Contract-Coding: Towards Repo-Level Generation via Structured Symbolic Paradigm

Yi Lin Lujin Zhao Yijie Shi[*]
State Key Laboratory of Networking and Switching Technology
Beijing University of Posts and Telecommunications
Beijing, 100876, China
{yilin, zlujin, yijieshi2000}@bupt.edu.cn

Abstract

The shift toward intent-driven software engineering (often termed "Vibe Coding") exposes a critical Context-Fidelity Trade-off: vague user intents overwhelm linear reasoning chains, leading to architectural collapse in complex repo-level generation. We propose Contract-Coding, a structured symbolic paradigm that bridges unstructured intent and executable code via Autonomous Symbolic Grounding. By projecting ambiguous intents into a formal Language Contract, our framework serves as a Single Source of Truth (SSOT) that enforces topological independence, effectively isolating inter-module implementation details, decreasing topological execution depth and unlocking Architectural Parallelism. Empirically, while state-of-the-art agents suffer from different hallucinations on the Greenfield-5 benchmark, Contract-Coding achieves 47% functional success while maintaining near-perfect structural integrity. Our work marks a critical step towards repository-scale autonomous engineering: transitioning from strict "specification-following" to robust, intent-driven architecture synthesis. Our code is available at https://github.com/imliinyi/Contract-Coding.

1 Introduction

The advent of Large Language Models (LLMs) has fundamentally transformed Automated Software Engineering (ASE) (Chen et al., 2021; Li et al., 2022), evolving from solitary code completion to collaborative Multi-Agent Systems (MAS) capable of handling complex workflows (Chen et al., 2023; Li et al., 2023a). While frameworks like MetaGPT (Hong et al., 2024) and ChatDev (Chen et al., 2024) demonstrate that specialized roles enhance problem-solving (Yang et al., 2024; Wang et al., 2025), they encounter a scalability wall.

A critical gap remains between academic research and real-world adoption. While *Specification-Driven* approaches (Spec-Coding) offer architectural rigor, they suffer from a "Formalization Bottleneck": assuming complete structural blueprints that users rarely possess (Zelikman et al., 2023; Wen et al., 2025). This contradicts the reality of *Intent-Driven* development ("Vibe Coding"), where users provide ambiguous functional goals. Conversely, synthesizing repositories directly from these intents hits a *Context-Fidelity Trade-off*. The dominant "linear chain-of-thought" paradigm (Wei et al., 2022) faces two systemic bottlenecks: (1) *Sequential Entropy Accumulation*, where early architectural ambiguities amplify as semantic noise (Noah et al., 2023); and (2) *The Context Bottleneck*, where accumulating implementation details exhausts finite windows, leading to *Architectural Collapse* (Liu et al., 2024a; Packer et al., 2024).

We challenge the assumption that repository synthesis must be sequential. We argue that the root cause of scalability failure is the entanglement of control complexity and implementation volume. To scale Vibe Coding, we must revisit the principle of *Information Hiding* (Parnas, 1972) via a structured-symbolic lens. Instead of parsing the noisy, full-context raw code of peers, agents should align with a high-fidelity "architectural thumbnail"—or *Language Contract*. This mechanism grounds vague intents into rigorous abstractions (Meyer, 1992), effectively performing *Semantic Compression* on the reasoning search space to mitigate hallucination risks.

Inspired by this separation of concerns, we propose Contract-Coding, a novel paradigm designed for Autonomous Symbolic Grounding. At its core is the *Language Contract*, a formalized consensus that transforms vague "vibes" into a deterministic *Single Source of Truth* (SSOT). Unlike transient prompt instructions, our Contract functions as a *Symbolic Constraint Projection* (Sheth et al., 2023): it autonomously compresses the high-dimensional

[*] Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 8187–8206
July 2-7, 2026 ©2026 Association for Computational Linguistics

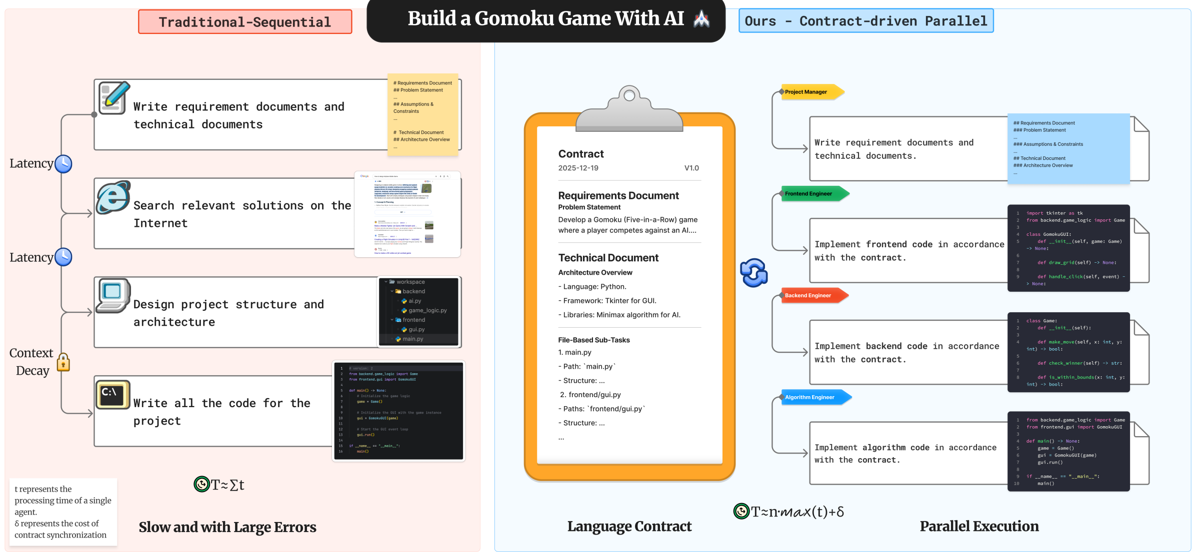

Figure 1: Linear vs. Contract-Driven Parallelism. Unlike sequential workflows that suffer from context accumulation (Left), our Language Contract serves as a topological hypervisor, decoupling dependencies to enable parallel execution and global consistency via a hierarchical graph (Right).

intent space into orthogonal constraint subspaces. This decoupling allows us to model the repository as a Contract-Driven Hierarchical Graph (HEG). By using the Contract as a topological hypervisor, we reduce the informational dependency depth of tasks. This allows agents to execute in parallel, conditioned solely on the symbolic Contract rather than on the unstable implementation history, verified by a Contract-Guided Auditing mechanism.

Our contributions are threefold:

- We formalize the Language Contract mechanism, which autonomously operationalizes high-level user intents into structured symbolic paradigm, bridging the gap between ambiguous Vibe Coding and rigorous Spec-Coding.
- We develop the Contract-Driven Hierarchical Graph paradigm. By utilizing the Contract as a semantic basis, we decrease informational dependency depth, unlocking massive *Architectural Parallelism* as a natural consequence of symbolic decoupling.
- We demonstrate that our approach has a great academic evolvement in greenfield repository generation. Crucially, we identify a Sub-linear Context Scaling effect: while repository scale expands significantly, the symbolic contract exhibits sub-linear growth, effectively compressing the reasoning context for complex systems.

2 Related Work

LLMs for Code and Autonomous Agents. Large Language Models (LLMs) have demonstrated profound capabilities in software engineering (OpenAI et al., 2024; Li et al., 2023b; Rozière et al., 2024; Guo et al., 2024; Xu et al., 2024). Beyond solitary code generation, the field has pivoted toward autonomous agents capable of planning and tool use (Xi et al., 2023; Wang et al., 2024a; Shen et al., 2023; Patil et al., 2023). Representative systems include general-purpose solvers like AutoGPT (Yang et al., 2023) and ToolFormer (Schick et al., 2023), as well as verticalized coding agents such as CodeAgent (Zhang et al., 2024), ToolCoder (Ding et al., 2025), OpenHands (Wang et al., 2025), and SWE-agent (Yang et al., 2024), which primarily focus on local bug-fixing or single-file synthesis.

Multi-Agent Collaboration Topologies. To mitigate the hallucination and context constraints of single agents, multi-agent systems (MAS) have been widely explored (Talebirad and Nadiri, 2023; Li et al., 2023a). Early frameworks such as MetaGPT (Hong et al., 2024) and ChatDev (Chen et al., 2024) adopt linear "waterfall" workflows, effectively acting as a sequential Chain-of-Thought. More recent works emphasize dynamic or adaptive interaction structures (Zhang et al., 2025; Li et al., 2024; Niu et al., 2025), including SPP (Wang et al., 2024b), AgentVerse (Chen et al., 2023), DyLAN

(Liu et al., 2024b), and PriorDynaFlow (Lin et al., 2025). However, these systems still rely on conversational routing, where information propagates linearly, leading to the "Context Exhaustion" bottleneck identified in our theoretical analysis.

Spec-Driven Synthesis vs. Contract-Driven Orchestration. Unlike specification-driven methods (Ma et al., 2025; Zhao et al., 2025) that treat constraints as static priors (Mao et al., 2025), Contract-Coding targets the spec-less "Vibe Coding" setting. We distinguish ourselves by (1) **Genesis**: autonomously synthesizing the contract from ambiguous intent rather than requiring formal inputs; and (2) **Topology**: elevating the contract to an active *Hypervisor*. (3) **Information Hiding**: utilizing the Contract as a semantic barrier. This **compresses** the reasoning scope effectively preventing the context saturation typical of sequential baselines.

3 Decoupling via Contract Constraints

In this section, we formalize the repo-level generation task and analyze the probabilistic failure modes of existing paradigms. We demonstrate that the prevailing sequential approach suffers from inherent entropic divergence, which we resolve via a latent symbolic formulation.

3.1 Task Formulation

We define the repository generation task as maximizing the likelihood of synthesizing a valid code repository $\mathcal{R} = \{f_1, f_2, ..., f_N\}$ given a high-level, under-specified user intent $\mathcal{I}$:

$$\mathcal{R}^* = \arg\max_{\mathcal{R}} P(\mathcal{R} \mid \mathcal{I}) \quad (1)$$

A valid $\mathcal{R}$ requires not only local syntactical correctness but also global *Symbolic Consistency*—meaning that any symbol (e.g., class 'Player') defined in file f_i must be correctly referenced by f_j, regardless of their generation order.

3.2 The Sequential Bottleneck

Standard multi-agent frameworks (e.g., MetaGPT, ChatDev) adopt a "Chain-of-Thought" topology. They approximate the joint probability using the probabilistic chain rule:

$$P(\mathcal{R} \mid \mathcal{I}) = \prod_{i=1}^{N} P(f_i \mid f_{<i}, \mathcal{I}) \quad (2)$$

where $f_{<i}$ represents the cumulative context history. While theoretically sound, this formulation introduces two systemic failures in the context of LLMs:

Sequential Error Propagation. Since f_i is conditioned on the raw implementation of $f_{<i}$, any latent defect ϵ in an early module $f_j (j \ll i)$ becomes part of the ground truth for all subsequent generation. The error term accumulates multiplicatively, causing the system to diverge from $\mathcal{I}$ toward a locally coherent but globally invalid state.

Context Exhaustion (The Signal-to-Noise Ratio). As N grows, the implementation context $f_{<i}$ expands linearly, inevitably exceeding the model's effective attention span. The informational density of the original intent $\mathcal{I}$ becomes diluted by verbose code details, leading to Symbolic Hallucination, where agents invent non-existent APIs to satisfy immediate local constraints.

3.3 Conditional Independence via Symbolic Grounding

To break this curse of dimensionality, we reformulate the generation process by introducing the Language Contract $\mathcal{C}$ as a discrete *Latent Symbolic Variable*.

Instead of treating $\mathcal{C}$ as mere documentation, we posit it as the Information Barrier between architectural intent and implementation details. The joint probability is re-factorized as:

$$P(\mathcal{R} \mid \mathcal{I}) \approx P(\mathcal{C} \mid \mathcal{I}) \prod_{i=1}^{N} P(f_i \mid \mathcal{C}) \quad (3)$$

This formulation introduces a fundamental topological shift:

Formalism 1 (Implementation Independence). Let $\mathcal{R}$ be the repository and $\mathcal{C}$ be the synthesized contract. Under the condition that $\mathcal{C}$ captures the sufficient interface semantics of $\mathcal{R}$(Guaranteed by the two-stage and audit mechanism), the implementation of any module f_i is conditionally independent of the raw implementation of other modules $\mathcal{R}_{\setminus i}$:

$$P(f_i \mid \mathcal{R}_{\setminus i}, \mathcal{C}) = P(f_i \mid \mathcal{C}) \quad (4)$$

This implies two critical properties:

1. **Decoupled Execution:** The generation of f_i depends solely on $\mathcal{C}$, not on the $O(N)$ volume of other source files.

2. **Sub-linear Context Scaling:** While strict independence assumes an immutable $\mathcal{C}$, in practice, $\mathcal{C}$ serves as a compressed "Global View." Our empirical observations (Figure 4) indicate that the symbolic tokens $|\mathcal{C}|$ grow sub-linearly relative to implementation tokens $|\mathcal{R}|$. Thus, agents maintain global awareness without suffering from context exhaustion.

4 The Contract-Coding Paradigm

Building upon the latent variable formulation, we implement CONTRACT-CODING. As illustrated in Figure 1, the framework operationalizes the transition from vague intent to verified code through three distinct phases: (1) Symbolic Projection, (2) Topological Orchestration, and (3) Active Contract Auditing.

4.1 Symbolic Projection

The objective is to synthesize a high-fidelity initial Contract $\mathcal{C}_0$ from intent $\mathcal{I}$. To bridge the gap between human-readable requirements and machine-executable constraints, we formalize this process as a Discrete Symbolic Evolution (DSE) over a hierarchical state space.

Hierarchical Contract Definition. We define the Language Contract $\mathcal{C}$ as a dual-layer entity, acting as the interface between the neural agents and the symbolic system:

- **The Constraint Projection (Structure):** Physically, $\mathcal{C}$ is structured as a collection of disjoint sections $\mathcal{C} = \{S_{\text{REQ}}, S_{\text{API}}, \dots\}$, organized into Product and Technical tiers. These sections provide the high-dimensional natural language context necessary for LLM reasoning.
- **The Executable Kernel (Symbolic):** For topological orchestration, we project the API Specification section (S_{API}) into a rigorous logical kernel $\mathcal{K} = \langle \mathcal{N}, \Sigma, \Delta \rangle$. Here, $\mathcal{N}$ maps functional modules to file paths, Σ defines strict type signatures, and Δ represents the state space.

This distinction is vital: while agents read the *Constraint Projection* to understand "why", the execution graph (HEG) operates solely on the *Executable Kernel* to determine "when" and "how".

Discrete Mutation Primitives. To prevent the semantic drift inherent in free-form text generation, we restrict the construction of $\mathcal{C}$ to a set of atomic mutation primitives $\mathbb{O} = \{\text{ADD}, \text{UPDATE}\}$. At any step t, an agent issues an action $a_t = \langle \text{op}, \text{sec}, \delta \rangle$, where $\text{sec} \in \mathcal{C}$ is the target section and δ is the structured content. The transition function applies this mutation: $\mathcal{C}_{t+1} = T(\mathcal{C}_t, a_t)$. Crucially, any change to the text triggers an immediate re-projection of the kernel $\mathcal{K}$. If an action results in a kernel that violates graph acyclicity or type safety, the system intercepts and rejects it before it pollutes the global state.

Encoding Non-Functional Requirements (NFRs) Beyond functional signatures, NFRs such as security protocols and complexity constraints are enforced via a hierarchical prompt-based mechanism within the Language Contract. Global constraints (e.g., global thread pool policies or shared security standards) are defined in Section 1.3 (Constraints) and Section 2.2 (Global Shared Knowledge). For module-specific NFRs, requirements are embedded in the `Description` field of the “Symbolic API Specifications” during the generation phase. The Code Reviewer agent subsequently audits implementations against these specific natural language prompts to ensure compliance.

Two-Stage Initialization Protocol. Leveraging the DSE mechanism, we mitigate "one-shot" hallucinations via a streamlined pipeline:

1. **Proposal (Generator):** The Generator Agent analyzes $\mathcal{I}$ and emits a sequence of mutation actions to construct the preliminary contract $\mathcal{C}_{draft}$. This effectively discretizes the continuous Vibe manifold into the structured state space.
2. **Rectification (Discriminator):** The Discriminator Agent performs a one-pass audit on $\mathcal{C}_{draft}$. It verifies architectural soundness (e.g., ensuring Δ is connected and acyclic) and enforces a *Completeness Constraint*—rejecting any module description in $\mathcal{K}$ that lacks sufficient detail.

This phase may be imperfect. Residual ambiguities are propagated to the *Topological Orchestration* phase, where they are resolved dynamically by downstream execution agents (e.g., Code Reviewer) via runtime ‘Update’ actions to the Contract.

4.2 Topological Orchestration (The HEG)

Traditional multi-agent systems often rely on hard-coded chains. In contrast, we propose a Dynamic

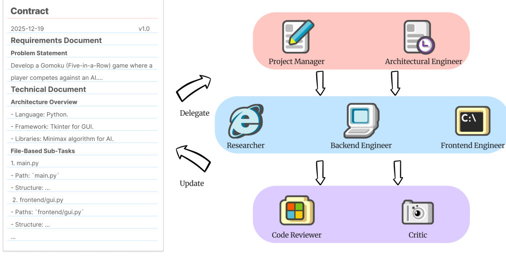

Figure 2: Contract-driven execution. The Contract defines the global state matrix, which drives the Hierarchical Execution Graph (HEG) to schedule parallel agents.

Hierarchical Execution Graph (HEG, Figure 2), where the execution topology $G^{(t)}$ is strictly derived from the Contract.

Let $\mathcal{T} = \{\tau_1, \dots, \tau_K\}$ be the set of atomic tasks defined in $\mathcal{C}$. Each task τ_i possesses a lifecycle status $\sigma_i \in \{\textsc{Todo}(\mathrm{T}), \textsc{Done}(\mathrm{D}), \textsc{Error}(\mathrm{E}), \textsc{Verified}(\mathrm{V})\}$. The HEG functions as a state-conditional scheduler Φ:

$$\Phi(\tau_i) = \begin{cases} \text{Worker}(\tau_i, \mathcal{C}) & \text{if } \sigma_i \in \{\mathrm{T}, \mathrm{E}\} \\ \text{Verifier}(\tau_i, \mathcal{C}, \mathcal{I}_{impl}) & \text{if } \sigma_i = \mathrm{D} \\ \emptyset & \text{if } \sigma_i = \mathrm{V} \end{cases} \tag{5}$$

This mapping induces three distinct operational modes:

Generative Expansion (Parallel Implementation). For tasks marked as TODO or ERROR, the system instantiates a worker node. Crucially, the input context provided to the worker is $\langle \tau_i, \mathcal{C} \rangle$. By providing the Full Contract $\mathcal{C}$ rather than raw code, we enable the agent to understand global dependencies while avoiding the noise of implementation details. This validation Formalism 4 allows multiple workers to execute in parallel without lock-step synchronization.

Critical Evaluation (Normative Verification). When a task transitions to DONE, the graph spawns a *Critic* node. This agent receives the implementation $\mathcal{I}_{impl}$ and validates it against the Contract $\mathcal{C}$. The objective is strictly normative: determining whether the code fulfills the contract signature, not whether it "looks good."

Convergence. The process iterates until $\forall \tau_i, \sigma_i = \textsc{Verified}$. This design eliminates explicit "dispatch" instructions; the workflow is an emergent property of the repository's state resolution. Our empirical observation is that the *Critic Evaluation* efficiently resolve most of the issues. As a final safety mechanism to ensure decidability, we can impose a Maximum Topological Depth T_{max}. If the system fails to converge within T_{max} layers, it terminates gracefully, yielding the current best-effort repository. This ensures that the system is bounded and avoids infinite resource consumption. To guarantee termination, we enforce a hard limit T_{max} (Maximum Topological Depth). Our empirical analysis suggests that the contract-first paradigm is highly stable: over 90% of inter-module conflicts are resolved during the initial planning, with remaining inconsistencies typically converging within a single iteration of the Active Auditing loop.

This validation of Formalism 4 allows multiple workers to execute in parallel without lock-step synchronization. The complete orchestration procedure, including the state transitions and parallel dispatching, is formalized in Algorithm 1.

4.3 Active Contract Auditing

To close the loop between execution and state, we introduce a deterministic Contract Auditor. Unlike passive loggers, the Auditor functions as the system's Homeostatic Controller. It continuously monitors the graph state and triggers *Topological Interventions* when deviations are detected.

Structural Alignment ($E(\mathcal{C})$). This metric enforces that the implementation physically aligns with the Contract. Let U be the set of existing file units.

$$E(\mathcal{C}) = \frac{1}{|\mathcal{T}|} \sum_{\tau \in \mathcal{T}} \mathbb{I}(\exists u \in U : \text{match}(\tau, u)) \tag{6}$$

Intervention: If $E(\mathcal{C})$ fails (i.e., a required symbol is missing), the Auditor performs Task Injection. It dynamically inserts a new implementation node into the HEG for the next time step, forcing the system to fill the "Hollow Skeleton."

State Synchronization ($S(\mathcal{C})$). The Auditor synchronizes high-level planning with low-level execution outcomes by parsing agent logs.

$$S(C) = \{\sigma_i^{(t+1)} | \sigma_i^{(t+1)} \leftarrow \text{Audit}(\sigma_i^{(t)}, \text{Output}_{\mathcal{A}})\} \tag{7}$$

Intervention: A failure here triggers Status Regression. For example, if a *Critic* agent flags a logic bug, the Auditor regresses the task status from

Fault Type	MetaGPT	OpenHands	Ours
Interface Mismatch	45%	23%	**0%**
Architecture Crash	25%	10%	**0%**
Context Exhaustion	25%	20%	**0%**
Local Logic Errors	55%	30%	30%

Table 1: Quantitative Fault Analysis on Roguelike. Our method eliminates structural failures (Mismatch, Crash, Exhaustion), with remaining errors strictly localized to logic implementation.

DONE to ERROR. This automatically triggers the HEG to reschedule the owner agent for repair.

Consistency Control ($V(\mathcal{C})$). Consistency is defined as the alignment between the Contract's recorded signatures and the actual workspace code:

$$V(\mathcal{C}) = \bigwedge_{\tau \in \mathcal{T}} (\text{state}_{\mathcal{C}}(\tau) \equiv \text{state}_{W}(\tau)) \quad (8)$$

Intervention: If a mismatch is detected (e.g., an agent silently modifies a function signature), the Auditor enforces Normative Alignment. It rejects the invalid state transition and injects a *Synchronization Task*, compelling the agent to either revert the code or formally propose a Contract Amendment (Update Action) to legitimize the change.

5 Experiments

We evaluate CONTRACT-CODING on a *Complexity Spectrum* of five greenfield repositories, ranging from simple algorithmic scripts to complex, event-driven systems. Our primary goal is to verify if the proposed *Symbolic Decoupling* can break the "Inverse Scalability Law" typically observed in autonomous agents.

5.1 Experimental Setup

Benchmarks: The Greenfield-5 Suite. Benchmarks: The Greenfield-5 Suite. We propose **Greenfield-5** as a core contribution to evaluate 0-to-1 repository synthesis from high-level intents. Unlike maintenance-oriented benchmarks like SWE-bench(Jimenez et al., 2024) that focus on local patching, Greenfield-5 targets architectural orchestration and global consistency across 10–25 interdependent files. The suite spans a complexity spectrum: (1) Gomoku (Logic); (2) Plane Battle and Snake++ (Event-Driven); (3) City Sim (Resource Management); and (4) Roguelike (System Architecture).. Full specifications are in Appendix F.

Baselines. We compare against: (1) **Commercial AI IDEs** (Lingma, Trae, Gemini Studio) serving as upper-bound references; and (2) **Academic Frameworks** (MetaGPT, ChatDev, FLOW, OpenHands) representing sequential multi-agent paradigms.

Metrics. We report "Success Rate" (a composite of Executability, Interactivity, and Rule Adherence) and "Efficiency" (Wall-clock Time). Additionally, "File Count" tracks code bloat. Results represent the mean of 10 independent runs.

Ablations. To validate our topological contributions, we compare the full model against: (1) **w/o HEG** (Sequential execution); (2) **w/o Contract** (Removing the symbolic layer); and (3) **Model Agnosticism** (Replacing the backbone with Qwen-Plus). Detailed configurations are in Appendix C.

5.2 Overall Success & Scalability(RQ1)

Table 2 reports performance across the Greenfield-5 suite. We observe three key trends:

The "Complexity Wall" for Legacy Agents. While traditional multi-agent frameworks (MetaGPT, ChatDev, FLOW) perform adequately on logic-centric tasks (Gomoku), they suffer from *Architectural Collapse* on multi-module challenges. Even the current SOTA OpenHands, despite achieving 100% success on Gomoku, experiences a significant performance drop-off to 50% on Snake++ and 30% on Roguelike. This highlights their struggle with "Context Saturation" and managing deep inter-file dependencies inherent in repo-level synthesis.

Efficiency and Structural Rigor vs. Commercial SOTA. Commercial AI IDEs establish a strong upper bound, achieving high Overall Success rates through proprietary optimizations. Notably, Gemini Studio demonstrates competitive speed (e.g., 49.3s on City Sim) and success (63% on Roguelike). We hypothesize this leverages its "ultra-long context window", enabling a single powerful model to process extensive cross-file information, thus effectively circumventing traditional linear reasoning bottlenecks. However, even this brute-force context scaling shows diminishing returns, with Gemini's performance declining on more complex tasks. Other commercial tools, like Trae, often incur extreme latency ($>$ 350s) and significant code bloat (Files=19.0).

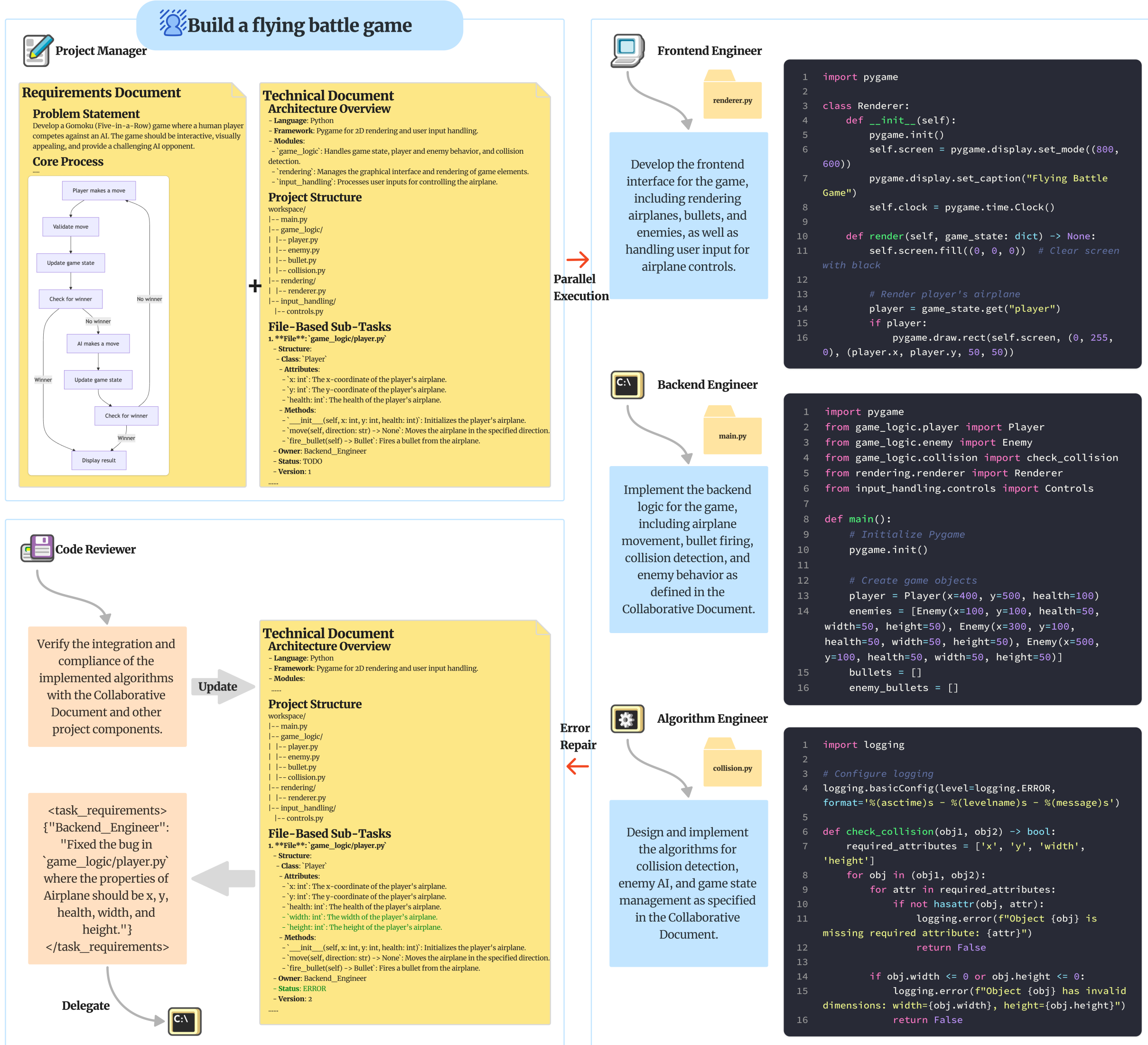

Figure 3: Contract-Guided Self-Healing Case Study. When divergent assumptions (e.g., missing attributes) occur between parallel Backend/Algorithm agents, Contract Auditing detects schema violations during merging and triggers Normative Alignment to auto-correct semantic mismatches.

Limits of Parallelism. On the scalability stress-test (Roguelike), our fully parallel approach sees a performance drop (47% success rate) compared to the SOTA (CodeBuddy), reflecting the challenge of integrating 25+ coupled files without intermediate feedback. However, distinct from the "Silent Failures" in legacy baselines, our failures are mostly localized logic errors. A detailed forensic analysis of failure modes across all methods (e.g., MetaGPT's "Hollow Skeleton" vs. Trae's "Logical Detachment") is provided in Appendix D.

5.3 The Context Scaling Effect(RQ2)

Is the efficiency gain purely coincidental? Figure 4 illustrates the token dynamics across the five tasks.

As the project complexity grows (T_{proj}), the size of the Language Contract (T_{cont}) grows at a much slower rate. This confirms that our *Constraint Projection* mechanism effectively compresses the high-dimensional intent space. For the *Roguelike* challenge (8,857 tokens), our agents only need to attend to a ~1,900-token contract, avoiding the “Lost-in-the-Middle” phenomenon plaguing standard RAG or long-context approaches. This 4.6x compression effectively downgrades a repository-level task to a series of manageable single-file tasks.

5.4 Case Study: Conflict Repair(RQ3)

To demonstrate the system's robustness against initial contract ambiguities and logical errors, we analyze a real-world Conflict-Repair cycle from the Plane Battle task (visualized in Figure 3). During

Method	Gomoku			Plane Battle			City Sim			Snake++			Roguelike		
	Overall	Files	Time	Overall	Files	Time	Overall	Files	Time	Overall	Files	Time	Overall	Files	Time
Legacy Multi-Agent Frameworks															
MetaGPT	70	5.2	126s	50	4.0	129s	30	3.0	185s	10	4.0	210s	10	10.9	261s
ChatDev	93	4.4	88s	90	5.1	79s	45	6.0	140s	25	7.2	165s	10	13.9	144s
FLOW	96	1.0	155s	76	1.0	76s	60	1.0	110s	30	1.0	130s	0	1.0	90s
OpenHands	100	3.3	173s	90	4.9	182s	63	7.3	296s	50	13.7	267s	30	14.1	311s
Commercial AI IDEs (SOTA)															
Lingma	**100**	1.0	127s	86	1.4	125s	75	8.0	220s	60	12.0	310s	33	24.7	620s
Trae	70	13.7	393s	80	14.5	411s	85	19.0	350s	65	22.0	412s	47	23.9	218s
Gemini Studio	100	10.3	103s	100	9.6	165s	87	13.0	49s	83	15.5	108s	63	16.6	72.1s
CodeBuddy	96	2.8	91s	**100**	3.0	92s	60	14.0	196s	60	11.7	414s	40	19.6	422s
Ours (Autonomous)															
Ours w/o HEG	**100**	4.0	205s	**100**	6.1	201s	85	11.2	480s	78	16.0	465s	47	25.0	510s
Ours (Full)	**100**	4.0	136s	**100**	6.2	117s	87	11.2	257s	**80**	16.2	198s	47	14.2	232s

Table 2: **Main Results on Repository-Level Generation.** Comparison across five tasks of decreasing complexity. **Overall**: Average of Executability, Interactivity, and Rule Adherence. **Files**: Generated file count (closer to ground truth is better). **Time**: End-to-end latency.

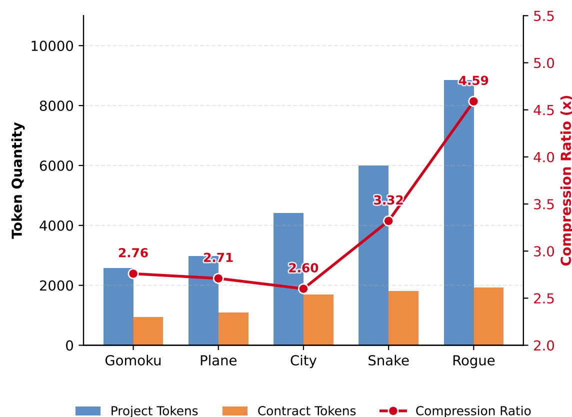

Figure 4: **Token Decoupling Analysis.** As project complexity grows, the Language Contract size remains stable. Red markers show the compression efficiency.

the parallel execution phase, a divergence occurred due to under-specified requirements: the Algorithm Engineer, implementing collision logic, implicitly assumed the existence of `width` and `height` attributes in the `Player` class. Conversely, the Backend Engineer, adhering strictly to the initial Contract, implemented only basic attributes (`x`, `y`, `health`).

This divergence was autonomously resolved via the system mechanism. The Code Reviewer detected a semantic mismatch between the Algorithm's invocation (demand) and the Backend's definition (supply). Crucially, recognizing this as an incorrectness in the Contract itself rather than a mere coding error, the Code Reviewer retroactively patched the Contract definition, formally adding the spatial dimensions to the schema as the new Single Source of Truth. The task was flagged as `ERROR`, triggering a targeted re-dispatch to the Backend Agent with the directive: "*Fix the schema definition; Player requires spatial dimensions...*" This highlights the systemic robustness of our paradigm: by synergizing the Language Contract with reflective auditing, the framework effectively prevents error propagation of early logical flaws. This dynamic self-healing capability serves as the decisive factor enabling our high task overall success rate, ensuring that local schema inconsistencies are resolved internally rather than propagating into global system failures.

5.5 Scalability Analysis: The Roguelike Challenge (RQ4)

To probe the upper limits of autonomous architecture, we introduce a complex Roguelike task requiring 15-25 interdependent files. Table 4 highlights the divergence between paradigms.

Systemic Collapse in Academic Baselines. Current academic models exhibit severe architectural decoupling at scale. As detailed in Appendix D, MetaGPT and ChatDev suffer from *Implementation Sparsity* and *Grounding Failures*, respectively, while FLOW faces *Context Collapse*, rendering them unable to synthesize valid multi-file projects. OpenHands may encounter *Lost-in-the-Middle*.

Commercial SOTA: The Resource-Efficiency Trade-off. Commercial IDEs (e.g., Gemini Studio) establish a strong upper bound (63% success). However, this relies on massive context windows (>200k tokens) and proprietary infrastructure. In stark contrast, our experiments were strictly confined to a 16k token limit. That CONTRACT-CODING achieves 100% Structural Integrity under this constraint validates our core hypothesis: *Architectural Decoupling is a more scalable path than Brute-Force Context Scaling.* Our method effectively "simulates" infinite context via symbolic compression, whereas other models (e.g., Gemini, Lingma) suffer from *Probabilistic Drift* or infinite recursive loops when context overflows.

Architectural Integrity over Algorithmic Fidelity. Although our method lags in absolute overall success (47%), the nature of failure is distinct. Our failures are strictly confined to "Local Logic Errors" (e.g., suboptimal heuristics) driven by base model limitations, not architectural collapse. This distinction is critical for ASE: we prioritize Structure over Detail. In real-world workflows, fixing local bugs within a perfect architecture is trivial; refactoring a "working" but structurally chaotic codebase is prohibitively expensive.

6 Conclusion

Contract-Coding, with Language Contract as SSOT, decouples architectural complexity from implementation details and enable efficient parallel execution. Validation on the Greenfield-5, compared with SOTA baselines, this method achieves an up to 4.6× token compression ratio as well as an overall improvement in success rate. Our work concludes that more complex vibe coding can be realized through auditable structural-symbolic contracts, instead of relying on larger context windows.

7 Limitations

Despite the performance gains, our framework has several limitations that provide avenues for future research. First, our evaluation is currently focused on the five tasks in Greenfield-5; while these provide rigorous stress tests for architectural coupling, evaluating generalizability across larger, more heterogeneous systems remains future work. Second, the multi-agent auditing loop introduces higher computational latency and token consumption, a trade-off we believe is justified for complex, high-stakes architectural generation where reliability is paramount.

Second, regarding Benchmark Alignment, we observe a significant gap between existing evaluation suites and the requirements of autonomous repository synthesis. Current benchmarks like SWE-bench (Yang et al., 2024) primarily focus on *Brownfield Repair* (i.e., surgical bug-fixing in mature codebases). In contrast, our work targets Greenfield Synthesis from high-level intents (often referred to as "vibe coding" in developer communities), which emphasizes architectural orchestration and cross-module consistency. Due to the lack of standardized benchmarks for multi-file generative synthesis, we utilized curated game-engine tasks. We are committed to developing a specialized, automated benchmark in future work to formally evaluate the Scaling-up laws of contract-driven paradigms across larger, heterogeneous software systems (> 100 files).

Finally, our baseline comparisons involve commercial "black-box" IDEs; although we standardized the autonomous environment, their evolving internal models may introduce temporal variance in reproducibility.

8 Ethical Considerations

8.1 Dual-Use Risks and Safety.

The advancement of autonomous software engineering lowers the technical barrier for software creation, which inevitably raises concerns regarding the rapid generation of malicious code (e.g., malware or automated hacking scripts). However, unlike end-to-end “black box” approaches that generate opaque deliverables, our *Contract-Driven Paradigm* inherently promotes Accountability. By enforcing a human-readable *Language Contract* as the intermediate control layer, our framework provides a mandatory auditing checkpoint, making it significantly easier to detect and intercept malicious intent before execution.

8.2 Carbon Footprint.

We acknowledge that multi-agent frameworks involve redundant communication, leading to higher token consumption compared to single-pass generation. We advocate for the responsible use of such systems, recommending their deployment primarily for complex, architectural-level tasks where the high reliability justifies the computational cost.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Qian Chen, Liu Wei, Liu Hongzhang, Chen Nuo, Dang Yufan, Li Jiahao, Yang Cheng, Chen Weize, Su Yusheng, Cong Xin, Xu Juyuan, Li Dahai, Liu Zhiyuan, and Sun Maosong. 2024. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. *Preprint*, arXiv:2308.10848.

Hanxing Ding, Shuchang Tao, Liang Pang, Zihao Wei, Jinyang Gao, Bolin Ding, Huawei Shen, and Xueqi Cheng. 2025. Toolcoder: A systematic code-empowered tool learning framework for large language models. *Preprint*, arXiv:2502.11404.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023a. Camel: Communicative agents for "mind" exploration of large language model society. *Preprint*, arXiv:2303.17760.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023b. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. 2024. Autoflow: Automated workflow generation for large language model agents. *CoRR*, abs/2407.12821.

Yi Lin, Lujin Zhao, and Yijie Shi. 2025. (p)rior(d)yna(f)low: A priori dynamic workflow construction via multi-agent collaboration. *Preprint*, arXiv:2509.14547.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024a. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2024b. A dynamic LLM-powered agent network for task-oriented agent collaboration. In *First Conference on Language Modeling*.

George Ma, Anurag Koul, Qi Chen, Yawen Wu, Sachit Kuhar, Yu Yu, Aritra Sengupta, Varun Kumar, and Murali Krishna Ramanathan. 2025. Specagent: A speculative retrieval and forecasting agent for code completion. *Preprint*, arXiv:2510.17925.

Zhenyu Mao, Jacky Keung, Fengji Zhang, Shuo Liu, Yifei Wang, and Jialong Li. 2025. Towards engineering multi-agent llms: A protocol-driven approach. *Preprint*, arXiv:2510.12120.

B. Meyer. 1992. Applying 'design by contract'. *Computer*, 25(10):40–51.

Boye Niu, Yiliao Song, Kai Lian, Yifan Shen, Yu Yao, Kun Zhang, and Tongliang Liu. 2025. Flow: Modularized agentic workflow automation. In *The Thirteenth International Conference on Learning Representations*.

Shinn Noah, Cassano Federico, Gopinath Ashwin, Narasimhan Karthik, and Yao Shunyu. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pages 8634–8652. Curran Associates, Inc.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Memgpt: Towards llms as operating systems. *Preprint*, arXiv:2310.08560.

D. L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *Preprint*, arXiv:2305.15334.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Preprint*, arXiv:2303.17580.

Amit Sheth, Kaushik Roy, and Manas Gaur. 2023. Neurosymbolic ai – why, what, and how. *Preprint*, arXiv:2305.00813.

Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *Preprint*, arXiv:2306.03314.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024a. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6).

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. 2024b. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. *Preprint*, arXiv:2307.05300.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

Jiaxin Wen, Jian Guan, Hongning Wang, Wei Wu, and Minlie Huang. 2025. Codeplan: Unlocking reasoning potential in large language models by scaling code-form planning. In *The Thirteenth International Conference on Learning Representations*.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, and 10 others. 2023. The rise and potential of large language model based agents: A survey. *Preprint*, arXiv:2309.07864.

Yiheng Xu, Hongjin SU, Chen Xing, Boyu Mi, Qian Liu, Weijia Shi, Binyuan Hui, Fan Zhou, Yitao Liu, Tianbao Xie, Zhoujun Cheng, Siheng Zhao, Lingpeng Kong, Bailin Wang, Caiming Xiong, and Tao Yu. 2024. Lemur: Harmonizing natural language and code for language agents. In *The Twelfth International Conference on Learning Representations*.

Hui Yang, Sifu Yue, and Yunzhong He. 2023. Autogpt for online decision making: Benchmarks and additional opinions. *Preprint*, arXiv:2306.02224.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Preprint*, arXiv:2405.15793.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. 2023. Parsel: Algorithmic reasoning with language models by composing decompositions. In *Advances in Neural Information Processing Systems*, volume 36, pages 31466–31523. Curran Associates, Inc.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. 2025. Cut the crap: An economical communication pipeline for LLM-based multi-agent systems. In *The Thirteenth International Conference on Learning Representations*.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *ACL*.

Qianhui Zhao, Li Zhang, Fang Liu, Junhang Cheng, Chengru Wu, Junchen Ai, Qiaoyuanhe Meng, Lichen

Zhang, Xiaoli Lian, Shubin Song, and Yuanping Guo. 2025. Towards realistic project-level code generation via multi-agent collaboration and semantic architecture modeling. *Preprint*, arXiv:2511.03404.

A Appendix: Implementation Details of Conflict Control

While the Hierarchical Execution Graph enables parallel execution, the inherent asynchrony introduces the risk of "lost updates." We mitigate this via a **Differential Interval Analysis** mechanism.

The system enforces a Synchronized Commit Protocol where updates from the current layer are aggregated only after all agents have completed tasks. We utilize the initial Contract state C_{base} as the **Unique Coordinate System**. By calculating line offsets relative solely to this immutable baseline rather than dynamic intermediate states, we eliminate Positional Coupling (errors caused by index shifting).

Qualitative Analysis: Why Union-First? We deliberately employ a **Union-First Strategy** for resolving semantic conflicts in the Contract, as opposed to a naive "Last-Write-Wins" policy.

- **Information Preservation:** In LLM-generated content, different agents may discover orthogonal requirements for the same interface (e.g., Agent A adds 'width', Agent B adds 'color'). A Last-Write-Wins policy would silently discard one agent's contribution, leading to downstream "Variable Not Found" errors.
- **Auditing Efficiency:** The Union-First approach preserves the superset of all proposed constraints. While this may introduce redundancy, removing redundant lines during the Audit phase (Reduction) is computationally easier and less prone to hallucination than regenerating missing logic from scratch (Synthesis).

We acknowledge that for extremely large teams, this strategy could lead to contract bloating. Future work will explore *Conflict-Aware Merging* using a dedicated arbiter LLM to semantically deduplicate the union set.

B Appendix: Experimental Environment

Hardware Specifications. Experiments were conducted on a MacBook Pro (M2 Pro) equipped with 32GB of unified memory and a 512GB SSD, running macOS 15. The explicit hardware constraint ensures that our efficiency metrics reflect performance on consumer-grade devices rather than high-end server clusters.

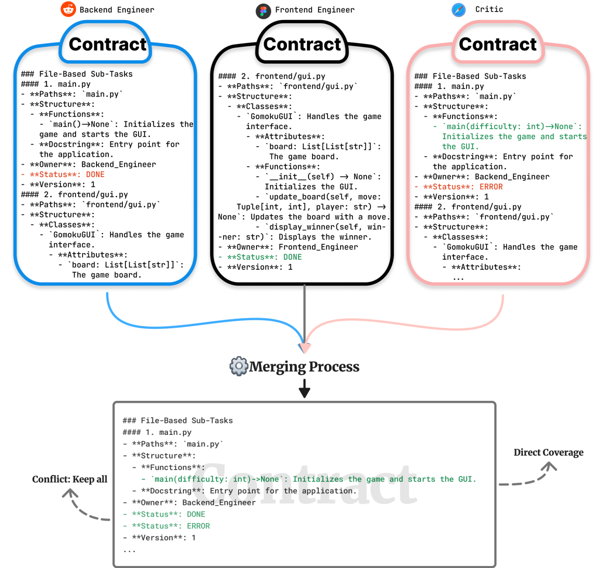

Figure 5: Conflict Control via Differential Interval Analysis. Using C_{base} as an immutable baseline, we decompose updates into Atomic Patches. Semantic conflicts are resolved via a Union-First Strategy, prioritizing information preservation to prevent silent data loss.

Hyperparameters. Except for the model ablation experiment, all academic method experiments utilized `gpt-4o-2024-11-20` with a temperature of 0.0 to ensure determinism and a context window limit of 16k tokens. The specific versions and configurations of the commercial AI IDE baselines are detailed in Table 3.

C Appendix: Ablation Experiments

C.1 Contract Auditing

To validate the Language Contract as an essential structured interface specification, we removed the global contract mechanism ("Ours w/o Contract"). In this setting, agents operate without a Single Source of Truth.

Impact on Performance. As shown in Table 7, removing the Contract causes a precipitous drop in Functional Success (100% → 65% in Plane Battle).

Diagnostic Analysis. To identify the cause, we utilize the Task Existence $E(C)$ metric. Without the Contract to explicitly enumerate and track subtasks, the system suffers from severe "Task Forget-

AI IDE	Version	VSCode Ver.	Backbone Model	Mode
Lingma	0.2.3	1.100.0	Default	Agent
CodeBuddy	4.1.1	1.100.0	GPT-5.1-Codex-Max	Craft
Trae	3.2.1	1.104.3	GPT-5 Medium	Solo Build
Gemini Studio	–	–	Gemini 3 Flash Preview	Build

Table 3: **Configuration of Commercial AI IDE Baselines.** Detailed specifications of the IDE versions, underlying VSCode engines, and backbone models used in our comparative evaluation. All AI IDEs choose their corresponding Vibe coding mode.

	Task 1: Gomoku (Logic)						Task 2: Plane Battle (Action)						
Method	**Exec**	**Inter**	**Rule**	**Succ.**	**Time(s)**	**Files**	**Exec**	**Inter**	**Rule**	**Succ.**	**Time(s)**	**Files**	**Parallel**
Legacy Multi-Agent Frameworks													
MetaGPT	70	70	70	70%	126.0	5.2	50	50	50	50%	128.5	4.0	×
ChatDev	100	90	90	93%	**87.9**	4.4	90	90	90	90%	79.4	5.1	×
FLOW	100	90	100	97%	155.0	1.0	100	80	50	77%	**75.7**	1.0	✓
OpenHands	100	100	100	100%	173.2	3.3	100	100	100	100%	182.0	4.9	×
Commercial AI IDEs													
Lingma	100	100	100	100%	127.0	1.0	90	90	80	83%	125.0	1.4	×
Trae	70	70	70	70%	393.0	13.7	80	80	80	80%	411.0	14.5	×
Gemini Studio	100	100	100	100%	103.0	10.3	100	100	100	100%	165.0	9.6	×
CodeBuddy	100	90	100	93%	91.0	2.8	100	100	100	100%	92.0	3.0	×
Ours (Autonomous)													
Ours w/o HEG	100	100	100	100%	205.0	4.0	100	100	100	100%	201.0	6.1	×
Ours (Full)	**100**	**100**	**100**	**100%**	136.0	4.0	**100**	**100**	**100**	**100%**	117.0	6.2	✓

Table 4: **Detailed Metrics: Basic Tasks.** Comparison on Logic-Oriented (Gomoku) and Action-Oriented (Plane Battle) tasks. Gemini Studio demonstrates high speed but produces bloated file structures compared to our optimized output.

ting," where the Existence rate drops to 82%. This confirms that the Contract is critical for preventing task omission in long-horizon generation; without it, parallel agents lose track of requirements, directly leading to system failure.

C.2 Model Agnosticism Study

To verify whether the performance of Contract-Coding stems from the architectural superiority of our paradigm rather than reliance on a specific proprietary model family (i.e., GPT-4o), we conducted a **Model Agnosticism** ablation. We replaced the underlying LLM with **Qwen-Plus**, a leading non-OpenAI model, while keeping all hyper-parameters and the Contract-Driven workflow unchanged.

Results. As detailed in Table 8, the framework maintained a **100% Functional Success Rate** on both tasks when powered by Qwen-Plus. This performance parity is significant: it confirms that the **Contract-Driven Hierarchical Graph** effectively lowers the reasoning threshold required for repository-scale generation. By decoupling high-entropy architectural planning into low-entropy, auditable interface constraints, our paradigm enables non-GPT-4 models to achieve SOTA performance. This proves that the system's robustness is derived from its *methodological topology*, not merely the latent knowledge of a specific proprietary LLM.

C.3 Analysis of Parallel Efficiency)

It is imperative to acknowledge that commercial AI IDEs (e.g., CodeBuddy, Lingma) benefit from significant Infrastructure Advantages, including proprietary low-latency inference engines, which are unavailable to open-source agent frameworks. Despite this physical disparity in raw token generation speed, our method achieves competitive end-to-end efficiency through Architectural Parallelism. As illustrated in Table 2, compared to the heavyweight IDE Trae (393.0s), which suffers from high latency and generates notably bloated code (Files=14.1), our method is over 3x faster (117.7s) while producing significantly cleaner structures (Files=6.2). Furthermore, while optimized tools like CodeBuddy exhibit faster raw execution (90.8s), but they are more inclined to generate simple code(Files=2.9).

Method	Task 3: City Sim (11 Files)						Task 4: Snake++ (16 Files)						
	Exec	Inter	Rule	Succ.	Time(s)	Files	Exec	Inter	Rule	Succ.	Time(s)	Files	Parallel
Legacy Multi-Agent Frameworks													
MetaGPT	50	20	20	20%	185	3.0	20	10	0	0%	210	4.0	×
ChatDev	60	40	35	30%	140	6.0	30	30	15	10%	165	7.2	×
FLOW	60	60	60	40%	110	1.0	30	30	30	0%	130	1.0	✓
OpenHands	100	40	50	63%	296.4	7.3	60	50	40	50%	266.8	13.7	×
Commercial AI IDEs													
Lingma	90	70	70	77%	446.8	10.6	80	50	50	60%	453.6	12.9	×
Trae	100	90	90	93%	231.2	15.3	90	90	90	90%	213.8	14.2	×
Gemini Studio	100	80	80	87%	49.3	13.0	100	90	70	83%	108.0	15.5	×
CodeBuddy	70	60	50	60%	195.8	14.0	70	60	50	60%	414	11.7	×
Ours (Autonomous)													
Ours w/o HEG	100	80	70	83%	480.0	11.2	100	70	70	80%	465.0	16.0	×
Ours (Full)	**100**	80	80	87%	257.0	11.2	**100**	70	70	80%	198.0	16.2	✓

Table 5: **Detailed Metrics: Intermediate Tasks.** Comparison on City Sim (Resource Management) and Snake++ (Event Driven). Note the sharp decline in *Gemini Studio*'s rule adherence due to context drift, while Contract-Coding maintains high fidelity.

Method	Arch. Integrity (%)		Func. Fidelity (%)			Overall Success	Time (s)	Qual. Files	Key Failure Mode
	Struc.	Dep.	Exec.	Inter.	Rules				
Legacy Multi-Agent Frameworks									
MetaGPT	60	40	20	10	0	10%	261.0	10.9	Implementation Sparsity
ChatDev	0	10	20	10	0	10%	144.2	13.9	Flat Structure Fallacy
FLOW	–	–	0	0	0	0%	90.6	1.0	Aggregative Collapse
OpenHands	80	50	70	10	10	30%	310.7	14.1	Lost-in-the-Middle
Commercial AI IDEs									
Lingma	100	60	40	40	20	33%	619.6	24.7	Logical Detachment
Trae	100	95	80	30	30	47%	217.9	23.9	Unreachable Logic
Gemini Studio	80	80	70	70	50	**63%**	**72.1**	16.6	Probabilistic Drift
CodeBuddy	100	88	60	40	20	40%	422.2	19.6	Calling Hallucination
Ours (Autonomous)									
Ours (Full)	**100**	92	**90**	20	30	47%	232.0	14.2	Local Logic Error

Table 6: **Detailed Results on Roguelike.** High-complexity tasks trigger distinct pathological behaviors in baselines.

Method	Overall Success (%)		Diagnostic	
	Plane Battle	Gomoku	Consistency ($V(C)$)	Existence ($E(C)$)
Ours w/o Contract	65	85	-	82%(Low)
Ours (Full)	**100**	**100**	**100% (High)**	**100% (High)**

Table 7: **Component Ablation Study.** Removing the Language Contract significantly impacts Overall Success and Task Existence.

Task	Backbone Model	Success Rate	Diff.	Outcome
Plane Battle	GPT-4o (Main) Qwen-Plus (Variant)	100% **100%**	- 0%	**Parity**
Gomoku	GPT-4o (Main) Qwen-Plus (Variant)	100% **100%**	- 0%	**Parity**

Table 8: **Robustness Across Model Families.** The framework achieves identical stability with Qwen-Plus, demonstrating model agnosticism.

By transforming the topological structure from a chain to a graph, our Hierarchical Execution Graph (HEG) effectively hides the latency of complex modules, enabling a rigorous, contract-driven multi-agent system to compete with infrastructure-optimized commercial products.

D Appendix: Forensic Analysis of Failure Cases

We conducted a manual code inspection of the failed repositories to identify the root causes of the performance gaps reported in RQ4. This section details the specific anti-patterns observed in each

Listing 1: The Structure of the Language Contract

```
# ==========================================
# SECTION 1: Product Requirement Document
# ==========================================
Product Requirement Document
  1.1 Project Overview |
    [High-level summary of the project goals and scope...]

  1.2 User Stories
    - "[User Story 1]"
    - "[User Story 2]"

  1.3 Constraints
    - "[Technical or Design Constraint 1]"
    - "[Constraint 2]"

# ==========================================
# SECTION 2: Technical Document
# ==========================================
Technical Document
  2.1 Project Structure
    - "[Directory/File Structure Tree]"

  2.2 Global Shared Knowledge
    - "[Shared Constants, Configs, or Global States]"

  2.3 Dependency Relationships
    - "[Mermaid]"

  2.4 Symbolic Api Specifications
    - File Path "[File Path]"
      Owner     "[Agent Name]"
      Version   "[Number]"
      Status    "[Status]"

      Classes
        - Class Name "[Class Name]"

          Attributes
            - Name "[Attribute Name]"
              Type "[Type]"
              Description "[Description]"

          Methods
            - Signature "def [Name]([Param Name]: [Type]) -> [Type]"
              Docstring "[Briefly explain logic and behavior]"
```

Figure 6: Visualizing the Language Contract. The Contract bridges the Product Requirements Documents (PRD) and Technical Manifold (API Specs). This structure allows the *Contract-Guided Auditing* mechanism to strictly validate implementation compliance via parsing, ensuring parallel context consistency.

baseline.

D.1 Forensics of Academic Baselines

MetaGPT: The "Hollow Skeleton" Phenomenon. Inspection of the repositories generated by MetaGPT revealed a disconnect between the Architect Agent's planning and the Engineer Agent's execution. In approximately 40% of cases, the directory structure was perfectly instantiated, yet critical logic files (such as `entities/player.py`) contained only placeholder comments or empty class definitions (e.g., `class Player: pass`). This suggests that without a persistent constraint like our Language Contract to enforce content density, downstream agents tend to minimize output length to satisfy the immediate prompt.

ChatDev: The Reflection-Action Gap. ChatDev exhibited a specific grounding failure related to file system operations. The generated code consistently utilized relative imports assuming a nested

Algorithm 1 Contract-Driven Orchestration & Auditing

Require: User Intent $\mathcal{I}$, Max Steps T

1: $\mathcal{C} \leftarrow$ SynthesizeContract($\mathcal{I}$) {Drafting & Rectification}
2: **while** $\mathcal{G}$ has unfinished nodes **do**
3:   *tasks* $\leftarrow$ GetReadyTasks($\mathcal{C}$) {Topological parallel set}
4:   **for each** $\tau_i \in$ *tasks* **do in parallel do**
5:     $File_i \leftarrow$ WorkerAgent($\tau_i, \mathcal{C}$) {Context is restricted to $\mathcal{C}$}
6:     $\delta \leftarrow$ Auditor.Compare($File_i, \mathcal{C}$.schema)
7:     **if** $\delta = \emptyset$ **then**
8:       $\mathcal{R}$.Commit($File_i$); MarkDone(τ_i)
9:     **else if** δ is Critical Mismatch **then**
10:       Reject($File_i$); Feedback(τ_i, δ)
11:     **else**
12:       $\mathcal{C}$.Update(δ) {Adaptive Contract Patching}
13:     **end if**
14:   **end for**
15: **end while**

directory structure (e.g., `from core.game import Game`), yet the agent saved all files physically into the root directory. Logs indicate that the Test phase correctly identified the `ModuleNotFoundError`, but the agent repeatedly attempted to patch the Python import statements rather than performing the necessary OS-level `mkdir` actions. This highlights a critical limitation in the agent's Action Space.

FLOW: Aggregative Context Collapse. FLOW's failure mode was characterized by the inability to maintain file separation. The final aggregation node frequently concatenated logic from multiple parallel sub-tasks into a single, incoherent script. This resulted in massive namespace conflicts and syntax errors, effectively collapsing the intended multi-module architecture into a non-executable monolithic block.

D.2 Forensics of Commercial AI IDEs

Gemini AI Studio: Probabilistic Drift. Repositories generated by Gemini AI Studio demonstrated the risks of high-speed, open-loop generation. In 20% of failures, the model hallucinated the user's high-level intent (e.g., generating a generic Grid Puzzle instead of the requested Roguelike). In another 20%, we observed "Interface Drift," where early-defined modules and late-generated modules used mismatched method signatures. This suggests that without an explicit Contract, the model's internal attention mechanism suffers from decay over long generation sequences.

Lingma: Logical Detachment. Lingma's output was characterized by subtle integration bugs that hint at context loss. Specifically, modules frequently lacked necessary error handling, leading to silent runtime crashes. Furthermore, we observed recurring violations of Object-Oriented principles, such as classes failing to implement abstract methods defined in base classes.

Trae: The "Orphaned Logic" Paradox. While Trae generated the most visually complex code structures, it suffered from "Orphaned Logic." Inspection revealed sophisticated `StateManager` classes that were fully implemented but never instantiated or invoked within the main `GameEngine` loop. Similarly, UI rendering loops often failed to poll the game logic state, resulting in a "frozen" interactive window despite correct rendering code. This indicates a failure in synthesizing the holistic system flow, where individual components are high-quality but their orchestration is disconnected.

E Appendix: Human Evaluation Protocol and Metrics

To ensure the rigor of our "greenfield" task evaluations, we established a standardized triple-blind human scoring protocol. Each repository was evaluated by three independent reviewers based on the following operational criteria:

1. Executability (E_x): Measured as a binary pass/fail. A repository is marked as *pass* if it satisfies: (a) Zero *ModuleNotFoundError* after standard pip installs; (b) Successful execution of *main.py* without runtime Traceback for at least 60 seconds of idle time.

2. Interactivity (I_n): Evaluates the feedback loop between the system and human input. Reviewers followed a fixed action sequence:

- **Gomoku:** Placing a stone on the grid results in a state change and triggers an AI response.
- **Plane Battle:** Directional keys accurately move the sprite; the "Fire" key instantiates bullet objects.

- **City Sim:** Mouse interaction with the UI menu allows selecting building types; clicking the grid successfully places structures and updates the visual HUD.
- **Snake++:** Directional inputs change the snake's heading instantly; activating "Speed Boost" results in a visible acceleration without input lag.
- **Roguelike:** The character moves between rooms and interacts with enemy items (e.g., orc).

3. Rule Adherence (R_a): Checks the logical integrity of game-specific mechanics:

- **Gomoku:** The system correctly identifies 5-in-a-row as a terminal winning state.
- **Plane Battle:** Collision between the player and projectiles triggers health reduction or game-over logic.
- **City Sim:** The resource dependency loop is enforced: Houses generate tax revenue if and only if sufficient Energy exists; Power Plants correctly deplete Money to build.
- **Snake++:** Modular power-up logic functions as defined: "Shields" prevent game-over on wall collision, and "Teleportation Walls" correctly wrap entity coordinates.
- **Roguelike:** Map generation adheres to procedural constraints (e.g., rooms are traversable).

F Appendix: The Greenfield-5 Benchmark Specification

We introduce **Greenfield-5**, a benchmark suite designed to stress-test autonomous agents on "Vibe Coding" scenarios. The suite is open-sourced at `https://anonymous.4open.science/r/ContractCoding`.

F.1 Task Definitions

Each task is defined by a high-level, ambiguous user prompt (The "Vibe") and a set of functional requirements.

- **Gomoku (Logic-Intensive):** Write a Gomoku program with AI that allows players to play against AI.
- **Plane Battle (Action-Oriented):** Build a flying battle game where airplanes can move up, down, left, and right, and also fire bullets. Bullets can hit enemies, and enemies will die if hit. Players need to control the airplane to avoid enemy bullets until all enemies are eliminated or the player is hit and killed.
- **City Sim (Resource Management):** Project: Micro City Sim Objective: Create a relaxing grid-based city builder in Python/Pygame focused on resource balancing. Core Features Needed: The Economy Loop: I need a simulation where Money and Energy are calculated in real-time. Houses generate Money (Tax) but consume Energy. Power Plants generate Energy but cost Money to build. If Energy is low, Houses stop paying taxes. (This logic needs to be robust). Building System: I want to select different structures from a menu and click the grid to place them. Include at least: Roads, Residential Zones, and Industrial Zones. Make it easy to add more building types later (Implicitly asks for polymorphism). Visual Feedback: A HUD that shows my current resources and alerts me if I'm out of power. The map should look clean. Requirements: Above 10 files.
- **Snake++ (Event-Driven):** Project: Snake Grand-Master Objective: Build a highly extensible Snake game using Python and Pygame. Functional Requirements: Advanced Movement: Standard snake growth logic with support for "Speed Boost" and "Teleportation Walls". Power-up Factory: A modular system to spawn different items (Apple for growth, Magnet for distance, Shield for wall-clip). Level/Map System: Different arena layouts (e.g., Box, Tunnel, Maze). Leaderboard & Save System: A module to handle high-score persistence and rank calculation. Architecture Constraint: The repository must contain 13-15 files.
- **Roguelike (System Architecture):** Project: Abyssal Echoes (Roguelike) Objective: Generate a repository-level Python project using Pygame. The architecture must be highly modular to support the following complex features: Procedural Map System: An isolated algorithm module generating rooms and tunnels. Data must be separated from rendering.

Event-Driven UI: A Message Log and HUD that updates via an Event Bus, never by direct coupling to game logic. Polymorphic Entities: Implement Player, Orc (Melee), and Mage (Ranged/Fleeing AI) sharing a base class. Interaction System: An inventory system where items can trigger effects across different modules (e.g., a Scroll that reveals the Fog of War map layer). Constraint: The codebase must be split into at least 15 files.

F.2 Evaluation Protocol and Metrics

To ensure rigorous comparison, we employ a dual-layered evaluation harness combining static structural analysis and dynamic functional execution.

1. Static Structural Analysis. Before execution, we verify the repository's topological health to quantify the alleviation of "Architectural Collapse."

- **Architectural Fidelity ($S_{arch} \in [0, 1]$):** Measures the structural alignment with the ground-truth reference architecture. Let F_{gen} and F_{ref} be the sets of file paths in the generated and reference repositories, respectively. We utilize the F1-score of the file set match:

$$S_{arch} = \frac{2 \cdot |F_{gen} \cap F_{ref}|}{|F_{gen}| + |F_{ref}|} \tag{9}$$

  This metric penalizes both "Hollow Skeleton" (missing functionality) and "Bloated" (redundant function files) hallucinations.

- **Linkage Consistency ($S_{link} \in [0, 1]$):** Evaluates the validity of inter-file symbol resolution. We parse the Abstract Syntax Tree (AST) of all Python files to extract the set of internal import statements $\mathcal{I}$. An import $i \in \mathcal{I}$ (e.g., `from A import B`) is valid if and only if file `A` exists and defines symbol `B`.

$$S_{link} = \frac{\sum_{i \in \mathcal{I}} \mathbb{I}(i \text{ is valid})}{|\mathcal{I}|} \tag{10}$$

  High S_{link} indicates a rigorously decoupled and connected dependency graph.

2. Dynamic Functional Metrics. For repositories that pass basic syntax checks, we calculate the **Overall Success Score** ($S_{overall}$) as the arithmetic mean of three execution sub-metrics:

$$S_{overall} = \frac{1}{3}(S_{exec} + S_{inter} + S_{rule}) \times 100\% \tag{11}$$

- **Executability ($S_{exec} \in \{0, 1\}$):** The repository must install dependencies (via `pip`) and launch the entry point without runtime crashes (e.g., `Traceback`, `ModuleNotFoundError`) for a 60-second keep-alive window.

- **Interactivity ($S_{inter} \in [0, 1]$):** Measured via a standardized UI action sequence (e.g., "Click Start" $\rightarrow$ "Move Player"). The system scores points for valid state transitions/responses to automated user inputs.

- **Rule Adherence ($S_{rule} \in [0, 1]$):** Deep verification of game-specific logic (e.g., "Health decreases exactly by 10 upon collision"). This detects subtle "Zombie Code" that runs but fails to implement intent.

System Prompt

```
You are an expert agent within a larger, collaborative multi-agent system. Your primary goal is to
contribute to the overall task by performing your specific role and then passing control to the next
appropriate agent(s).

# Important Instruct Reminders

1. Do what has been asked; nothing more, nothing less.
2. NEVER create files unless they're absolutely necessary for achieving your goal. This means NO
documentation (like README.md), configuration, or test files unless you are explicitly told to create
them.
3. ALWAYS prefer editing an existing file to creating a new one.
4. Use OpenAI function calling to execute tools.

# Collaboration Guideline

1. **Collaboration is Key**: All agents work together to achieve the project's goals.
2. **Document Management**: You have access to a shared "Collaborative Document". All agents in this
workflow can read and write to.
   It is the central place for sharing knowledge, plans, API definitions, file contents, or any other
    IMPORTANT information.
3. **Document Conciseness**: The content of Collaborative Document should be as concise as possible,
providing key information or API interfaces.
4. **Context Management**: Keep only necessary information in the document. Remove outdated specifica-
tions, redundant details, and verbose descriptions.
5. **API Minimalism**: API descriptions should include only: endpoint path, method, required parameters
, and response format. Omit lengthy explanations.

## Document Structure
The Collaborative Document MUST contain:

1) Requirements Document (Problem-space, contract-agnostic)

2) Technical Document (Solution-space, file-led)
    - Sub-Tasks(File-based)

# DOCUMENT ACTION LANGUAGE GUIDELINE
The `<document_action>` tag contains a JSON array of action objects. All agents share the SAME
`Collaborative Document`. `content` is a MARKDOWN string representing the FULL document.
**`update|add`**: Updates or Adds content to the Collaborative Document.
- Section-patch mode (preferred): `content` is a JSON object where keys are section names and values are
MARKDOWN section bodies (NO section heading lines).
  - Allowed section keys:
    - "Project Overview"
    - "User Stories (Features)"
    - "Constraints"
    - "Directory Structure"
    - "Global Shared Knowledge"
    - "Dependency Relationships"
    - "Symbolic API Specifications"
 - The system will apply your section patches onto the current document and keep the overall document in
  the required MD template format.
 - IMPORTANT: A section patch is a FULL REPLACEMENT of that section's body. If you change only part of a
  section, you MUST still output the entire final section body, including unchanged lines.
 - WARNING: Partial section patches (e.g., only "* **Status:** DONE" without any "**File:** ...") may
  be rejected to prevent clobbering the section.
- Example: `[{"type": "update|add", "section": "key", "content": "...markdown to append..."}]`

# INSTRUCTIONS: Your response MUST follow this structure EXACTLY, this is VERY IMPORTANT.
1.  **Thinking Process**: In a `<thinking>` block, provide a step-by-step analysis of the current situ-
ation, your reasoning, and your plan.
2.  **Output**: In an `<output>` block, provide your primary output. A human-readable text summary of
your work, analysis, or conclusion.
3.  **Document Actions (Optional)**: If you need to modify the shared document, provide a
`<document_action>` block containing a valid JSON array of action objects. If you don't need to modify
the document, omit this entire block.
```

Contract Prompt

```
You are the Project Manager responsible for producing a thorough, correct, and task-driven implementation plan. Your plan must be contract-first, dynamically structured (no pre-committed stack), parallelizable, and observable. It must apply across different programming tasks and domains.

### Role Principles
- Dynamic structure: choose only the minimal architecture and tools needed for the current task; don’t pre-commit to a fixed stack.
- Prioritize correctness: Ensure the rationality of task coordination after your decomposition
- Contract-first: define interfaces and algorithm contracts (paths/methods/data shapes/types).
- Parallelization & simplicity: decompose into concurrent modules while minimizing complexity.
- Focus only on essential implementation tasks: Each task should produce concrete deliverables.
- Read the Collaborative Document and the user task. Produce a plan that is implementable, explicitly scoped, and adaptable.
- Include only what is necessary for the current task; every choice (tech/structure) must be justified by necessity.

###Task decomposition
- **Keep the workflow streamlined.**
- **Maintain clarity and logical flow.** Decomposition should be intuitive, avoiding redundant or tedious steps.
- **Prioritize core functions.** Focus on the main features required, rather than edge situations or advanced features.
- **Decomposition principle.** The decomposition of subtasks can be based on the files and their implemented functions.

### CRITICAL INSTRUCTION: Document Creation
- You MUST use the `document_action` tool with type `add` to CREATE the initial Collaborative Document.
- The content of the document MUST be placed INSIDE the JSON list in <document_action>.
- The document MUST follow the template below EXACTLY.

**Collaboration Document Template:**
{Contract Template}
```