Trustworthy Multi-Agent Code Generation with Execution-Based Validation

Nicola Capuano*,†, Alberto Marmo†, Pierluigi Ritrovato† and Raffaele Solimeno†

University of Salerno, Via Giovanni Paolo II 132, 84084 Fisciano (SA), Italy

Abstract

Recent advances in Large Language Models are driving the shift from code assistants to autonomous software development agents. However, at project level, autonomy alone does not ensure reliability: generated systems may appear plausible while still failing at startup, violating functional requirements, or exposing integration errors that emerge only at runtime. This paper presents an execution-validated multi-agent framework that combines role-specialized agents with execution-based validation. The framework supports requirements extraction, planning, iterative code generation, and automated validation through runtime checks and end-to-end testing, whose outcomes are fed back into the pipeline to drive targeted self-correction. Experiments on the web-project subset of ProjectEval show that the proposed architecture substantially improves over direct generation with the same underlying model. A baseline multi-agent configuration raises pass@1 from 7.79% to 16.88%, while an extended version with execution-based validation further increases pass@1 to 31.17% and significantly reduces structurally non-executable cases. Overall, the results indicate that trustworthy agentic software development benefits from combining multi-agent coordination, explicit intermediate validation, and execution-grounded feedback.

Keywords

Multi-Agent Systems, Trustworthy AI, Project-Level Code Generation, Execution-Based Validation

1. Introduction

Large Language Models (LLMs) are rapidly transforming software development, enabling a shift from code completion tools to increasingly autonomous systems capable of supporting multiple phases of the software engineering process. In this perspective, software generation is no longer seen as a one-shot prediction task, but as a structured workflow involving requirements interpretation, planning, implementation, validation, and revision. This evolution is closely aligned with the broader paradigm of *Agentic AI*, where intelligence emerges not only from language generation, but also from the ability to coordinate actions, interact with tools, and adapt behavior on the basis of observed outcomes [1].

However, increasing autonomy does not automatically imply increasing reliability. This limitation becomes particularly evident in project-level code generation, where the target is not an isolated function but a complete software project composed of multiple interdependent files, components, and interfaces. In such settings, generated code may appear structurally plausible while still failing at runtime, violating requirements, or exhibiting integration errors that only emerge through execution [2]. As a consequence, the main challenge is no longer just generating code, but engineering autonomous development systems whose behavior can be considered trustworthy and verifiable [3].

In this paper, we present a multi-agent framework for project-level code generation that combines role specialization with execution-based validation. The proposed architecture decomposes the development process into distinct phases for requirements extraction, strategic planning, iterative task-based code generation, and post-generation verification. A central aspect of the framework is the integration of an automated validation loop based on project execution and *Selenium*-driven [4] end-to-end testing,

WOA 2026: the 27th Workshop From Objects to Agents, June 15-17, 2026, Salerno, Italy

*Corresponding author.

†These authors contributed equally.

ncapuano@unisa.it (N. Capuano); a.marmo12@studenti.unisa.it (A. Marmo); pritrovato@unisa.it (P. Ritrovato); r.solimeno2@studenti.unisa.it (R. Solimeno)

https://docenti.unisa.it/nicola.capuano (N. Capuano); https://docenti.unisa.it/pierluigi.ritrovato (P. Ritrovato)

0000-0003-0862-3643 (N. Capuano); 0000-0003-4534-6972 (P. Ritrovato)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-10-30

CEUR-WS.org/Vol-4307/paper25.pdf

whose outcomes are used to generate targeted correction tasks and support self-corrective behavior. In this way, validation is not treated as a final external assessment step, but as an internal feedback mechanism that improves the functional reliability of the generated software.

The framework is evaluated on the web-project subset of *ProjectEval* [5] using *Level-2* checklist specifications. The experimental setting is particularly suitable for our goals because it targets complete software projects rather than isolated functions, and it assesses generated applications through execution-based tests. Results show that the proposed approach improves functional correctness over direct generation and significantly reduces the number of non-executable projects, suggesting that the combination of multi-agent coordination and execution-grounded feedback can provide a practical path toward more trustworthy forms of autonomous software development.

The remainder of the paper is organized as follows. Section 2 discusses the related work. Section 3 presents the proposed framework. Section 4 describes the experimental evaluation and discusses the obtained results. Section 5 concludes the paper and outlines future work.

2. Related Work

LLMs have shown strong capabilities in automated code generation, initially in settings where the target is an isolated function or code snippet derived from a natural language description. Early benchmarks such as *HumanEval* established execution-based evaluation as a standard approach for measuring functional correctness, while later datasets such as *APPS*, *CoderEval*, and *JavaBench* extended the scope toward more realistic programming tasks and more diverse software settings [6, 7, 8, 9]. Despite these advances, most of this line of work still evaluates models on self-contained units of code, leaving largely underexplored the difficulties that arise when generation must scale to complete software projects composed of multiple interacting files, modules, and interfaces.

This limitation has encouraged a shift from static code generation toward agentic approaches to software development. In this perspective, software generation is no longer treated as a one-shot prediction task, but as a structured process involving planning, decomposition, implementation, debugging, and revision. Recent literature on *Agentic AI* and LLM-based agents for software engineering emphasize that effective autonomous development requires more than raw generative capability: it also depends on decision-making, tool use, memory, and adaptation based on environmental feedback [3, 10].

Within this trend, *Multi-Agent Systems* are especially relevant because they enable explicit role specialization and clearer coordination across different phases of the software lifecycle [11]. *MetaGPT* is an influential example of this paradigm, showing how assigning software-engineering roles such as product manager, architect, and developer to different agents can improve coherence and task organization through structured intermediate artifacts [12]. Related work such as *CodePori* further confirms that distributing planning, implementation, and review responsibilities can improve scalability and robustness in complex software generation tasks [13].

A second major research direction concerns the validation of LLM-generated code. A recurring finding in the literature is that syntactic plausibility is an unreliable proxy for correctness, especially when generated systems must operate in realistic execution environments. More recent approaches therefore integrate validation directly into the generation loop. For example, *OpenCodeInterpreter* couples generation with execution and refinement, showing that runtime signals can substantially improve solution quality by exposing errors that remain invisible at the textual level [2]. Similarly, *AutoSafeCoder* combines multi-agent generation with static analysis and fuzz testing to improve the security and robustness of generated code [14].

These issues become even more critical at project level, where correctness depends not only on individual functions but also on module interfaces, dependency management, application startup, and end-to-end integration across the full system. For this reason, a growing body of work has moved toward project-level code generation and evaluation. Studies on project-level generation stress the importance of architectural planning, shared structured representations, and cross-file consistency management when code must be generated across multiple interdependent components [15, 16].

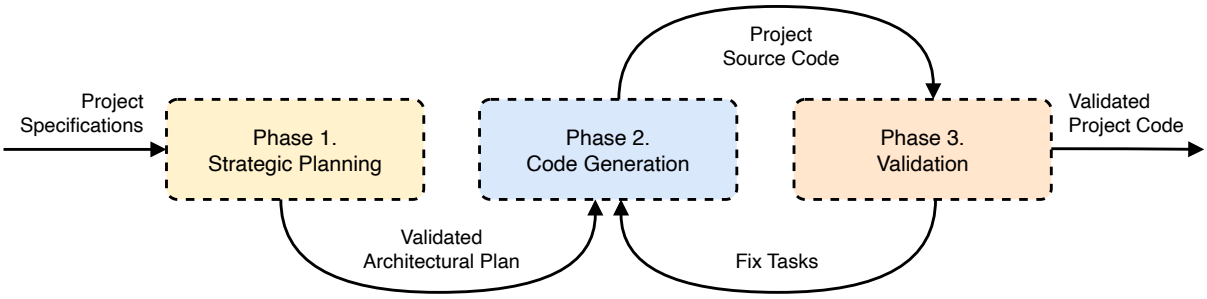

Figure 1: High-level overview of the proposed framework, organized into three macro-phases: Strategic Planning, Code Generation, and Validation. The validation phase can generate fix tasks for iterative refinement.

On the evaluation side, the *ProjectEval* benchmark [5] is particularly relevant because it explicitly targets complete software projects and assesses them through execution-based protocols, including *Selenium*-driven testing for web applications [5]. The benchmark provides strong evidence that good performance on function-level tasks does not necessarily transfer to realistic software engineering settings, where failures often arise from startup errors, interface inconsistencies, and integration bugs rather than from isolated logic mistakes.

Our work is positioned at the intersection of these directions. Unlike role-based multi-agent frameworks such as *MetaGPT* and *CodePori*, which mainly emphasize organizational workflow modeling and task decomposition, our framework focuses on project-level executability and functional correctness through explicit plan validation and execution-grounded repair. Unlike *OpenCodeInterpreter*, which applies execution and refinement mainly to individual programming tasks, our approach targets complete multi-file web projects and transforms runtime failures into structured fix tasks. Finally, unlike *AutoSafeCoder*, which focuses on security through static analysis and fuzz testing, our framework addresses end-to-end behavioral failures in generated applications. This combination of multi-agent coordination, intermediate validation, and execution-grounded feedback is the core contribution of our framework.

3. The Proposed Framework

The proposed framework for autonomous project-level code generation models software development as a multi-phase process in which specialized agents handle distinct tasks and exchange structured intermediate artifacts. The framework is developed in two configurations. The ***baseline architecture*** includes requirements extraction, planning, iterative code generation, and a lightweight execution check. The ***extended architecture*** adds an execution-based testing loop with *Selenium*-driven end-to-end testing and targeted self-correction, making it possible to distinguish the contribution of the multi-agent design from that of execution-grounded validation.

3.1. Framework Overview

At a high level, the framework is organized into three macro-phases as shown in Figure 1. Together, these phases define a workflow in which the input specification is progressively transformed into an executable software project and then validated through runtime checks and, in the extended configuration, automated end-to-end testing.

The ***Strategic Planning*** macro-phase is responsible for transforming the natural-language project specification into a structured representation that can guide downstream generation. In this phase, the framework extracts the functional constraints stated in the input, normalizes them into an explicit requirements contract, and produces a high-level architectural plan for the project. This phase is detailed in Section 3.2.

The ***Code Generation*** macro-phase is responsible for converting the validated plan into source code. Rather than generating the entire project in a single pass, the framework decomposes the plan

into a list of executable tasks, each associated with a specific file or module. Code generation then proceeds iteratively, so that project structure can emerge progressively while maintaining alignment with the validated architecture. This design supports a controlled development process and can integrate correction tasks (coming from the validation phase) back into the same generation loop. This phase is detailed in Section 3.3.

The ***Validation*** macro-phase is where the generated project is inspected, and validated. In the baseline architecture, this phase is limited to a lightweight execution check aimed at verifying whether the generated application can start successfully. In the extended architecture, the same phase is enriched with an automated testing loop that executes the generated project, produces end-to-end tests, collects structured failure reports, and feeds correction tasks back into the generation process. This phase is detailed in Section 3.4.

A central design choice of the framework concerns inter-agent coordination. Communication among agents is mediated by structured *JSON contracts* rather than free-form textual exchanges. Each agent receives a typed representation of the relevant intermediate artifact and produces an output that conforms to an explicit schema. This choice improves robustness by reducing interpretive ambiguity, facilitates debugging and monitoring, and makes the overall workflow more controllable.

The framework operates with temperature set to zero across all agents, prioritizing reproducibility and structural stability. This is especially important for planning agents, which must generate schema-conforming structured outputs, and for code-generation agents, whose behavior must remain consistent across iterative correction cycles. Prompt design also plays a key role in enforcing task boundaries and expected output formats, since each agent is instructed to operate only within its assigned responsibility and to produce outputs that can be directly consumed by downstream nodes.

From an implementation perspective, the framework is naturally mapped onto a graph-based orchestration model and is implemented using *LangGraph* [17]. In this setting, each agent corresponds to a graph node with a well-defined responsibility, while edges encode the transitions between phases. This design makes the workflow explicit and modular, supports persistent shared state across the generation process, and facilitates the integration of iterative correction loops.

3.2. Phase 1: Strategic Planning

This phase is responsible for transforming the input project specification into a validated architectural plan. As shown in Figure 2, it is organized as a sequence of specialized agents that progressively extract requirements, formulate a development strategy, and verify its consistency before any source code is produced. This design reflects a key architectural principle of the framework: conceptual errors should be intercepted as early as possible, when they can still be corrected at the planning level rather than after they have propagated into the generated project.

The phase starts with the ***Requirement Extractor***, which processes the natural-language project specification and converts it into an explicit *requirements contract*. Its role is intentionally conservative: rather than proposing design choices, the agent extracts and normalizes what is explicitly required by the input, including mandatory functionalities, optional features, relevant constraints, and application-level assumptions. Inputs to this agent are expected to conform to the *Level 2* format of the *ProjectEval* benchmark (see section 4.1). The resulting contract acts as the main reference artifact for the rest of the pipeline, ensuring that subsequent planning decisions remain grounded in the original specification.

The extracted contract is then consumed by the ***Strategy Planner***, whose goal is to produce a high-level *architectural plan* for the target project. This plan identifies the main project components, their responsibilities, the expected organization of files, and the overall implementation strategy. The planner establishes the global blueprint of the project. To reduce the risk of carrying planning errors into downstream generation, the architectural plan is subjected to two validation steps.

The first plan validation step is performed by the ***Compliance Checker***, which evaluates whether the proposed plan is consistent with the requirements contract. Its function is not to improve the plan, but to detect omissions, contradictions, or deviations from the requested functionality. If a lack of compliance is identified, the plan is passed to the ***Plan Refiner***, which revises the architectural plan

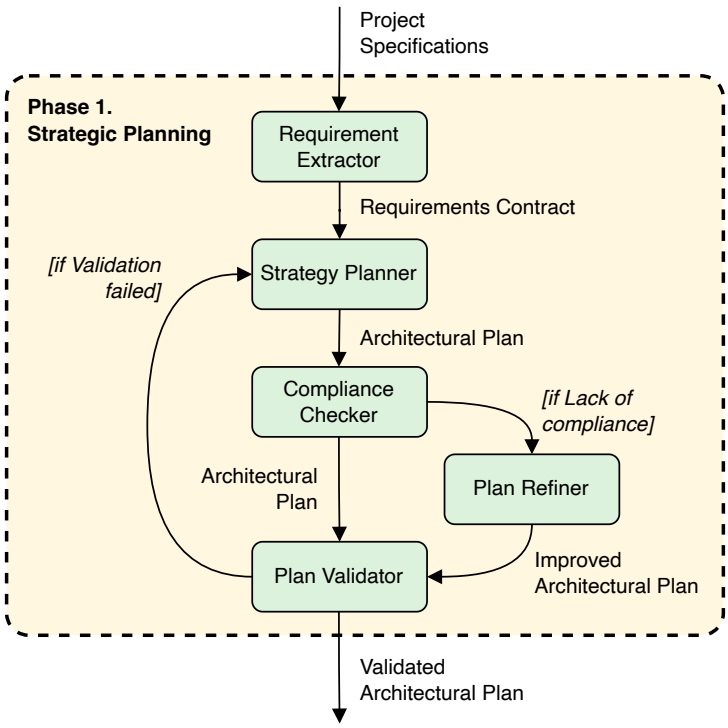

Figure 2: Workflow of the Strategic Planning phase, from the project specification to the production of a validated architectural plan.

while preserving the valid parts of the original proposal.

The second plan validation step is carried out by the ***Plan Validator***, which performs a final validation before the framework proceeds to code generation. While the *Compliance Checker* focuses primarily on alignment with the requirements contract, the *Plan Validator* assesses the overall soundness of the plan as a project-level blueprint. This additional verification step helps ensure that the plan is not only compliant with the specification, but also sufficiently coherent and complete to support downstream task decomposition and iterative implementation. If validation fails, the plan is sent back to the *Strategy Planner* for revision; otherwise, this phase produces a *validated architectural plan*, which serves as the entry point to the next macro-phase.

3.3. Phase 2: Code Generation

This phase is responsible for translating the validated architectural plan into the source code of the target project. As shown in Figure 3, it is organized as an iterative workflow in which the project structure defined during strategic planning is decomposed into executable tasks and progressively materialized into code artifacts. Rather than generating the entire project in a single pass, the framework follows a task-by-task strategy that supports finer control over implementation, preserves intermediate state, and enables the integration of correction tasks when needed.

The phase starts with the ***Task Generator***, which receives the validated architectural plan and converts it into a set of structured *task specifications*. Each task corresponds to a project file to be implemented, and includes the information required to guide downstream generation, such as the target artifact, its functional role, and its dependencies on the rest of the project. This decomposition step is essential because it transforms the architectural blueprint into an operational representation that can be executed incrementally by the generation agents.

The task specifications are then processed by the ***Task Iterator***, which controls the iterative code development process. Its role is to select the current task and route it to the appropriate *Code Generator*. In addition, it can also receive *fix-tasks* produced by the validation phase, allowing the same generation loop to support both initial implementation and targeted correction.

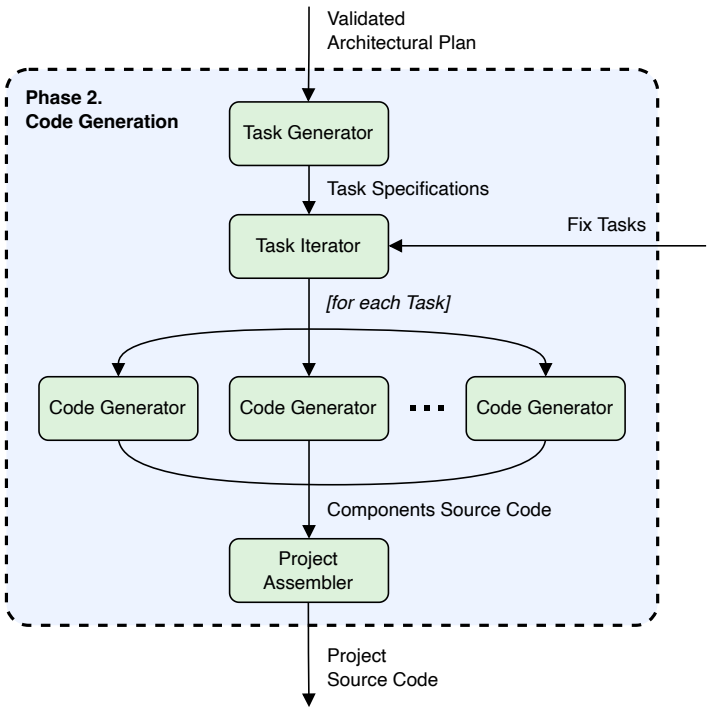

Figure 3: Workflow of the Code Generation phase, from the validated architectural plan to the generation of executable project source code.

The actual source code is produced by one or more ***Code Generators***, each responsible for implementing the current task specification and generating the corresponding source code. This organization reflects the iterative nature of the phase, in which multiple project components are generated in a way that preserves consistency with the validated architecture and with the code already produced.

The ***Project Assembler*** collects the generated code artifacts and assembles them into a coherent project structure. It acts as the consolidation layer of the phase, ensuring that each generated file is placed in the appropriate position within the project structure and that the output of the code generators is integrated correctly with the rest of the project. Once all tasks have been executed, the phase outputs the assembled project source code, which becomes the input to the subsequent validation phase.

3.4. Phase 3: Validation

This phase is responsible for checking whether the generated project can be executed correctly and, when necessary, for producing targeted correction tasks. In the baseline architecture, validation is limited to project execution and runtime-error detection. In the extended architecture, the same phase is augmented with automatic test generation and execution, enabling a richer feedback loop based on end-to-end behavioral validation.

3.4.1. Validation in the Baseline Architecture

As shown in Figure 4, in the baseline architecture, the validation process is relatively straightforward. The generated project is first processed by the ***Project Runner***, whose role is to execute the application and determine whether it can start successfully. This step acts as a lightweight execution check, aimed at detecting startup failures, configuration problems, missing dependencies, or other blocking runtime errors that prevent the system from reaching an executable state.

If runtime errors are encountered, the failure information is passed to the ***Fix-Task Generator***, which translates execution diagnostics into structured *fix tasks*. These tasks specify which parts of the project require correction and are then fed back into the code-generation phase for targeted repair. In this way,

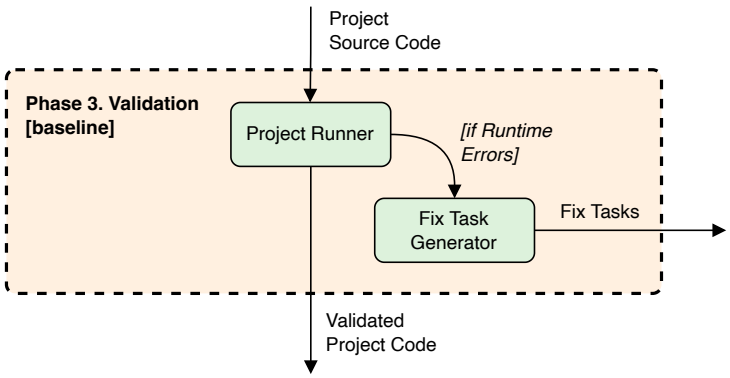

Figure 4: Workflow of the Validation phase in the baseline architecture, which is based on project execution and runtime-error-driven fix-task generation.

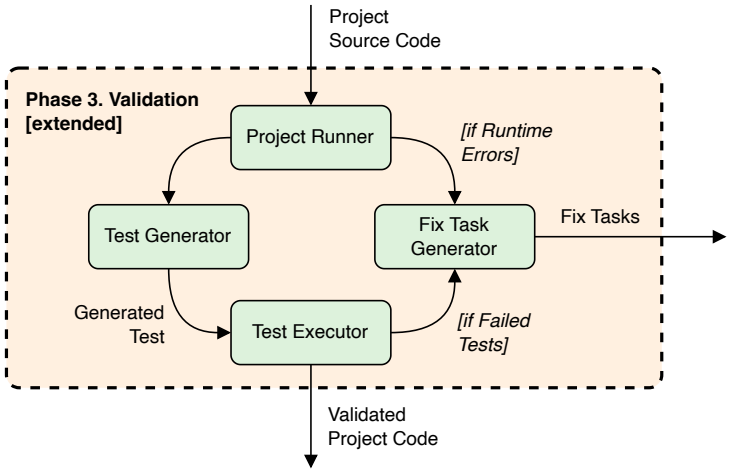

Figure 5: Workflow of the Validation phase in the extended architecture, which combines project execution, test generation, test execution, and fix-task generation.

even in the baseline architecture, validation is not merely a terminal check, but already provides a first form of corrective feedback grounded in observed execution outcomes.

3.4.2. Validation in the Extended Architecture

The extended architecture strengthens the Validation phase with an explicit execution-based testing loop. As shown in Figure 5, the phase still starts from the ***Project Runner***, which performs the initial runtime check on the generated project source code. However, in this configuration project execution is no longer the only validation signal. Once the application is found to be executable, the workflow proceeds to a dedicated testing stage.

The ***Test Generator*** is responsible for producing executable tests for the generated project. Its role is to translate the available project context into a set of operational checks that formalize expected behavior. In the context of web applications, this includes end-to-end interactions that exercise the system through its external interface. More specifically, the generated tests can inspect routing behavior, form handling, input validation, page rendering, and other integration points that are only observable when the application is executed as a whole. This design allows the framework to inspect whether the generated project satisfies the expected behavior under realistic execution conditions.

The generated tests are then passed to the ***Test Executor***, which runs them against the project and collects the resulting outcomes. Test execution takes place in an isolated environment, ensuring that each validation cycle is reproducible and not affected by residual state from previous iterations. To this end, the framework relies on *Selenium*, which enables browser automation through the *WebDriver*

protocol. The resulting feedback includes test outcomes, runtime diagnostics, and failure messages that can be traced back to the responsible project components.

If all tests succeed, the phase outputs validated project code. If instead the execution reveals failed tests, the resulting diagnostics are forwarded to the ***Fix-Task Generator***, which converts them into structured fix tasks and routes them back into the generation loop. In this way, the extended Validation phase implements a feedback cycle in which runtime and testing evidence drive targeted repair actions.

4. Experimental Evaluation

To assess the effectiveness of the proposed framework on project-level code generation, we conducted an experimental evaluation on the *ProjectEval* benchmark. The study is designed to quantify the contribution of the two main architectural choices underlying the framework, namely the multi-agent decomposition of the development process and the execution-based validation loop. The following subsections first introduce the benchmark and the adopted experimental setup, then define the evaluation metrics, and finally present and discuss the obtained results.

4.1. The ProjectEval Benchmark

ProjectEval [5] is a benchmark designed for the evaluation of programming agents on complete software project generation tasks. Unlike traditional function-level benchmarks, which assess the ability of a model to synthesize isolated code units, *ProjectEval* targets realistic multi-file software development scenarios and evaluates generated projects through execution-based protocols. The benchmark includes 20 real-world projects for a total of 284 test cases. Projects are divided into two categories: *web applications* (16 projects with 231 test cases), evaluated through Selenium-based user interaction, and *batch or console programs*, (4 projects with 53 test cases), evaluated through direct execution.

A distinctive feature of *ProjectEval* is the organization of inputs into three levels of increasing specificity. At ***Level 1***, the agent receives a high-level natural language prompt describing the overall project goal. This is the least constrained setting. At ***Level 2***, the input is a structured natural language checklist of functionalities and requirements, providing a more explicit specification of the expected behavior. At ***Level 3***, the agent receives a code skeleton, including comments and docstrings that guide implementation. These three levels make it possible to evaluate project-generation systems under different degrees of specification detail, from loosely defined requests to highly constrained implementation settings. Examples of the three input levels are shown in Figure 6.

The evaluation protocol is execution-based, meaning that the generated projects are not assessed by comparing their source code with a canonical implementation. Instead, each generated project is instantiated and validated through automated tests that check its actual behavior. To enable this process, after generation, the generation system receives its own project together with a *Parameter Description* (PD), that is, a structured specification of the information required by the benchmark to interact correctly with that particular implementation. This may include, for example, element identifiers, route paths, form-field names, or other project-dependent values needed to bind the generic test logic to the generated application.

Based on this description, the generation system must provide the corresponding *Parameter Values* (PV), which are then used by the *ProjectEval* evaluation engine to execute the test suite against the generated project. Test outcomes are finally classified as *passed* or *failed*.

4.2. Experimental Setup

The experimental evaluation was conducted on the *ProjectEval* benchmark, with a specific focus on its web-project subset, including 16 projects and 231 test cases. All experiments were performed using ***Qwen2.5-Coder-32B-Instruct*** [18] as the underlying language model. This model was selected because it is open-source, specifically optimized for code generation, and particularly well suited to *Python* and web-development tasks. Since this model was not included in the original *ProjectEval* study, its

Level 1: NL Prompt

Create a website for a Todo application, additional chart for analysis will be better.

Level 2: NL Checklist

```
"Page-Todo List":
 "Create New List": "Allows users to create a new Todo List",
 "View Tasks":"Displays all tasks under a selected list",
```

Level 3: Skeleton

```
class TodoListView(ListView):
    def get_queryset(self):
    """
    Gets the list of TodoLists, updating their completion status.
    """
```

Figure 6: The three input levels of the *ProjectEval* benchmark (adapted from [5]).

direct-generation baselines at input Levels 1, 2, and 3 were obtained under the same evaluation protocol, allowing a fair comparison between direct generation and the proposed architecture (see table 1).

The proposed framework receives *Level 2* input from *ProjectEval*, namely the Natural Language Checklist. This configuration was selected because it most naturally reflects a realistic software engineering setting, in which implementation starts from an explicit functional requirements specification rather than from an underspecified prompt or from a partial code skeleton.

At each correction iteration, the previous project version was archived before the newly generated fixes were applied. This choice made it possible to inspect the refinement trajectory step by step and to detect possible regressions introduced by the correction process. When failures are detected, the framework triggers a correction cycle regulated by a configurable maximum iteration budget, which limits the number of corrective attempts and ensures termination. For the experiments reported in this paper, the maximum iteration budget was set to 3.

It should be noted that the tests generated and executed during the repair phase are independent from the official *ProjectEval* evaluation suite. Internal tests are dynamically produced by the *Test Generator* using only the project specification and the generated project artifacts available to the framework. The *ProjectEval* test cases are never exposed during generation or repair and are executed exclusively during the final evaluation stage. Consequently, no evaluation-specific information is available during correction, preventing test leakage and ensuring a fair assessment of the proposed architecture.

4.3. Evaluation Metrics

ProjectEval natively distinguishes only between *passed* and *failed* test cases. While this binary outcome is sufficient to compute the overall benchmark score, it does not make explicit whether a failure is due to incorrect functional behavior or to the impossibility of executing the test at all. To make this distinction explicit and provide a more informative analysis, we introduce an additional category, namely *skipped*.

Accordingly, our evaluation considers the distribution of outcomes across *passed*, *failed*, and *skipped* test cases. Passed test cases indicate correct behavior for the corresponding benchmark checks. Failed test cases include all non-passing cases, both when the application runs but does not satisfy the expected functionality and when it cannot be executed at all. The *skipped* category identifies this second subset,

namely test cases that cannot be executed because the generated application does not start correctly. As such, skipped test cases are treated as a proxy for the structural executability of the generated systems.

Based on this distribution, the main aggregate metric adopted in this study is *pass@1*. In its classical formulation [6], the more general *pass@k* metric estimates the probability that at least one among k sampled candidate solutions for the same problem passes the complete test suite. This definition is appropriate in settings where multiple independent solutions are generated for the same input and correctness is evaluated at the problem level. When $k = 1$, the metric reduces to the probability that a single generated solution is correct:

$$\text{pass@}k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \qquad \text{pass@}1 = \frac{c}{n}$$

where n denotes the total number of sampled solutions and c the number of correct ones.

ProjectEval, however, adopts a revised formulation, similar to that proposed in [19], that is better suited to project-level evaluation. For each run, the model generates one complete project for every instance in the evaluation set, and performance is computed globally over the whole benchmark as the ratio between the total number of passed test cases and the total number of test cases. When $k > 1$, the entire evaluation process is repeated over k independent generations, and the final score is obtained by averaging the resulting benchmark-level pass rates. Formally, this can be written as:

$$\text{pass@}k = \frac{1}{k} \sum_{i=1}^{k} \frac{P_i}{T} \qquad \text{pass@}1 = \frac{P_1}{T}$$

where P_i denotes the number of test cases passed at the i-th independent generation, and T is the total number of test cases in the benchmark.

In the present work, we adopt the revised *ProjectEval* evaluation metric and set $k = 1$, both for methodological consistency and for direct comparability with the benchmark results. Although the framework could in principle generate multiple alternative projects, our goal is to assess whether it can produce a correct solution in its first delivered output, which is the most relevant setting for practical deployment in realistic software engineering scenarios.

4.4. Experimental Results

Table 1 summarizes the main experimental results. To provide a broader point of reference, the table also includes representative direct-generation results from the original *ProjectEval* paper, grouped into open-source general-purpose models (first group), code-specific models (second group), and commercial models (third group). These entries are reported for contextual comparison only, since they were obtained on the full *ProjectEval* benchmark with 284 test cases, whereas the experiments of this work were conducted on the 231 test cases of the web-project subset.

To provide a more direct comparison, the table also includes the results of *Qwen2.5-Coder-32B-Instruct* under direct generation at input Levels 1, 2, and 3 on the web-project subset (fourth group). Since this model was not included in the original benchmark, these results were obtained under the same evaluation protocol as the proposed architecture, making them the most appropriate baseline for assessing the contribution of our framework.

Under direct generation, *Qwen2.5-Coder-32B-Instruct* achieves a pass@1 of 6.06%, 7.79%, and 7.79% at input Levels 1, 2, and 3, respectively. When the same model is embedded in the baseline version of the proposed architecture, performance increases to 39 passed test cases out of 231, corresponding to a pass@1 of 16.88%. This already represents a substantial improvement over direct generation at Level 2. When the extended architecture is activated, performance further rises to 72 passed test cases and a pass@1 of 31.17%, doubling the result of the baseline architecture and strongly improving over all direct-generation configurations of the same model. The gain from 39 to 72 passed test cases provides a direct estimate of the contribution of the execution-based correction loop.

Table 1
Comparison of generation results on *ProjectEval* across open-source general-purpose models, code-specific models, commercial models, and the configurations evaluated in this work. Results marked with * are taken from the original *ProjectEval* paper [5].

System	Level	Total	Passed	Failed	Skipped	Pass@1
Llama-2-7B*	L2	284	0	284	–	0.00%
Llama-3.1-7B*	L2	284	2	282	–	0.70%
Phi-4-14B*	L2	284	4	280	–	1.41%
Gemma-7B*	L2	284	2	282	–	0.70%
Gemma-2-9B*	L2	284	3	281	–	1.06%
Mistral-7B-v0.3*	L2	284	4	280	–	1.41%
CodeGemma*	L3	284	6	278	–	2.11%
CodeLlama*	L3	284	4	280	–	1.41%
GPT-3.5-turbo*	L2	284	7	277	–	2.46%
GPT-4o*	L2	284	51	233	–	17.96%
Gemini-1.5-Pro*	L2	284	18	266	–	6.34%
Gemini-2.0-Flash*	L2	284	19	265	–	6.69%
Gemini-2.0-Pro*	L2	284	16	268	–	5.63%
Qwen2.5-Coder-32B-Instruct	L1	231	14	217	88	6.06%
Qwen2.5-Coder-32B-Instruct	L2	231	18	213	69	7.79%
Qwen2.5-Coder-32B-Instruct	L3	231	18	213	57	7.79%
Proposed Architecture (baseline)	L2	231	39	192	97	16.88%
Proposed Architecture (extended)	L2	231	**72**	**159**	**35**	**31.17%**

A closer inspection of the outcome distribution helps explain this improvement. The baseline architecture records 97 skipped test cases, whereas the extended architecture reduces them to 35. This reduction suggests that the execution-based validation loop is particularly effective in increasing the structural executability of the generated systems, allowing a much larger portion of the benchmark to be actually exercised. At the same time, the number of failed test cases decreases from 192 to 159, while the number of passed ones increases substantially.

Taken together, these results indicate that the proposed architecture contributes at two complementary levels: first, the multi-agent decomposition improves the quality of the initially generated projects; second, the execution-based validation and targeted correction cycle significantly increases both executability and functional correctness.

From a broader perspective, the table also shows that the extended architecture reaches a pass@1 higher than all the direct-generation models reported from the original benchmark, including the strongest commercial baselines. While this comparison should be interpreted with caution because of the different evaluation subsets, it still highlights the effectiveness of combining structured multi-agent generation with execution-grounded refinement.

4.5. Discussion

The experimental results suggest that the effectiveness of the proposed framework does not depend solely on the coding capabilities of the underlying language model, but also on the way the software generation process is structured. A first relevant observation is that the baseline architecture already yields a substantial improvement over direct generation with the same model. Since this configuration does not yet include the Selenium-based validation loop, the gain can be attributed to the architectural decomposition itself, in particular to the explicit separation between requirement extraction, strategic planning, intermediate validation, and iterative task-based generation. This indicates that, at project level, organizing generation as a coordinated workflow is already more effective than treating software development as a single prompting step.

A second key point concerns the role of execution-based validation. The additional improvement obtained by the extended architecture shows that testing is not only useful as a final assessment mechanism, but also as an internal source of corrective feedback. By grounding refinement in observed execution outcomes, the framework becomes able to detect and address errors that would remain invisible to static generation alone, especially those involving route configuration, inter-file consistency, template integration, and application-level behavior. In this sense, the validation loop does not simply measure correctness: it actively contributes to achieving it.

The interaction between failed and skipped test cases also provides an important interpretative insight. In the proposed evaluation setting, skipped test cases identify benchmark checks that cannot be executed because the generated application does not start correctly. Their reduction in the extended architecture therefore reflects a clear gain in structural executability and reflects a transition from non-executable projects to executable but still imperfect ones, which is in fact a necessary intermediate step toward full functional correctness.

From a qualitative perspective, the framework appears particularly effective in correcting startup and integration errors, while more subtle application-logic defects remain comparatively harder to resolve. This distinction is consistent with the nature of the available feedback. Runtime errors and Selenium failures usually provide precise diagnostic signals, often directly pointing to missing routes, unresolved imports, invalid HTML element bindings, or broken interaction flows. By contrast, logic errors may require a deeper semantic understanding of the intended functionality and are therefore less directly recoverable from execution traces alone. This suggests that the current architecture is especially strong in improving structural robustness and end-to-end coherence, while further advances may be needed to better support the correction of domain-specific logic flaws.

More broadly, these findings support a methodological conclusion. In project-level code generation, the main challenge is not only to generate plausible code, but to orchestrate a reliable development process in which planning, implementation, validation, and revision are explicitly coordinated. Under this perspective, the contribution of the proposed framework lies not only in improving benchmark performance, but also in showing that trustworthy autonomous software generation benefits from a process-oriented design, where execution is used as a grounding mechanism for refinement rather than as a purely final evaluation step.

At the same time, the study has some limitations. The experiments are restricted to the web-project subset of *ProjectEval*, and therefore do not yet establish the generality of the approach across other project categories. Moreover, the full framework is evaluated with a single LLM, so additional experiments would be needed to assess the robustness of the architecture across different model families. Finally, the adopted benchmark-level pass rate captures functional success at the test-case level, but does not directly measure other relevant software quality dimensions such as maintainability, readability, or security. These aspects remain important directions for future work.

5. Conclusions

This work investigated how project-level code generation can be made more reliable by moving from one-shot generation to a structured multi-agent development process. The proposed framework combines requirements-driven planning, iterative task-based implementation, and execution-based validation within a single coordinated workflow. In this perspective, the main contribution of the paper is not only architectural, but methodological: it shows that autonomous software generation becomes more dependable when correctness is pursued through explicit intermediate artifacts and feedback loops, rather than being delegated entirely to the initial generative step.

The experimental results support this view. The baseline architecture already improves substantially over direct generation with the same model, showing that role specialization and explicit planning provide a concrete benefit even before testing is introduced. The extended configuration then delivers a second, larger improvement by integrating runtime checks and Selenium-based end-to-end validation into the correction loop. This combination makes it possible not only to increase the number of passed

test cases, but also to turn previously non-executable projects into systems that can be meaningfully assessed and progressively repaired.

More broadly, the findings suggest that project-level code generation should be understood as a problem of reliable orchestration rather than mere code synthesis. At this scale, failures often arise from interactions among files, routes, templates, and runtime dependencies, and therefore cannot be resolved by textual plausibility alone. Execution-based validation is valuable precisely because it provides grounded signals about these cross-component failures and enables targeted refinement. In this sense, the proposed framework offers a practical step toward more reliable and execution-validated forms of autonomous software development, where validation is not external to generation but embedded within it.

Several directions remain open for future work. A natural extension is the integration of static-analysis modules to complement dynamic validation with structural quality checks. Another is the application of the framework to additional web frameworks and to broader project categories beyond the current subset. Finally, further work is needed to improve the correction of more subtle application-logic errors and to study adaptive policies for controlling the refinement budget. These directions would help extend the framework from a proof of effectiveness on a realistic benchmark to a more general foundation for robust autonomous software engineering.

Declaration on Generative AI

The authors acknowledge the use of ChatGPT (OpenAI) as a support tool during the preparation of this manuscript, particularly for language refinement and text revision. All scientific choices, experimental design, results, and final content remain the sole responsibility of the authors.

References

[1] H. Wang, J. Gong, H. Zhang, J. Xu, Z. Wang, Ai agentic programming: A survey of techniques, challenges, and opportunities, 2025. URL: https://arxiv.org/abs/2508.11126. arXiv:2508.11126.

[2] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, et al., Opencodeinterpreter: Integrating code generation with execution and refinement, in: L.-W. Ku, A. Martins, V. Srikumar (Eds.), Findings of the Association for Computational Linguistics: ACL 2024, Association for Computational Linguistics, Bangkok, Thailand, 2024, pp. 12834–12859. URL: https://aclanthology.org/2024.findings-acl.762/. doi:10.18653/v1/2024.findings-acl.762.

[3] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, et al., From llms to llm-based agents for software engineering: A survey of current, challenges and future, 2025. URL: https://arxiv.org/abs/2408.02479. arXiv:2408.02479.

[4] SeleniumHQ, Selenium: Automates browsers. that's it!, https://github.com/SeleniumHQ/selenium, 2026. GitHub repository, accessed 2026-04-18.

[5] K. Liu, Y. Pan, Y. Xiang, D. He, J. Li, et al., Projecteval: A benchmark for programming agents automated evaluation on project-level code generation, in: W. Che, J. Nabende, E. Shutova, M. T. Pilehvar (Eds.), Findings of the Association for Computational Linguistics: ACL 2025, Association for Computational Linguistics, Vienna, Austria, 2025, pp. 20205–20221. URL: https://aclanthology.org/2025.findings-acl.1036/. doi:10.18653/v1/2025.findings-acl.1036.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, et al., Evaluating large language models trained on code, 2021. URL: https://arxiv.org/abs/2107.03374. arXiv:2107.03374.

[7] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, et al., Measuring coding challenge competence with apps, 2021. URL: https://arxiv.org/abs/2105.09938. arXiv:2105.09938.

[8] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, et al., Codereval: A benchmark of pragmatic code generation with generative pre-trained models, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, Association for Computing Machinery, New York, NY, USA, 2024. URL: https://doi.org/10.1145/3597503.3623316. doi:10.1145/3597503.3623316.

[9] J. Cao, Z. Chen, J. Wu, S.-C. Cheung, C. Xu, Javabench: A benchmark of object-oriented code generation for evaluating large language models, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 870–882. URL: https://doi.org/10.1145/3691620.3695470. doi:10.1145/3691620.3695470.

[10] M. A. Shehab, M. Wardat, S. Omari, Y. Jararweh, Evaluating large language models for code generation: Assessing accuracy, quality, and performance, in: 2024 2nd International Conference on Foundation and Large Language Models (FLLM), 2024, pp. 407–416. doi:10.1109/FLLM63129.2024.10852439.

[11] M. Abou Ali, F. Dornaika, J. Charafeddine, Agentic ai: a comprehensive survey of architectures, applications, and future directions, Artificial Intelligence Review 59 (2025) 11. URL: https://doi.org/10.1007/s10462-025-11422-4. doi:10.1007/s10462-025-11422-4.

[12] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, et al., Metagpt: Meta programming for a multi-agent collaborative framework, 2024. URL: https://arxiv.org/abs/2308.00352. arXiv:2308.00352.

[13] Z. Rasheed, M. A. Sami, K.-K. Kemell, M. Waseem, M. Saari, et al., Codepori: Large-scale system for autonomous software development using multi-agent technology, 2024. URL: https://arxiv.org/abs/2402.01411. arXiv:2402.01411.

[14] A. Nunez, N. T. Islam, S. K. Jha, P. Najafirad, Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing, 2024. URL: https://arxiv.org/abs/2409.10737. arXiv:2409.10737.

[15] L. Schmid, T. Hey, M. Armbruster, S. Corallo, D. Fuchß, et al., Software architecture meets llms: A systematic literature review, 2025. URL: https://arxiv.org/abs/2505.16697. arXiv:2505.16697.

[16] D. Fedrushkov, D. Tereshchenko, S. Kovalchuk, A. Aliev, Improving project-level code generation using combined relevant context, in: M. H. Lees, W. Cai, S. A. Cheong, Y. Su, D. Abramson, J. J. Dongarra, P. M. A. Sloot (Eds.), Computational Science – ICCS 2025, Springer Nature Switzerland, Cham, 2025, pp. 438–445.

[17] LangChain, Inc., Langgraph: Low-level orchestration framework for building stateful agents, https://github.com/langchain-ai/langgraph, 2026. GitHub repository, accessed 2026-04-18.

[18] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, K. Dang, Y. Fan, Y. Zhang, A. Yang, R. Men, F. Huang, B. Zheng, Y. Miao, S. Quan, Y. Feng, X. Ren, X. Ren, J. Zhou, J. Lin, Qwen2.5-coder technical report, 2024. URL: https://arxiv.org/abs/2409.12186. arXiv:2409.12186.

[19] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, P. Liang, SPoC: search-based pseudocode to code, Curran Associates Inc., Red Hook, NY, USA, 2019.