agent reasoning boosts solution accuracy. Similar trends hold for DeepSeek-Coder and Claude-3.5-Sonnet, with Claude-3.5-Sonnet (CVCP) achieving the strongest overall gains—most notably in Sorting (So.) (21.80) and Combinatorial Algorithms (CA.) (9.50). These results suggest that the multi-agent CVCP design, through collaborative verification and symmetry-aware adversarial testing, effectively enhances robustness and adaptability across a wide range of algorithmic domains, particularly for harder and more reasoning-intensive problem types.

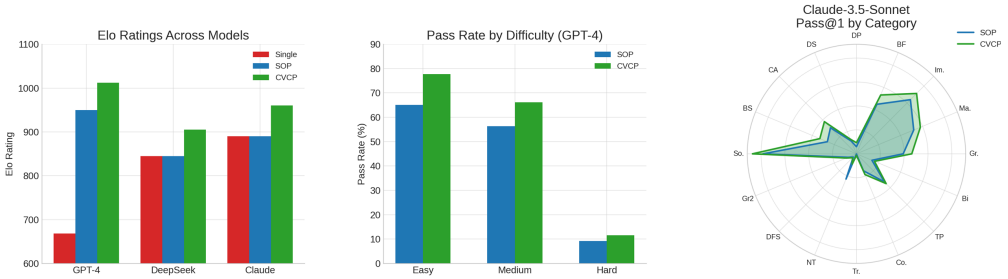

Figure 3. Comparison of model performance across CVCP, SOP, and single-agent baselines. (**Left**) Elo rating of different model backbones under three configurations (Single, SOP, CVCP), showing consistent improvement under CVCP. (**Middle**) Pass rate across problem difficulty levels (Easy, Medium, Hard) for GPT-4, where CVCP demonstrates notable gains on harder problems. (**Right**) Radar plot of pass@1 rate across 16 algorithm categories for Claude-3.5-Sonnet, illustrating the broader performance improvements brought by CVCP.

4.5. Human Evaluation

In order to provide a more comprehensive assessment beyond outcome-based metrics such as Elo rating, pass rate, and pass@k, we conducted a human evaluation to analyze the code quality, readability, and algorithmic optimality of generated programs. This evaluation was intended to capture qualitative aspects of the outputs that are often missed by automated metrics.

We randomly selected 50 competitive programming problems from the CodeElo benchmark and collected corresponding code solutions from nine different system variants. These variants consist of three backbone LLMs (GPT-4, DeepSeek-Coder, and Claude-3.5)

under three settings: the base version, an SOP-style multi-agent version, and our full CVCP-enhanced version. This produced a total of 450 code samples.

To evaluate these solutions, we recruited five experienced Python developers (Python Software Foundation, Version 3.8, Wilmington, DE, USA). Each evaluator independently assessed a subset of the samples while being blinded to the source model. Each solution was rated on a 1–5 scale across three axes: (1) general code quality, (2) readability and style, and (3) algorithmic efficiency and optimality. All samples were reviewed by at least three annotators, with cross-validation introduced for cases where discrepancies in rating exceeded 1 point.

From Table 6, we can see that the CVCP-enhanced systems outperform both the base and SOP-style systems across all three dimensions. Notably, the improvements are consistent across different backbone models, indicating that CVCP's benefit is not merely a function of the underlying LLM quality but stems from its principled design in coordinating agents with constraint-aware, adversarial, and review-driven mechanisms.

Table 6. Human evaluation scores (1–5 scale) on 50 randomly sampled problems from CodeElo. Each value is averaged across three criteria: code quality, readability, and algorithmic optimality.

Model	Setup	Code Quality	Readability	Algo Optimality
GPT-4	Base	3.94	3.81	3.76
GPT-4	+SOP	4.05	3.87	3.85
GPT-4	+CVCP	4.35	4.18	4.27
DeepSeek-Coder	Base	3.46	3.28	3.14
DeepSeek-Coder	+SOP	3.62	3.45	3.29
DeepSeek-Coder	+CVCP	3.95	3.77	3.66
Claude-3.5	Base	3.68	3.42	3.35
Claude-3.5	+SOP	3.82	3.56	3.49
Claude-3.5	+CVCP	4.11	3.95	3.88

5. Discussion

In this section, we address potential threats to the validity of our findings by discussing the internal and external validity concerns that could affect the interpretation and generalization of the experimental results.

5.1. Internal Validity

One of the key threats to internal validity lies in the model variations between the different models used in the experiments. While we attempted to standardize the inference parameters (such as temperature and max tokens), the models under evaluation, such as GPT-4 and Claude-3.5-Sonnet, have distinct architectures, pretraining data, and fine-tuning processes that may contribute to performance disparities. These inherent differences between models might influence their baseline performances, making it challenging to isolate the impact of the Cross-Verification Collaboration Protocol (CVCP) from the models' intrinsic strengths.

Additionally, the evaluation metric we used, pass@1, may not fully capture the models' overall ability to generate high-quality solutions. While pass@1 is a commonly used metric in competitive programming, it only measures whether a model successfully solves a problem on its first attempt, without considering additional submission attempts or the quality of the solution itself. This simplification could overlook other important factors, such as the model's ability to handle edge cases or optimize solution quality under different constraints, which are also crucial in competitive programming scenarios.

Finally, another internal validity concern arises from the distribution of test cases in the CodeELO dataset. The nature of the problems, their difficulty levels, and the types of algorithmic challenges in the dataset may inadvertently favor certain models based on their pretraining or structural characteristics. If the dataset is not representative of the full spectrum of competitive programming tasks, this could skew the performance results. However, given the wide range of problems and categories covered by Codeforces contests, we believe this threat is somewhat mitigated, though it remains a potential source of bias.

5.2. External Validity

The external validity of our findings is contingent upon how well the results can be generalized to other types of competitive programming challenges and real-world tasks. Since our experiments were conducted on problems from Codeforces contests, they may not fully reflect the challenges posed by other competitive programming platforms, such as LeetCode, AtCoder, or specialized contests like ICPC. Each platform has a different problem distribution and difficulty range, which could lead to varying performance results across different platforms. While we believe the lessons learned from Codeforces are broadly applicable, further research is needed to evaluate CVCP in other competitive settings to understand its generalization capabilities.

Moreover, the CVCP framework was specifically designed to optimize performance on algorithmic tasks found in competitive programming. While this represents a highly structured and challenging environment, real-world programming tasks often involve additional complexities, such as meeting business requirements, integrating with other systems, or adhering to non-functional constraints like performance and scalability. Therefore, while our results are promising, they may not directly translate to real-world software engineering problems, which require models to handle not just theoretical tasks but also practical and domain-specific constraints. As a result, it is unclear whether the improvements observed with CVCP in competitive programming would also apply to more applied, industry-oriented tasks.

Another significant threat to external validity arises from the scalability of multi-agent systems like CVCP. While the framework showed clear advantages in our experiments, the challenges of scaling it to larger, more complex tasks remain. CVCP relies on a set of specialized agents that must coordinate effectively to achieve high performance. In real-world applications, the increase in task complexity, team size, and collaboration requirements could introduce new challenges that may reduce the framework's effectiveness. Additionally, future research should explore whether CVCP remains effective as models grow larger and handle more diverse problem domains. Thus, while CVCP shows great potential in competitive programming, its scalability and adaptability to more varied and complex environments remain an open question.

5.3. Scaling Analysis

To explore how performance scales with model capacity, we conduct a comparative analysis using CodeLlama with three variants: CodeLlama-7B-Instruct and CodeLlama-13B-Instruct, alongside the 34B version.

Model Size vs. Elo Performance: We instantiate the CVCP agent framework using each model variant while keeping the protocol and decoding parameters fixed. Table 7 shows that Elo ratings scale positively with model size, with 7 B models achieving limited success (with Elo 600), and 34 B models surpassing 900 on average. The performance gap is most prominent on Div. 1 + 2 and Hard problems, highlighting larger models' advantages in complex reasoning and planning.

Table 7. Elo ratings of CodeLlama models across sizes and collaboration protocols.

Model Variant	Size	SOP Elo	CVCP Elo
CodeLlama-Instruct	7 B	540	610
CodeLlama-Instruct	13 B	670	730
CodeLlama-Instruct	34 B	845	905

5.4. Failure Case Analysis

Although CVCP demonstrates strong improvements over SOP baselines, certain failure modes persist. One common failure arises when the MVPR module over-generalizes symmetry patterns, leading to incorrect test transformations that misguide downstream agents. Another case occurs when the AVR module fails to flag subtly incorrect logic due to weak adversarial signals, especially in greedy or heuristic-based problems.

For example, in a Codeforces Div. 2 problem requiring greedy interval merging, GPT-4 (CVCP) produced the following incorrect implementation (Listing 1). Although the code passes most test cases, it fails edge cases where overlapping intervals require transitive merging—an aspect missed due to incomplete test transformations and a lack of recursive validation logic.

Listing 1. Failure case from a greedy interval merging task.

```
def merge_intervals(intervals):
    intervals.sort()
    result = []
    for i in range(len(intervals)):
        if not result or result[-1][1] < intervals[i][0]:
            result.append(intervals[i])
        else:
            result[-1][1] = intervals[i][1]  # Fails to take max(...)
    return result
```

This highlights a limitation of the current collaboration strategy: while symmetry-aware testing and voting modules help catch major flaws, subtle logical bugs may still slip through when adversarial inputs do not fully cover the edge case space. Addressing this limitation is part of our ongoing research on adaptive test generation and recursive validation.

6. Conclusions

In this paper, we proposed the Cross-Verification Collaboration Protocol (CVCP), a novel symmetry-aware multi-agent coordination framework that enhances code generation performance in competitive programming. By decomposing the generation pipeline into modular roles—such as Multi-View Problem Reading (MVPR), Assumption Tagging (ATI), Round-Trip Review (RTRP), and Cross-Test Adversarial Pairing (CTAP)—CVCP ensures high specification fidelity, robustness under symmetry perturbation, and collaborative decision-making.

Our evaluation on the CodeELO benchmark, which includes 387 Codeforces problems from May to November 2024, demonstrates consistent improvements over traditional SOP (Single One-Pass) pipelines. Across three backbone LLMs (GPT-4, DeepSeek-Coder, Claude-3.5-Sonnet), CVCP yields significant gains in Elo Rating, Pass Rate, and Pass@k. In particular, DeepSeek-Coder with CVCP improves its Pass@1 by 30% (from 22.3 to 29.0), showing CVCP's capacity to elevate even mid-tier models through architectural enhancements.

Limitations. While promising, our work has limitations. The framework's performance is still bounded by the reasoning ability and cost constraints of underlying LLMs. Moreover, our benchmark is currently limited to CodeELO, which—though diverse—still focuses

on algorithmic contest tasks. Real-world programming involves dynamic requirement shifts, integration with legacy systems, and adherence to non-functional constraints (e.g., scalability, latency), which CVCP does not yet model. Additionally, the multi-agent design introduces additional inference rounds, increasing latency and compute cost.

Future Work and Real-World Applications. To bridge toward industrial relevance, we envision several concrete extension paths. First, we plan to validate CVCP on more open-ended software tasks from platforms like GitHub (GitHub, Inc., Version 3.10.0, San Francisco, CA, USA), a widely-used platform for version control and collaborative software development. These tasks reflect realistic engineering workflows involving ambiguous requirements, iterative refinement, and long-term maintenance. Second, CVCP's modularity naturally lends itself to integration with IDEs and CI/CD pipelines—for instance, deploying ATI to flag unverifiable assumptions during code review, or using CTAP for adversarial test case injection in pre-merge testing.

Furthermore, the Round-Trip Review mechanism is particularly suited to contexts where specification drift is common—such as large team projects or evolving product requirements—making CVCP applicable not only to competitive programming, but also to enterprise-scale software development and automated QA workflows. We believe this alignment opens the door for future work that rigorously quantifies CVCP's utility beyond synthetic benchmarks and into real software engineering environments.

Author Contributions: Methodology, A.S.; Software, A.A.; Validation, A.S.; Visualization, A.A.; Writing—original draft, A.S.; Writing—review and editing, A.A. All authors have read and agreed to the published version of the manuscript.

Funding: The authors declare that this study received funding from Taylor's University.

Data Availability Statement: The data presented in this study are available on request from the corresponding author due to containing sensitive implementation details of the proposed framework that require additional verification before full release.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Dong, Y.; Jiang, X.; Jin, Z.; Li, G. Self-collaboration code generation via chatgpt. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 1–38. [CrossRef]
2. Hong, S.; Zheng, X.; Chen, J.; Cheng, Y.; Wang, J.; Zhang, C.; Wang, Z.; Yau, S.K.S.; Lin, Z.; Zhou, L.; et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv* **2023**, arXiv:2308.00352.
3. Ishibashi, Y.; Nishimura, Y. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization. *arXiv* **2024**, arXiv:2404.02183.
4. Perez, I.; Dedden, F.; Goodloe, A. *Copilot 3*; Technical Report; NASA: Washington, DC, USA, 2020.
5. Seo, M.; Baek, J.; Lee, S.; Hwang, S.J. Paper2Code: Automating Code Generation from Scientific Papers in Machine Learning. *arXiv* **2025**, arXiv:2504.17192. [CrossRef]
6. Wei, Y.; Wang, Z.; Liu, J.; Ding, Y.; Zhang, L. Magicoder: Empowering code generation with oss-instruct. *arXiv* **2023**, arXiv:2312.02120.
7. Wang, H.; Yang, W.; Yang, L.; Wu, A.; Xu, L.; Ren, J.; Wu, F.; Kuang, K. Estimating individualized causal effect with confounded instruments. In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, 14–18 August 2022; pp. 1857–1867.
8. Wang, H.; Kuang, K.; Lan, L.; Wang, Z.; Huang, W.; Wu, F.; Yang, W. Out-of-distribution generalization with causal feature separation. *IEEE Trans. Knowl. Data Eng.* **2023**, *36*, 1758–1772. [CrossRef]
9. Wang, H.; Li, H.; Zou, H.; Chi, H.; Lan, L.; Huang, W.; Yang, W. Effective and Efficient Time-Varying Counterfactual Prediction with State-Space Models. In Proceedings of the Thirteenth International Conference on Learning Representations, Singapore, 24–28 April 2025.

10. Geng, M.; Wang, S.; Dong, D.; Wang, H.; Cao, S.; Zhang, K.; Jin, Z. Interpretation-based code summarization. In Proceedings of the 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), Melbourne, Australia, 15–16 May 2023; pp. 113–124.
11. Geng, M.; Wang, S.; Dong, D.; Wang, H.; Li, G.; Jin, Z.; Mao, X.; Liao, X. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–13.
12. Akyar, I. Standard operating procedures (what are they good for?). *Latest Res. Qual. Control* **2012**, *12*, 367–391.
13. Quan, S.; Yang, J.; Yu, B.; Zheng, B.; Liu, D.; Yang, A.; Ren, X.; Gao, B.; Miao, Y.; Feng, Y.; et al. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. *arXiv* **2025**, arXiv:2501.01257.
14. Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F.L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. Gpt-4 technical report. *arXiv* **2023**, arXiv:2303.08774. [CrossRef]
15. Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; et al. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv* **2024**, arXiv:2401.14196.
16. Jiang, X.; Dong, Y.; Wang, L.; Fang, Z.; Shang, Q.; Li, G.; Jin, Z.; Jiao, W. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 1–30. [CrossRef]
17. Li, J.; Zhao, Y.; Li, Y.; Li, G.; Jin, Z. Acecoder: An effective prompting technique specialized in code generation. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 1–26. [CrossRef]
18. Zhang, S.; Chen, Z.; Shen, Y.; Ding, M.; Tenenbaum, J.B.; Gan, C. Planning with large language models for code generation. *arXiv* **2023**, arXiv:2303.05510. [CrossRef]
19. Jiang, J.; Wang, F.; Shen, J.; Kim, S.; Kim, S. A survey on large language models for code generation. *arXiv* **2024**, arXiv:2406.00515.
20. Liu, J.; Xia, C.S.; Wang, Y.; Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 21558–21572.
21. Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. Code llama: Open foundation models for code. *arXiv* **2024**, arXiv:2308.12950. [CrossRef]
22. Team, G.; Anil, R.; Borgeaud, S.; Alayrac, J.B.; Yu, J.; Soricut, R.; Schalkwyk, J.; Dai, A.M.; Hauth, A.; Millican, K.; et al. Gemini: A family of highly capable multimodal models. *arXiv* **2024**, arXiv:2312.11805.
23. Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv* **2024**, arXiv:2203.13474.
24. Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; et al. Deepseek-v3 technical report. *arXiv* **2024**, arXiv:2412.19437.
25. Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; Jiang, D. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv* **2023**, arXiv:2306.08568. [CrossRef]
26. Dong, G.; Yuan, H.; Lu, K.; Li, C.; Xue, M.; Liu, D.; Wang, W.; Yuan, Z.; Zhou, C.; Zhou, J. How abilities in large language models are affected by supervised fine-tuning data composition. *arXiv* **2023**, arXiv:2310.05492.
27. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.D.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374. [CrossRef]
28. Wang, S.; Wen, M.; Lin, B.; Wu, H.; Qin, Y.; Zou, D.; Mao, X.; Jin, H. Automated patch correctness assessment: How far are we? In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Melbourne, Australia, 21–25 September 2020; pp. 968–980.
29. Lin, B.; Wang, S.; Liu, Z.; Liu, Y.; Xia, X.; Mao, X. Cct5: A code-change-oriented pre-trained model. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, San Francisco, CA, USA, 3–9 December 2023; pp. 1509–1521.
30. Wang, S.; Lin, B.; Sun, Z.; Wen, M.; Liu, Y.; Lei, Y.; Mao, X. Two birds with one stone: Boosting code generation and code search via a generative adversarial network. *Proc. ACM Program. Lang.* **2023**, *7*, 486–515. [CrossRef]
31. Wang, S.; Geng, M.; Lin, B.; Sun, Z.; Wen, M.; Liu, Y.; Li, L.; Bissyandé, T.F.; Mao, X. Natural language to code: How far are we? In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, San Francisco, CA, USA, 3–9 December 2023; pp. 375–387.
32. Sun, Z.; Du, X.; Song, F.; Wang, S.; Li, L. When neural code completion models size up the situation: Attaining cheaper and faster completion through dynamic model inference. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–12.
33. Geng, M.; Dong, D.; Lu, P. Hierarchical Semantic Graph Construction and Pooling Approach for Cross-language Code Retrieval. In Proceedings of the 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), Chiang Mai, Thailand, 22–26 October 2023; pp. 393–402.

34. Geng, M.; Wang, S.; Dong, D.; Gu, S.; Peng, F.; Ruan, W.; Liao, X. Fine-grained code-comment semantic interaction analysis. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Pittsburgh, PA, USA, 16–17 May 2022; pp. 585–596.
35. Qian, C.; Liu, W.; Liu, H.; Chen, N.; Dang, Y.; Li, J.; Yang, C.; Chen, W.; Su, Y.; Cong, X.; et al. Chatdev: Communicative agents for software development. *arXiv* **2023**, arXiv:2307.07924.
36. Wu, Q.; Bansal, G.; Zhang, J.; Wu, Y.; Li, B.; Zhu, E.; Jiang, L.; Zhang, X.; Zhang, S.; Liu, J.; et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv* **2023**, arXiv:2308.08155.
37. Topsakal, O.; Akinci, T.C. Creating large language model applications utilizing langchain: A primer on developing llm apps fast. In Proceedings of the International Conference on Applied Engineering and Natural Sciences, Konya, Turkey, 10–12 July 2023; Volume 1, pp. 1050–1056.
38. Yang, H.; Yue, S.; He, Y. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv* **2023**, arXiv:2306.02224. [CrossRef]