When Parallelism Pays Off: Cohesion-Aware Task Partitioning for Multi-Agent Coding

Xu Yang[*1], **Lunyiu Nie**[*1]
Ethan Chandra[1], **Stanislav Gannutin**[1], **Fangru Lin**[2], **Swarat Chaudhuri**[1]

[1]The University of Texas at Austin [2]University of Oxford

Abstract

LLM-based multi-agent systems decompose complex tasks such as coding through parallelization and context isolation, yet inter-agent communication overhead can offset the efficiency gains. We formalize multi-agent orchestration as a graph partitioning problem that captures the *communication-to-computation trade-off*: task decomposition shortens critical-path computation, but cross-agent dependencies require costly context transfer. We instantiate this view in repository-level software engineering with Cohesion-aware Coder (Co-Coder), which builds dependency graphs from code architecture design, isolates structural hub files, partitions the graph via community detection, and executes the partition with a dependency-aware scheduler. Across 28 real-world tasks on DevEval and CodeProjectEval with two base models, Co-Coder advances the Pareto frontier over all baselines, with up to a $3.13\times$ speedup and a 42% cost reduction over the sequential version. Against Claude Code with Agent Teams, it raises pass rate by 9.8% to 17.8% while running up to $1.41\times$ faster at up to 22% lower cost.

1 Introduction

Large Language Model (LLM) agents are increasingly deployed not as monolithic solvers but as *multi-agent systems* that decompose a task across several specialized workers running in parallel [7, 23, 31, 35, 24]. The appeal of this paradigm is straightforward: more agents should mean more throughput, more structured context management, and shorter wall-clock latency. In practice, however, these benefits do not always hold: recent findings show that adding agents can degrade both speed and task performance compared to a single capable agent [2, 20, 13, 22].

We argue that this behavior is not an idiosyncrasy of LLMs but a familiar one, and that it is best understood through the lens of the *communication-to-computation trade-off* that has organized four decades of distributed computing research [28, 3, 6, 12, 11, 27]. In any distributed parallel system, total runtime is shaped by two competing terms: (i) the useful work each worker performs in isolation, and (ii) the coordination cost paid whenever one worker depends on another. Multi-agent

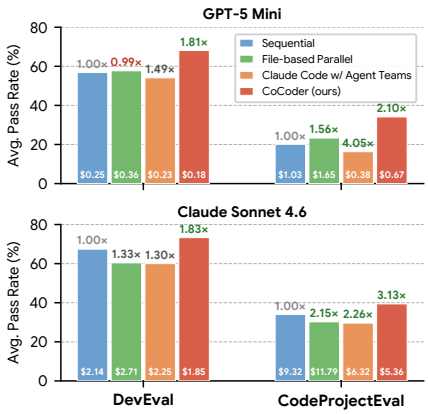


Figure 1: Average pass rate with GPT-5 Mini and Claude Sonnet 4.6. Above bars: wall-clock speedup vs. Sequential; inside bars: avg. API cost per task (USD).

[*] Equal Contribution. We release our code at `https://github.com/Flitternie/CoCoder`.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

systems are no exception: every cross-agent dependency
must be resolved by transmitting context tokens, which inflate prompts, decoding time, and inference cost. While a growing body of work studies multi-agent communication topologies [35, 24] and parallel function calling [14], the trade-off itself has not been formalized as an objective that an orchestrator can be evaluated against, and the structural problem of *how tasks are partitioned across agents* in the first place has received little attention.

We make this perspective concrete in the setting of repository-level software engineering, a canonical testbed in which task dependencies are unusually legible: files import from one another, call graphs are recoverable by static analysis, and the cost of cross-agent context transfer can be measured directly. The bottleneck of current multi-agent coding systems [7, 23, 1] is precisely the trade-off above: when heavily interdependent files are assigned to different agents, those agents must repeatedly halt generation to query peers or ingest updated repository state, and wall-clock latency grows with the agent count rather than shrinking. The symptoms are visible in Figure 1: on CodeProjectEval, a file-based parallel system inflates API cost by 60% for the $1.56\times$ efficiency gain; on two benchmarks, Claude Code with Agent Teams achieves the fastest wall-clock time but falls below the sequential version in terms of code quality. Both are failure modes of poorly-orchestrated parallelism: one wastes cost, the other sacrifices code quality.

Inspired by classic distributed computing systems, we propose **Co**hesion-aware **Coder** (Co-Coder), an orchestration framework that uses directed acyclic graphs to partition and schedule multi-agent systems. Co-Coder represents the project as a weighted dependency graph in which vertex weights encode per-file generation cost and edge weights encode the cost of transferring context across an agent boundary, and casts orchestration as a graph partitioning problem whose objective combines critical-path computation cost with cross-partition communication cost, the same two terms studied in classical distributed computing. Because code dependency graphs are directed, heterogeneous, and dominated by a few hub files (project-wide utilities and top-level entry points), generic spectral methods are a poor fit; Co-Coder instead isolates these hubs, then applies the Infomap algorithm [25], which minimizes the description length of a random walk on the directed graph and thereby acts as a surrogate for cross-partition information flow. A final post-processing step recovers latent intra-cluster parallelism, and the resulting partition is executed by a dependency-aware list scheduler [6] that exposes parallelism without global synchronization barriers.

Across 28 real-world projects on two repository-level coding benchmarks, DevEval [15] and CodeProjectEval [34], and two base models, GPT-5 Mini and Claude Sonnet 4.6, Co-Coder advances the Pareto frontier across pass rate, latency, and cost. It consistently improves pass rate over the sequential version while delivering up to a $3.13\times$ speedup and a 42% cost reduction. Compared with Claude Code with Agent Teams, a strong industrial baseline, Co-Coder achieves a higher pass rate in every setting while running up to $1.41\times$ faster and at up to 22% lower cost. The gains are largest on projects with the densest cross-file dependencies, where baseline systems including Claude Code with Agent Teams either inflate cost without improving quality or sacrifice quality for speed. This confirms that the communication-to-computation trade-off, rather than raw concurrency, is what governs multi-agent system behavior.

In summary, our contributions are:

- We highlight the communication-to-computation trade-offs in multi-agent systems and formulate an objective that combines critical-path computation cost with inter-agent communication cost.
- We propose Co-Coder, an orchestrator that instantiates this objective via cohesion-based graph partitioning and dependency-aware scheduling on repository-level multi-agent coding tasks.
- We show empirically that Co-Coder advances the Pareto-frontier over sequential, file-parallel, and Claude Code baselines, with the largest gains on the most dependency-dense projects.

2 Problem Formulation

We formalize a type of systems in which K parallel LLM agents jointly complete a composite task. The task decomposes into a set of subtasks with dependencies among them: some subtasks can only begin once others have been completed, and agents that work on dependent subtasks must share context about one another's outputs. The system's end-to-end overheads are therefore shaped by two competing factors: the time agents spend doing productive work in isolation, and the

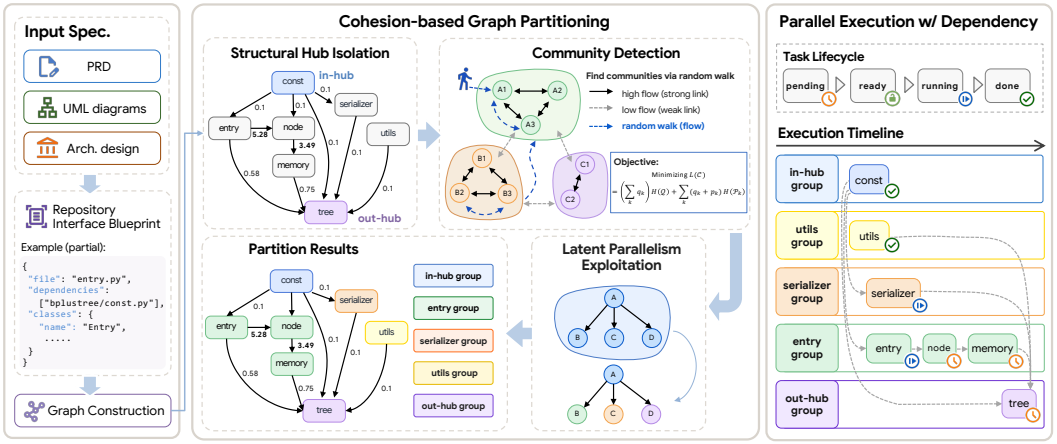


Figure 2: Overview of Co-Coder. The input specifications from the benchmarks are first parsed into a Repository Interface Blueprint that encodes file-level dependencies, which subsequently constructs the weighted graph. We then perform cohesion-based partitioning over the graph via structural hub isolation, community detection, and latent parallelism exploitation. Finally, on the right side, parallel coding agents execute in a dependency-aware timeline, where multiple tasks across different groups follow a lifecycle and may fire in parallel once their upstream dependencies are completed. Note that dependency arrows are reversed for illustration: arrows point from dependency to dependent, opposite to the actual dependency graph convention.

communication overhead they incur when resolving cross-agent dependencies. We formalize this tension by representing the task structure as a weighted dependency graph and casting orchestration as the problem of partitioning that graph to minimize a joint cost over both terms.

Task Structure as a Weighted Dependency Graph. We represent the composite task as a directed graph $G = (V, E)$. Each vertex $v_i \in V$ corresponds to one elementary subtask assigned to a single agent. A directed edge $(v_i, v_j) \in E$ indicates that v_i depends on v_j: the agent handling v_i must observe some output of v_j before it can complete its own work. Each vertex carries a non-negative weight w_i quantifying the *computation cost* of completing v_i, measured as the expected wall-clock time an agent spends working on it in isolation. Each edge carries a non-negative coupling weight c_{ij} quantifying the *communication cost* incurred when v_i and v_j are handled by different agents: a heavier c_{ij} means the two subtasks share more output interface and would require more context transfer to coordinate across an agent boundary. The specific instantiation of V, E, w_i, and c_{ij} for repository-level coding is described in Section 3.1.

Partitioning as Agent Assignment. Orchestrating K agents corresponds to partitioning V into K disjoint groups $P = \{P_1, \ldots, P_K\}$, where all subtasks in P_k are handled by the same agent. The partition P induces a scheduled execution DAG $G_P = (V, E \cup E_P)$, where E_P contains the intra-partition serialization edges that arise because each agent processes its subtasks one at a time. The system-wide *computation cost* is the critical-path latency of this DAG,

$$W(P) \;=\; \max_{\pi \,\in\, \Pi(G_P)} \;\sum_{v_i \,\in\, \pi} w_i, \tag{1}$$

where $\Pi(G_P)$ is the set of directed paths in G_P. $W(P)$ captures two sources of serialization: the sequential workload within each agent's group, and cross-partition dependency chains that force a downstream agent to wait for an upstream subtask owned by a different agent. The system-wide *communication cost* is the total coupling weight across partition boundaries,

$$C(P) \;=\; \sum_{\substack{(v_i, v_j) \in E \\ k(v_i) \neq k(v_j)}} c_{ij}, \tag{2}$$

where $k(v_i)$ denotes the partition index of v_i. Coarser partitions internalize dependencies and reduce $C(P)$ but lengthen intra-agent chains and inflate $W(P)$; finer partitions expose parallelism and reduce $W(P)$ but push more dependencies across agent boundaries and inflate $C(P)$.

Optimization Objective. We seek the partition that minimizes the joint cost of both terms,

$$T(P) = W(P) + \alpha \cdot C(P), \qquad P^* = \arg\min_P T(P), \tag{3}$$

where $\alpha > 0$ represents a trade-off weighting factor and $K = |P|$ is determined by the partition itself. This objective is the direct analogue of the communication-to-computation cost studied in classical distributed task scheduling [27, 28]: $W(P)$ plays the role of the critical-path computation time, and $\alpha C(P)$ plays the role of inter-processor communication overhead. Section 3 introduces Co-Coder, which targets $T(P)$ with a cohesion-based heuristic rather than solving Eq. (3) exactly, in the setting of repository-level software engineering.

3 The Co-Coder Framework

The objective $T(P)$ in Section 2 is defined over a weighted dependency graph that is not given directly — it must be inferred from the project's requirements before any partitioning can take place. Once the graph is available, the partition must balance two competing demands: grouping tightly coupled files to minimize inter-agent communication, and avoiding serialization bottlenecks to keep wall-clock time short. Finally, the partition must be realized by a scheduler that respects cross-partition dependencies without introducing unnecessary synchronization barriers. Co-Coder addresses each concern in sequence through three stages: *graph construction* (Section 3.1), *cohesion-based partitioning* (Section 3.2), and *dependency-aware execution* (Section 3.3).

3.1 Graph Construction

Repository Interface Blueprint. The graph G is not directly observable from the input, which consists of free-form requirement documents (e.g., product specifications, UML class and sequence diagrams). To bridge this gap, we prompt an LLM to produce a *Repository Interface Blueprint* (RIB), a structured outline of the project at file granularity. For each file f_i, the RIB enumerates the symbols it defines (class names, function signatures with typed parameters, and global constants) and the set of files it imports from. The symbol inventory seeds the per-vertex computation weight w_i from interface size, which serves as a proxy for generation latency (validated against measured generation time in Appendix E). The import graph yields the edge set E. The RIB is generated by an LLM agent that reads the requirement documents with file-access tools, and it is refined through a lightweight self-critique loop (Appendix A).

Edge Weights from Symbol Sharing. The RIB tells us which files depend on which, but not how strongly. We estimate the coupling weight c_{ij} from the symbols the two files share: it should be large when one file defines symbols the other must consume, and small when both files merely co-reference symbols defined elsewhere. We capture this asymmetry by weighting each symbol higher in a file's feature vector when the file defines it than when it only references it, after excluding builtin types and boilerplate methods. For every edge $(v_i, v_j) \in E$ we set

$$c_{ij} = \gamma \cdot \frac{\mathbf{s}_i \cdot \mathbf{s}_j}{\|\mathbf{s}_i\| \, \|\mathbf{s}_j\|}, \tag{4}$$

where $\mathbf{s}_i$ is the sparse symbol vector for v_i and $\gamma > 0$ calibrates the weights into the same units as α in Eq. (3). Definition–use edges receive a high$\times$low inner product and thus a large c_{ij}; edges of incidental co-occurrence receive a low$\times$low product and a small c_{ij}.

3.2 Cohesion-based Graph Partitioning

With the weighted dependency graph G in hand, the next stage is to partition G in a way that jointly addresses the two cost terms in $T(P) = W(P) + \alpha\, C(P)$ (Section 2): the critical-path latency $W(P)$ taken over the scheduled DAG $G_P = (V,\, E \cup E_P)$ that augments E with intra-agent serialization edges E_P, and the cross-partition communication cost $C(P) = \sum_{(v_i,v_j)\in E,\, k(v_i)\neq k(v_j)} c_{ij}$. Both terms demand *cohesion*: the degree to which files within a group are more tightly coupled to each other than to files outside it. High cohesion keeps the costliest dependencies inside group boundaries (suppressing inter-agent communication) while limiting the length of intra-group serialization chains (keeping execution on the critical path short). Co-Coder produces high-cohesion partitions through

a three-step pipeline summarized in Algorithm 1: (i) isolate structural hubs that obscure modular structure (lines 1–2); (ii) cluster the remaining graph into cohesive communities (lines 3–4); (iii) lift independent files out of their clusters to expose latent parallelism (lines 5–8).

Isolating Structural Hubs. Real code dependency graphs are highly heterogeneous: a small number of files act as project-wide utilities or top-level entry points [21], and these have a disproportionate effect on $T(P)$. Vertices v_i with high in-degree are widely depended-upon utilities (constants, shared schemas, exception hierarchies), which we call *in-hubs*: their dependents are spread across most partitions, so the cut edges incident to an in-hub are nearly invariant to its assignment, while merging it into any group P_k inserts it into that agent's serialization chain and inflates $W(P_k)$ by w_i. Singleton-isolating each in-hub therefore leaves $C(P)$ essentially unchanged but lifts v_i off every group's critical path. Vertices with high out-degree are top-level aggregators and controllers, which we call *out-hubs*: placed in any group P_k, an out-hub v_i forces all of its in-group upstreams into a chain terminating at v_i, inflating $W(P_k)$ in proportion to the number of upstreams pulled in. Grouping all out-hubs into a single integration group concentrates this late-stage serialization at one agent without polluting the critical paths of the cohesive groups. Both hub types also bias generic clustering by collapsing the graph into one large community; isolating them first allows the remaining vertices to be clustered cleanly. The subgraph G_C induced by the remaining non-hub vertices is passed to the next step.

Community Detection. With hubs removed, G_C has a more uniform structure amenable to community detection. We cluster G_C into cohesive groups using Infomap [25], a community-detection algorithm that operates natively on directed weighted graphs and infers the number of clusters from the data. Let E_C be the edge set of G_C. Infomap models G_C as a random walk whose transition probabilities follow the coupling-weighted topology:

$$p(v_j \mid v_i) \;=\; \frac{c_{ji}}{\sum_{(v_l,\, v_i) \,\in\, E_C} c_{li}}, \tag{5}$$

where the walk moves from each file to the files that depend on it, so the walk tends to remain within strongly coupled modules and infrequently traverses low-weight cross-module edges. For a candidate partition $\mathcal{C} = \{C_1, \ldots, C_m\}$, let $q_k = \sum_{v_i \in C_k,\, v_j \notin C_k} \pi_i\, p(v_j \mid v_i)$ be the exit rate of cluster C_k and $p_k = \sum_{v_i \in C_k} \pi_i$ its stationary residence probability, where π_i is the stationary visit probability of v_i. Infomap selects $\mathcal{C}$ by minimizing the two-level description length,

$$L(\mathcal{C}) \;=\; \Big(\sum_k q_k\Big) H(\mathcal{Q}) \;+\; \sum_k (q_k + p_k)\, H(\mathcal{P}_k), \tag{6}$$

where $H(\mathcal{Q})$ is the entropy of the inter-cluster exit distribution and $H(\mathcal{P}_k)$ is the entropy of module C_k's codebook over its internal node visits and the exit event. Intuitively, a high $H(\mathcal{Q})$ indicates frequent cross-cluster transitions (low cohesion), while a high $H(\mathcal{P}_k)$ indicates that an agent must process many weakly related files. Minimizing $L(\mathcal{C})$ therefore steers the partition away from cutting heavily weighted edges, effectively suppressing $C(P)$. The granularity of the partition is controlled by the Markov time t: at larger t the random walk traverses more edges per step, making it easier to escape small modules and raising their q_k, so Infomap merges them into fewer, larger clusters. This reduces communication cost $C(P)$ at the expense of longer intra-group serialization chains (higher $W(P)$), providing a knob for navigating the computation–communication trade-off in Eq. (3).

Latent Parallelism Exploitation. Infomap produces cohesive clusters that suppress $C(P)$, but cohesion alone does not address $W(P)$: within a single cluster, vertices that share an upstream dependency but have no dependency on each other are still processed serially, unnecessarily stretching the critical path. We recover this latent parallelism by lifting any vertex v_i that (i) has no intra-cluster dependents and (ii) shares each of its intra-cluster upstreams with at least one cluster sibling. Lifting only v_i to a singleton converts the edge (v_i, v_j) into a cut at cost $\alpha\, c_{ij}$, but removes w_i from the cluster's serialization chain and shortens the critical path of G_P by the same amount; condition (ii) ensures each upstream v_j retains an intra-cluster dependent, so no other cluster edge is disturbed. Lifting therefore can reduce $T(P)$ when $\alpha\, c_{ij} < w_i$, and conditions (i) and (ii) act as a structural proxy for this trade-off, identifying leaf vertices whose coupling to their upstreams is typically outweighed by their generation cost w_i. Note that repositories without structural hubs may not require hub isolation, and those without such sibling leaves may not require lifting.

Algorithm 1: Cohesion-based Graph Partitioning

Input: Weighted directed graph $G = (V, E, \{w_i\}, \{c_{ij}\})$ constructed from Section 3.1
Output: Partition $P = \{P_1, \ldots, P_K\}$ of V, one group per agent

$V_{\text{in}} \leftarrow$ high in-degree vertices (*in-hubs*); $V_{\text{out}} \leftarrow$ high out-degree vertices (*out-hubs*);
$P \leftarrow \{\, \{v\} : v \in V_{\text{in}} \,\} \cup \{V_{\text{out}}\}$ `// isolate hubs off critical paths`
$G_C \leftarrow G[\, V \setminus (V_{\text{in}} \cup V_{\text{out}}) \,]$; $\{C_1, \ldots, C_m\} \leftarrow \text{Infomap}(G_C)$ `// partition by Eq.` (6)
$P \leftarrow P \cup \{C_1, \ldots, C_m\}$
foreach *cluster* C_k **do**
  $S_k \leftarrow \{\, v_i \in C_k$: (i) no vertex in C_k depends on v_i; (ii) each intra-cluster upstream of v_i has another dependent in $C_k \,\}$ `// leaves liftable to singletons`
  $C_k \leftarrow C_k \setminus S_k$; $P \leftarrow P \cup \{\{v_i\} : v_i \in S_k\}$ `// lift to exploit parallelism`
end
return P

3.3 Dependency-Aware Parallel Execution

The partition P assigns files to agents, but cross-partition edges still impose ordering constraints: an agent cannot faithfully generate a downstream file before its upstream dependencies are available. Naively gating execution on the topological depth of the dependency DAG forces every agent to wait at the slowest layer boundary and erodes the parallelism that the partition was designed to expose. Co-Coder instead executes the partition through a *shared task list*, in which each file is registered as a task whose state advances from *pending* to *ready* the moment all of its upstream files have completed, and is then picked up by the agent that owns its group. There are no global synchronization barriers between dependency layers, and faster agents immediately move on to the next ready task. This realizes the standard greedy list-scheduling policy [6] on the partitioned graph and approaches the critical-path latency $W(P)$ defined in Eq. (1).

After all files are generated, the leader agent compiles the project and runs the visible test suite that ships with the task, which is kept separate from the held-out tests used for evaluation. It reads the failure report and repairs the responsible files directly. Because only the leader edits files during repair, concurrent conflicting edits across agents are avoided. In the repair loop, the agent re-runs the visible tests after each round and stops as soon as all of them pass, iterating for up to a fixed number of rounds, as listed in Table 8.

4 Experimental Setups

4.1 Benchmarks

We evaluate on two repository-level code generation benchmarks. **DevEval** [15] consists of compact, self-contained repositories for end-to-end agent pipelines; we use its Python subset of 10 projects, whose ground-truth reference implementations average 3.1 files and 243 LOC. **CodeProjectEval** [34] is curated from real-world open-source libraries and contributes 18 projects, with reference implementations averaging 11.9 files and 2,371 LOC per project, targeting deep cross-file dependencies and large integration surfaces. Both benchmarks provide unit tests for automated evaluation. In both, the task is to generate the full repository from scratch: the requirement documents (product specification, UML diagrams, and architecture design, including the file structure) are the task input, and every method receives the same documents.

4.2 Metrics

We report three per-repository metrics: (i) *pass rate*, the fraction of held-out unit tests passed; (ii) *wall-clock latency* in seconds, measured from pipeline start to the completion of the final agent, covering all stages including blueprint generation and partitioning but excluding evaluation on the held-out tests; and (iii) *API cost* in USD, summed across all involved agents' input and output token usage. Each repository is evaluated over three independent runs and we report mean values.

Table 1: Per-project results on DevEval using GPT-5 Mini. *#LOC* represents lines of code in the ground truth implementation; *#Tests* is the total number of unit tests; *#Pass* reports the number of unit tests passed; *Time* represents wall-clock latency in seconds; *Cost* is total token cost in USD. Values denote mean$_{\pm \text{sd}}$ over three runs. Best #Pass per row in **bold**; the Avg row reports per-task mean pass rate (%); the $\times$ values are per-task means of Time and Cost normalized to Sequential (lower is better), with green indicating better than Sequential and red indicating worse.

Task	#LOC	#Tests	Sequential			File-based Parallel			Claude Code w/ Agent Teams			**Co-Coder (ours)**		
			#Pass	Time	Cost	#Pass	Time	Cost	#Pass	Time	Cost	#Pass	Time	Cost
ArXiv_digest	198	38	30.0$_{\pm 0.0}$	449$_{\pm 44}$	0.13$_{\pm 0.02}$	29.7$_{\pm 0.5}$	782$_{\pm 116}$	0.23$_{\pm 0.06}$	20.0$_{\pm 14.1}$	573$_{\pm 560}$	0.21$_{\pm 0.17}$	**31.3**$_{\pm 0.5}$	475$_{\pm 79}$	0.11$_{\pm 0.01}$
chakin	62	1	**1.0**$_{\pm 0.0}$	394$_{\pm 45}$	0.15$_{\pm 0.02}$	**1.0**$_{\pm 0.0}$	456$_{\pm 220}$	0.16$_{\pm 0.01}$	0.7$_{\pm 0.5}$	150$_{\pm 84}$	0.09$_{\pm 0.04}$	**1.0**$_{\pm 0.0}$	319$_{\pm 56}$	0.11$_{\pm 0.02}$
geotext	134	4	2.0$_{\pm 0.8}$	740$_{\pm 213}$	0.37$_{\pm 0.21}$	2.0$_{\pm 0.0}$	779$_{\pm 130}$	0.30$_{\pm 0.06}$	1.7$_{\pm 1.2}$	587$_{\pm 436}$	0.48$_{\pm 0.42}$	**2.7**$_{\pm 0.5}$	691$_{\pm 290}$	0.29$_{\pm 0.14}$
hone	274	7	2.3$_{\pm 1.2}$	994$_{\pm 83}$	0.37$_{\pm 0.08}$	2.0$_{\pm 0.0}$	556$_{\pm 52}$	0.43$_{\pm 0.08}$	4.0$_{\pm 1.6}$	218$_{\pm 95}$	0.15$_{\pm 0.06}$	**6.0**$_{\pm 0.0}$	441$_{\pm 32}$	0.24$_{\pm 0.02}$
Hybrid_Images	144	19	**17.0**$_{\pm 0.0}$	296$_{\pm 56}$	0.05$_{\pm 0.01}$	**17.0**$_{\pm 0.0}$	210$_{\pm 50}$	0.07$_{\pm 0.00}$	**17.0**$_{\pm 0.0}$	798$_{\pm 51}$	0.25$_{\pm 0.06}$	**17.0**$_{\pm 0.0}$	179$_{\pm 45}$	0.05$_{\pm 0.01}$
lice	376	25	8.0$_{\pm 0.0}$	514$_{\pm 87}$	0.16$_{\pm 0.02}$	9.0$_{\pm 1.4}$	520$_{\pm 101}$	0.18$_{\pm 0.05}$	8.7$_{\pm 0.9}$	138$_{\pm 52}$	0.10$_{\pm 0.05}$	**9.7**$_{\pm 1.7}$	325$_{\pm 95}$	0.10$_{\pm 0.03}$
pso	168	5	**5.0**$_{\pm 0.0}$	472$_{\pm 85}$	0.13$_{\pm 0.01}$	**5.0**$_{\pm 0.0}$	345$_{\pm 47}$	0.14$_{\pm 0.01}$	**5.0**$_{\pm 0.0}$	581$_{\pm 333}$	0.26$_{\pm 0.12}$	**5.0**$_{\pm 0.0}$	202$_{\pm 11}$	0.08$_{\pm 0.01}$
readtime	284	8	**2.0**$_{\pm 0.0}$	853$_{\pm 133}$	0.23$_{\pm 0.02}$	**2.0**$_{\pm 0.0}$	486$_{\pm 168}$	0.25$_{\pm 0.03}$	1.3$_{\pm 0.9}$	643$_{\pm 366}$	0.13$_{\pm 0.10}$	**2.0**$_{\pm 0.0}$	400$_{\pm 101}$	0.14$_{\pm 0.01}$
stocktrends	384	7	3.0$_{\pm 0.0}$	2257$_{\pm 304}$	0.70$_{\pm 0.05}$	3.0$_{\pm 0.0}$	3412$_{\pm 372}$	1.55$_{\pm 0.12}$	**4.3**$_{\pm 1.9}$	1251$_{\pm 232}$	0.52$_{\pm 0.11}$	3.0$_{\pm 0.0}$	885$_{\pm 68}$	0.50$_{\pm 0.02}$
TextCNN	403	10	1.7$_{\pm 1.2}$	1029$_{\pm 111}$	0.25$_{\pm 0.05}$	2.7$_{\pm 2.0}$	518$_{\pm 187}$	0.32$_{\pm 0.02}$	2.0$_{\pm 1.4}$	424$_{\pm 279}$	0.15$_{\pm 0.09}$	**5.0**$_{\pm 0.0}$	504$_{\pm 148}$	0.21$_{\pm 0.02}$
Avg	–	–	56.8%	800 (1×)	0.25 (1×)	57.7%	806 (1.01×)	0.36 (1.44×)	54.1%	536 (0.67×)	0.23 (0.92×)	**68.1%**	**442** (0.55×)	**0.18** (0.72×)

Table 2: Per-project results on CodeProjectEval using GPT-5 Mini. Metrics and conventions are the same as in Table 1.

Task	#LOC	#Tests	Sequential			File-based Parallel			Claude Code w/ Agent Teams			**Co-Coder (ours)**		
			#Pass	Time	Cost	#Pass	Time	Cost	#Pass	Time	Cost	#Pass	Time	Cost
bplustree	1,509	356	18.0$_{\pm 0.8}$	2192$_{\pm 236}$	0.99$_{\pm 0.11}$	22.0$_{\pm 9.3}$	2312$_{\pm 266}$	1.30$_{\pm 0.11}$	55.7$_{\pm 73.8}$	1055$_{\pm 571}$	0.67$_{\pm 0.55}$	**73.0**$_{\pm 53.7}$	1311$_{\pm 189}$	0.56$_{\pm 0.12}$
cookiecutter	2,805	375	59.0$_{\pm 6.7}$	4260$_{\pm 341}$	1.80$_{\pm 0.14}$	46.7$_{\pm 7.9}$	2180$_{\pm 877}$	2.37$_{\pm 0.18}$	15.7$_{\pm 18.1}$	338$_{\pm 246}$	0.21$_{\pm 0.25}$	**74.7**$_{\pm 1.2}$	1693$_{\pm 314}$	0.87$_{\pm 0.11}$
csvs-to-sqlite	816	25	5.0$_{\pm 7.1}$	1525$_{\pm 183}$	0.46$_{\pm 0.05}$	10.7$_{\pm 7.5}$	1739$_{\pm 265}$	0.68$_{\pm 0.13}$	**16.7**$_{\pm 5.9}$	1746$_{\pm 1184}$	0.51$_{\pm 0.37}$	15.7$_{\pm 0.5}$	1020$_{\pm 195}$	0.25$_{\pm 0.02}$
deprecated	597	176	110.0$_{\pm 1.4}$	639$_{\pm 41}$	0.22$_{\pm 0.03}$	112.3$_{\pm 6.6}$	1225$_{\pm 275}$	0.42$_{\pm 0.06}$	76.7$_{\pm 54.2}$	140$_{\pm 64}$	0.11$_{\pm 0.08}$	**115.7**$_{\pm 5.2}$	752$_{\pm 63}$	0.21$_{\pm 0.03}$
drf-simplejwt	1,712	191	6.7$_{\pm 2.4}$	3633$_{\pm 200}$	1.42$_{\pm 0.11}$	4.7$_{\pm 0.5}$	2022$_{\pm 493}$	1.97$_{\pm 0.21}$	0.0$_{\pm 0.0}$	128$_{\pm 90}$	0.02$_{\pm 0.00}$	**45.3**$_{\pm 25.1}$	1238$_{\pm 198}$	0.82$_{\pm 0.13}$
flask	9,314	482	0.0$_{\pm 0.0}$	7460$_{\pm 499}$	3.04$_{\pm 0.23}$	0.0$_{\pm 0.0}$	2786$_{\pm 223}$	5.30$_{\pm 0.47}$	0.0$_{\pm 0.0}$	762$_{\pm 190}$	0.60$_{\pm 0.52}$	0.0$_{\pm 0.0}$	3173$_{\pm 671}$	2.81$_{\pm 0.81}$
imapclient	3,531	267	46.3$_{\pm 6.8}$	3482$_{\pm 530}$	1.13$_{\pm 0.04}$	49.0$_{\pm 2.2}$	1330$_{\pm 548}$	1.78$_{\pm 0.18}$	3.3$_{\pm 4.7}$	261$_{\pm 100}$	0.20$_{\pm 0.12}$	**63.3**$_{\pm 2.0}$	1104$_{\pm 345}$	0.54$_{\pm 0.08}$
parsel	1,128	250	30.3$_{\pm 9.1}$	1503$_{\pm 75}$	0.55$_{\pm 0.12}$	14.3$_{\pm 10.5}$	1207$_{\pm 566}$	0.56$_{\pm 0.22}$	0.0$_{\pm 0.0}$	444$_{\pm 366}$	0.05$_{\pm 0.01}$	**56.7**$_{\pm 8.2}$	1034$_{\pm 73}$	0.34$_{\pm 0.01}$
portalocker	1,958	71	0.0$_{\pm 0.0}$	1977$_{\pm 34}$	0.71$_{\pm 0.08}$	0.0$_{\pm 0.0}$	1055$_{\pm 353}$	0.86$_{\pm 0.17}$	0.0$_{\pm 0.0}$	1485$_{\pm 1134}$	0.66$_{\pm 0.45}$	0.0$_{\pm 0.0}$	897$_{\pm 74}$	0.35$_{\pm 0.02}$
pyjwt	2,690	294	79.0$_{\pm 47.2}$	2222$_{\pm 275}$	0.79$_{\pm 0.06}$	126.0$_{\pm 15.1}$	1215$_{\pm 203}$	1.48$_{\pm 0.11}$	29.0$_{\pm 41.0}$	156$_{\pm 119}$	0.21$_{\pm 0.25}$	**140.0**$_{\pm 9.1}$	1310$_{\pm 95}$	0.50$_{\pm 0.06}$
python-hl7	2,434	100	16.7$_{\pm 3.7}$	2534$_{\pm 586}$	0.78$_{\pm 0.25}$	33.3$_{\pm 8.0}$	2231$_{\pm 28}$	1.62$_{\pm 0.03}$	20.3$_{\pm 14.7}$	1046$_{\pm 827}$	0.81$_{\pm 0.56}$	**42.3**$_{\pm 6.6}$	1557$_{\pm 249}$	0.58$_{\pm 0.13}$
rsa	2,949	100	35.7$_{\pm 1.2}$	2640$_{\pm 183}$	0.93$_{\pm 0.13}$	43.7$_{\pm 2.9}$	1144$_{\pm 166}$	1.37$_{\pm 0.10}$	13.3$_{\pm 18.9}$	211$_{\pm 142}$	0.17$_{\pm 0.19}$	**54.3**$_{\pm 5.9}$	1331$_{\pm 195}$	0.62$_{\pm 0.06}$
simpy	2,184	149	10.3$_{\pm 7.5}$	3147$_{\pm 267}$	1.42$_{\pm 0.17}$	39.3$_{\pm 3.4}$	3472$_{\pm 469}$	1.89$_{\pm 0.17}$	24.0$_{\pm 33.9}$	1081$_{\pm 721}$	0.69$_{\pm 0.48}$	**58.0**$_{\pm 7.5}$	1710$_{\pm 269}$	0.68$_{\pm 0.09}$
tinydb	2,170	204	110.0$_{\pm 18.6}$	2067$_{\pm 290}$	0.60$_{\pm 0.08}$	61.7$_{\pm 43.6}$	696$_{\pm 12}$	0.85$_{\pm 0.01}$	67.3$_{\pm 95.2}$	301$_{\pm 80}$	0.23$_{\pm 0.14}$	**125.7**$_{\pm 21.4}$	656$_{\pm 123}$	0.30$_{\pm 0.03}$
trailscraper	890	93	14.7$_{\pm 2.5}$	2695$_{\pm 263}$	0.86$_{\pm 0.12}$	17.7$_{\pm 3.4}$	1562$_{\pm 122}$	1.34$_{\pm 0.10}$	7.3$_{\pm 3.1}$	297$_{\pm 139}$	0.30$_{\pm 0.10}$	**31.7**$_{\pm 3.8}$	1237$_{\pm 318}$	0.55$_{\pm 0.06}$
voluptuous	3,100	161	38.7$_{\pm 4.0}$	2624$_{\pm 378}$	1.09$_{\pm 0.13}$	44.0$_{\pm 1.4}$	2400$_{\pm 409}$	1.62$_{\pm 0.19}$	15.3$_{\pm 21.7}$	448$_{\pm 338}$	0.70$_{\pm 0.91}$	**50.3**$_{\pm 1.9}$	1566$_{\pm 54}$	0.62$_{\pm 0.05}$
xmnlp	1,504	23	0.7$_{\pm 0.5}$	3536$_{\pm 964}$	1.12$_{\pm 0.37}$	0.0$_{\pm 0.0}$	2456$_{\pm 282}$	3.37$_{\pm 0.47}$	0.0$_{\pm 0.0}$	2043$_{\pm 1666}$	0.62$_{\pm 0.58}$	**2.7**$_{\pm 0.9}$	1250$_{\pm 342}$	0.96$_{\pm 0.14}$
zxcvbn	1,386	31	13.0$_{\pm 1.4}$	1479$_{\pm 32}$	0.66$_{\pm 0.13}$	14.0$_{\pm 0.8}$	854$_{\pm 60}$	0.87$_{\pm 0.13}$	**16.3**$_{\pm 9.7}$	306$_{\pm 113}$	0.14$_{\pm 0.02}$	**16.3**$_{\pm 0.5}$	823$_{\pm 157}$	0.50$_{\pm 0.10}$
Avg	–	–	20.1%	2756 (1×)	1.03 (1×)	23.3%	1771 (0.64×)	1.65 (1.60×)	16.3%	**680** (0.25×)	**0.38** (0.37×)	**34.1%**	1315 (0.48×)	0.67 (0.65×)

For aggregate comparison, we normalize latency and cost against the sequential version: we compute the ratio of a method's per-task average latency/cost across all repositories to that of the sequential baseline's mean on the same benchmark. Both ratios are lower-is-better.

4.3 Baselines

We compare our system against the following methods, all of which receive the same input specifications, parsed into a shared Repository Interface Blueprint: (a) **Sequential**, where a single coding agent generates the entire repository, autonomously deciding task order and managing its own context; (b) **File-based Parallel**, where, from the same blueprint, a leader agent spawns one coding agent per file and the agents run concurrently and may communicate with each other, but no structural partitioning or dependency-aware scheduling is applied; (c) **Claude Code w/ Agent Teams**, where, as an external reference, we run Anthropic's Claude Code CLI in its native agent-team mode, in which a team lead spawns and coordinates independent Claude Code sessions that message each other directly; and (d) **Co-Coder (ours)**, our full system (Section 3) with cohesion-based graph-partitioning over the same blueprint to guide the multi-agent orchestration.

All methods also run the same check-and-fix loop on the visible check tests shipped with each task, which are disjoint from the held-out unit tests used for evaluation. For a fair comparison, all methods use GPT-5 Mini or Claude Sonnet 4.6 as the base model; software versions are listed in Appendix B.

5 Results

DevEval. As shown in Table 1, Co-Coder achieves the highest average pass rate (68.1%) among all methods, improving over the sequential baseline by 11.3% while reducing wall-clock latency by 45% (442 s vs. 800 s) and API cost by 28% ($0.18 vs. $0.25). On simpler projects where a single agent already succeeds (e.g., `Hybrid_Images`, `pso`), all four methods perform comparably. The advantage

of Co-Coder emerges on projects requiring cross-file coordination: on `hone`, Co-Coder passes 6.0/7 tests compared to 2.3 for the sequential baseline, and on `TextCNN` it reaches 5.0/10 versus 1.7. This suggests that cohesion-aware task partitioning helps agents produce more consistent implementations across tightly coupled files.

File-based Parallel fails to reduce latency compared to the Sequential version on DevEval (806 s vs. 800 s) yet inflates cost by 44% ($0.36 vs. $0.25) for a negligible pass-rate gain (57.7% vs. 56.8%). Poor task partitioning causes parallel agents to produce conflicting interfaces, requiring redundant re-generation that incurs extra costs without improving code quality. Claude Code w/ Agent Teams reduces latency (536 s) but suffers the lowest pass rate (54.1%), indicating that self-coordinated inter-agent orchestration cannot substitute for Co-Coder's explicit cohesion-aware task partitioning.

CodeProjectEval. On the more challenging CodeProjectEval benchmark (Table 2), the performance gap widens considerably. Co-Coder achieves 34.1% average pass rate, a gain of 14.0% over the sequential baseline (20.1%) and 10.8% over File-based Parallel (23.3%). Meanwhile, Co-Coder reduces latency by 52% (1315 s vs. 2756 s) and cost by 35% ($0.67 vs. $1.03) relative to the sequential baseline. These gains are most pronounced on projects with deep dependency structures: on `simpy`, Co-Coder passes 58.0/149 tests while Sequential passes only 10.3; on `pyjwt`, Co-Coder reaches 140.0/294 versus 79.0.

The failure mode of naïve parallelism is particularly evident on this benchmark. File-based Parallel increases cost by 60% ($1.65 vs. $1.03) relative to Sequential yet gains only 3.2% in pass rate, since concurrently generated files frequently violate cross-file type contracts, leading to expensive yet unproductive iterations. Claude Code w/ Agent Teams achieves the lowest latency (680 s) and cost ($0.38) but again at the expense of the lowest pass rate (16.3%), falling below even the sequential baseline. This confirms that aggressive parallelism without structural guidance degrades code quality on complex projects where inter-agent dependencies are critical.

Results with Claude Sonnet 4.6. To test whether the gains depend on the base model, we repeat the main experiments with Claude Sonnet 4.6 in place of GPT-5 Mini, keeping all other settings unchanged. Tables 3 and 4 report per-project results. Co-Coder again achieves the highest pass rate at the lowest latency and cost on both benchmarks: on DevEval it reaches 73.3% versus 67.4% for Sequential at 0.55× the latency and 0.87× the cost, and on CodeProjectEval 39.4% versus 34.0% at 0.32× and 0.57×, consistent with the GPT-5 Mini results.

Efficiency trade-offs. Across both benchmarks, a consistent pattern emerges: File-based Parallel trades cost for speed without improving quality, Claude Code w/ Agent Teams trades quality for speed, and Co-Coder improves all three metrics simultaneously. The key mechanism is the cohesion-based graph partition, which groups tightly coupled files to minimize inter-agent communication, combined with the event-driven shared task list that resolves inter-group dependencies without global synchronization. This design enables agents to proceed as soon as upstream files are ready, achieving the latency benefits of parallelism while preserving the interface consistency that correctness requires. A component-level ablation of hub isolation and the repair loop is reported in Appendix D; the appendix also validates the cost-model proxies (Appendix E), assesses blueprint quality and its effect on the partition (Appendix A), reports hyperparameter sensitivity (Appendix B), and adds results on C++ and Java projects (Appendix F).

6 Discussion

Why graph partition gains scale with project complexity. The advantage of cohesion-aware partitioning grows with the density of inter-file coupling. In simple projects (e.g., `Hybrid_Images` with 19 tests across loosely connected files), most methods can produce consistent interfaces because files share few type contracts. As project size increases, however, cross-file dependencies grow super-linearly: a file that imports symbols from k other files creates k potential points of interface mismatch when those files are generated independently. This trend is supported by our analysis: across the 16 CodeProjectEval projects with non-zero pass rates, edge density ($|E|/|V|$; Table 11) correlates positively with absolute test-pass improvement over Sequential (Pearson r=0.65, Spearman ρ=0.60, both $p<0.05$). Naïve parallelism exposes these dependency edges to concurrent, uncoordinated generation, so conflicts become increasingly likely as coupling rises. In contrast, Co-Coder's cohesion-

Table 3: Per-project results on DevEval with Claude Sonnet 4.6 as the base model. Metrics and conventions are the same as in Table 1.

Task	#LOC	#Tests	Sequential			File-based Parallel			Claude Code w/ Agent Teams			**Co-Coder (ours)**		
			#Pass	Time	Cost	#Pass	Time	Cost	#Pass	Time	Cost	#Pass	Time	Cost
ArXiv_digest	198	38	31.0$_{\pm 0.8}$	365$_{\pm 46}$	0.94$_{\pm 0.13}$	31.0$_{\pm 0.0}$	474$_{\pm 66}$	1.40$_{\pm 0.13}$	31.0$_{\pm 0.0}$	413$_{\pm 52}$	1.40$_{\pm 0.32}$	**31.3**$_{\pm 0.5}$	332$_{\pm 51}$	1.01$_{\pm 0.13}$
chakin	62	1	0.7$_{\pm 0.5}$	642$_{\pm 58}$	1.29$_{\pm 0.08}$	0.7$_{\pm 0.5}$	274$_{\pm 59}$	1.16$_{\pm 0.17}$	**1.0**$_{\pm 0.0}$	305$_{\pm 21}$	0.99$_{\pm 0.24}$	**1.0**$_{\pm 0.0}$	251$_{\pm 11}$	0.99$_{\pm 0.03}$
geotext	134	4	2.3$_{\pm 0.5}$	640$_{\pm 66}$	1.70$_{\pm 0.12}$	2.3$_{\pm 0.5}$	423$_{\pm 31}$	1.33$_{\pm 0.12}$	**2.7**$_{\pm 0.5}$	731$_{\pm 153}$	1.63$_{\pm 0.55}$	2.3$_{\pm 0.5}$	682$_{\pm 49}$	1.91$_{\pm 0.18}$
hone	274	7	6.7$_{\pm 0.5}$	1298$_{\pm 86}$	3.61$_{\pm 0.18}$	5.0$_{\pm 1.6}$	671$_{\pm 179}$	3.99$_{\pm 0.44}$	**7.0**$_{\pm 0.0}$	616$_{\pm 188}$	2.33$_{\pm 0.27}$	**7.0**$_{\pm 0.0}$	458$_{\pm 15}$	2.32$_{\pm 0.05}$
Hybrid_Images	144	19	**17.0**$_{\pm 0.0}$	249$_{\pm 13}$	0.66$_{\pm 0.04}$	**17.0**$_{\pm 0.0}$	334$_{\pm 43}$	0.92$_{\pm 0.21}$	**17.0**$_{\pm 0.0}$	289$_{\pm 71}$	0.94$_{\pm 0.13}$	**17.0**$_{\pm 0.0}$	148$_{\pm 12}$	0.48$_{\pm 0.05}$
lice	376	25	**13.3**$_{\pm 2.9}$	787$_{\pm 119}$	1.76$_{\pm 0.14}$	9.7$_{\pm 0.5}$	586$_{\pm 54}$	1.97$_{\pm 0.19}$	8.7$_{\pm 1.3}$	949$_{\pm 543}$	1.58$_{\pm 0.79}$	**13.3**$_{\pm 1.7}$	481$_{\pm 32}$	1.46$_{\pm 0.05}$
pso	168	5	**5.0**$_{\pm 0.0}$	429$_{\pm 59}$	1.08$_{\pm 0.15}$	**5.0**$_{\pm 0.0}$	239$_{\pm 8}$	1.02$_{\pm 0.09}$	3.3$_{\pm 2.4}$	266$_{\pm 84}$	0.61$_{\pm 0.21}$	**5.0**$_{\pm 0.0}$	200$_{\pm 8}$	0.80$_{\pm 0.02}$
readtime	284	8	4.0$_{\pm 0.0}$	779$_{\pm 66}$	1.92$_{\pm 0.11}$	2.0$_{\pm 1.6}$	354$_{\pm 43}$	1.97$_{\pm 0.14}$	3.3$_{\pm 0.9}$	664$_{\pm 303}$	1.48$_{\pm 0.35}$	**4.3**$_{\pm 0.5}$	355$_{\pm 59}$	1.42$_{\pm 0.09}$
stocktrends	384	7	3.0$_{\pm 0.0}$	2213$_{\pm 456}$	6.10$_{\pm 1.22}$	3.0$_{\pm 0.8}$	2626$_{\pm 442}$	11.70$_{\pm 2.63}$	1.3$_{\pm 1.3}$	1302$_{\pm 117}$	9.04$_{\pm 0.98}$	**3.7**$_{\pm 0.5}$	1292$_{\pm 111}$	6.38$_{\pm 0.45}$
TextCNN	403	10	3.7$_{\pm 0.9}$	904$_{\pm 53}$	2.35$_{\pm 0.05}$	3.0$_{\pm 0.0}$	268$_{\pm 32}$	1.63$_{\pm 0.11}$	0.0$_{\pm 0.0}$	864$_{\pm 590}$	2.52$_{\pm 0.73}$	**4.3**$_{\pm 0.9}$	353$_{\pm 70}$	1.73$_{\pm 0.18}$
Avg	–	–	67.4%	831 (1×)	2.14 (1×)	60.4%	625 (0.75×)	2.71 (1.27×)	60.0%	640 (0.77×)	2.25 (1.05×)	**73.3%**	**455** (0.55×)	**1.85** (0.87×)

Table 4: Per-project results on CodeProjectEval with Claude Sonnet 4.6 as the base model. Metrics and conventions are the same as in Table 1.

Task	#LOC	#Tests	Sequential			File-based Parallel			Claude Code w/ Agent Teams			**Co-Coder (ours)**		
			#Pass	Time	Cost	#Pass	Time	Cost	#Pass	Time	Cost	#Pass	Time	Cost
bplustree	1,509	356	14.0$_{\pm 6.2}$	2845$_{\pm 52}$	8.83$_{\pm 0.18}$	30.3$_{\pm 10.1}$	2779$_{\pm 671}$	14.84$_{\pm 2.31}$	27.0$_{\pm 17.7}$	4173$_{\pm 2255}$	4.28$_{\pm 0.49}$	**35.0**$_{\pm 32.3}$	1450$_{\pm 89}$	6.00$_{\pm 0.97}$
cookiecutter	2,805	375	111.7$_{\pm 4.5}$	4464$_{\pm 215}$	14.63$_{\pm 1.51}$	108.0$_{\pm 5.1}$	729$_{\pm 12}$	11.68$_{\pm 0.79}$	80.7$_{\pm 29.3}$	1475$_{\pm 107}$	7.59$_{\pm 2.65}$	**124.0**$_{\pm 6.7}$	948$_{\pm 413}$	6.55$_{\pm 1.94}$
csvs-to-sqlite	816	25	**15.3**$_{\pm 0.9}$	1189$_{\pm 190}$	3.58$_{\pm 0.63}$	9.0$_{\pm 6.4}$	1964$_{\pm 154}$	9.34$_{\pm 1.32}$	4.3$_{\pm 6.1}$	1055$_{\pm 469}$	2.07$_{\pm 1.01}$	**15.3**$_{\pm 0.9}$	911$_{\pm 77}$	3.56$_{\pm 0.18}$
deprecated	597	176	147.0$_{\pm 2.2}$	647$_{\pm 106}$	1.75$_{\pm 0.32}$	137.0$_{\pm 5.7}$	913$_{\pm 57}$	3.87$_{\pm 0.07}$	139.7$_{\pm 16.0}$	484$_{\pm 35}$	1.08$_{\pm 0.09}$	**150.0**$_{\pm 5.7}$	608$_{\pm 57}$	1.96$_{\pm 0.19}$
drf-simplejwt	1,712	191	36.7$_{\pm 47.7}$	3295$_{\pm 71}$	9.51$_{\pm 0.09}$	4.0$_{\pm 0.0}$	565$_{\pm 20}$	9.19$_{\pm 0.21}$	5.0$_{\pm 0.8}$	687$_{\pm 62}$	2.05$_{\pm 0.05}$	**60.0**$_{\pm 39.6}$	665$_{\pm 23}$	5.41$_{\pm 0.16}$
flask	9,314	482	0.0$_{\pm 0.0}$	7238$_{\pm 169}$	23.41$_{\pm 0.92}$	0.0$_{\pm 0.0}$	2214$_{\pm 109}$	29.74$_{\pm 1.23}$	0.0$_{\pm 0.0}$	1610$_{\pm 271}$	27.75$_{\pm 2.19}$	0.0$_{\pm 0.0}$	1252$_{\pm 42}$	9.71$_{\pm 1.77}$
imapclient	3,531	267	69.7$_{\pm 2.1}$	3867$_{\pm 193}$	11.65$_{\pm 0.19}$	68.3$_{\pm 2.4}$	1155$_{\pm 166}$	12.17$_{\pm 0.98}$	66.3$_{\pm 3.7}$	1159$_{\pm 185}$	7.42$_{\pm 2.96}$	**70.3**$_{\pm 7.4}$	1145$_{\pm 229}$	7.01$_{\pm 1.46}$
parsel	1,128	250	17.0$_{\pm 2.9}$	1681$_{\pm 96}$	4.81$_{\pm 0.29}$	39.0$_{\pm 1.4}$	1206$_{\pm 302}$	6.16$_{\pm 1.10}$	25.7$_{\pm 8.3}$	751$_{\pm 52}$	1.93$_{\pm 0.23}$	**47.3**$_{\pm 20.0}$	788$_{\pm 75}$	3.20$_{\pm 0.10}$
portalocker	1,958	71	0.0$_{\pm 0.0}$	2478$_{\pm 218}$	6.81$_{\pm 0.61}$	0.0$_{\pm 0.0}$	1267$_{\pm 301}$	8.20$_{\pm 1.35}$	0.0$_{\pm 0.0}$	1271$_{\pm 531}$	5.66$_{\pm 1.11}$	0.0$_{\pm 0.0}$	996$_{\pm 199}$	4.79$_{\pm 0.50}$
pyjwt	2,690	294	179.7$_{\pm 0.9}$	2977$_{\pm 26}$	9.46$_{\pm 0.46}$	170.0$_{\pm 1.4}$	755$_{\pm 73}$	9.81$_{\pm 0.86}$	147.3$_{\pm 38.5}$	859$_{\pm 81}$	6.50$_{\pm 2.78}$	**190.7**$_{\pm 5.0}$	1015$_{\pm 82}$	4.81$_{\pm 0.36}$
python-hl7	2,434	100	29.0$_{\pm 21.9}$	3023$_{\pm 155}$	9.27$_{\pm 0.81}$	43.3$_{\pm 4.5}$	2741$_{\pm 137}$	18.39$_{\pm 1.91}$	41.7$_{\pm 12.8}$	2256$_{\pm 715}$	7.19$_{\pm 0.95}$	**44.0**$_{\pm 7.0}$	909$_{\pm 573}$	4.29$_{\pm 2.48}$
rsa	2,949	100	61.7$_{\pm 11.2}$	3190$_{\pm 392}$	8.92$_{\pm 0.66}$	43.0$_{\pm 22.9}$	1060$_{\pm 400}$	8.95$_{\pm 0.84}$	46.3$_{\pm 32.8}$	1279$_{\pm 385}$	5.42$_{\pm 0.91}$	**65.7**$_{\pm 2.4}$	1051$_{\pm 190}$	5.70$_{\pm 0.42}$
simpy	2,184	149	64.0$_{\pm 20.6}$	3407$_{\pm 329}$	9.70$_{\pm 0.96}$	37.7$_{\pm 22.1}$	2783$_{\pm 256}$	19.33$_{\pm 0.89}$	50.7$_{\pm 23.8}$	1372$_{\pm 546}$	7.23$_{\pm 1.92}$	**76.7**$_{\pm 21.3}$	1239$_{\pm 93}$	5.49$_{\pm 0.18}$
tinydb	2,170	204	100.0$_{\pm 3.7}$	2004$_{\pm 149}$	5.65$_{\pm 0.27}$	106.0$_{\pm 4.2}$	561$_{\pm 55}$	5.71$_{\pm 0.11}$	117.0$_{\pm 2.2}$	940$_{\pm 437}$	3.64$_{\pm 0.43}$	**141.7**$_{\pm 24.5}$	701$_{\pm 43}$	3.40$_{\pm 0.28}$
trailscraper	890	93	**33.0**$_{\pm 2.9}$	2743$_{\pm 57}$	8.14$_{\pm 0.20}$	26.3$_{\pm 2.9}$	1028$_{\pm 131}$	9.56$_{\pm 0.79}$	**33.0**$_{\pm 4.6}$	981$_{\pm 156}$	6.47$_{\pm 2.34}$	30.0$_{\pm 1.4}$	931$_{\pm 50}$	5.23$_{\pm 0.26}$
voluptuous	3,100	161	56.3$_{\pm 11.0}$	2992$_{\pm 39}$	10.21$_{\pm 0.20}$	**64.7**$_{\pm 6.9}$	1497$_{\pm 206}$	11.18$_{\pm 0.54}$	40.3$_{\pm 3.4}$	1099$_{\pm 447}$	2.94$_{\pm 0.56}$	63.0$_{\pm 4.9}$	1081$_{\pm 94}$	5.07$_{\pm 0.55}$
xmnlp	1,504	23	1.7$_{\pm 0.5}$	4849$_{\pm 72}$	15.54$_{\pm 0.25}$	0.7$_{\pm 0.9}$	1492$_{\pm 260}$	19.95$_{\pm 2.89}$	**5.0**$_{\pm 2.2}$	1356$_{\pm 148}$	11.31$_{\pm 1.30}$	4.0$_{\pm 2.2}$	1120$_{\pm 118}$	10.22$_{\pm 0.57}$
zxcvbn	1,386	31	**18.3**$_{\pm 1.3}$	1648$_{\pm 110}$	5.97$_{\pm 0.46}$	17.7$_{\pm 1.3}$	717$_{\pm 101}$	4.12$_{\pm 0.62}$	17.7$_{\pm 1.3}$	1279$_{\pm 383}$	3.24$_{\pm 0.54}$	**18.3**$_{\pm 1.7}$	633$_{\pm 40}$	4.08$_{\pm 0.76}$
Avg	–	–	34.0%	3030 (1×)	9.32 (1×)	30.2%	1412 (0.47×)	11.79 (1.26×)	29.6%	1338 (0.44×)	6.32 (0.68×)	**39.4%**	**969** (0.32×)	**5.36** (0.57×)

based partition places strongly coupled files within the same agent, internalizing the majority of dependency edges while resolving the remainder through the shared task list's topological ordering.

Latency–correctness trade-off. Our results reveal a fundamental tension between minimizing wall-clock latency and maximizing code correctness. Claude Code w/ Agent Teams achieves the lowest latency among baselines on CodeProjectEval by aggressively distributing work across independent agents, but this comes at the cost of the lowest pass rate—falling below even the sequential baseline on CodeProjectEval (16.3% vs. 20.1%). The sequential baseline, conversely, maintains interface consistency but cannot exploit any parallelism. Co-Coder navigates this trade-off through event-driven scheduling: rather than launching all tasks simultaneously or executing them serially, it releases each file for generation as soon as its upstream dependencies are satisfied. This yields latency reductions of 45–52% over Sequential while simultaneously improving pass rate, because agents begin work with access to the actual interfaces produced by upstream files rather than speculative placeholders.

Effect of the base model. Switching from GPT-5 Mini to Claude Sonnet 4.6 makes the sequential baseline stronger, raising its pass rate from 56.8% to 67.4% on DevEval and from 20.1% to 34.0% on CodeProjectEval. Both parallel baselines fall below the sequential version on both benchmarks, while Co-Coder still improves on it. A stronger single agent maintains interface consistency on its own, so uncoordinated parallel agents lose more by breaking it, while the cohesion-based partition preserves it. Co-Coder's pass-rate margin narrows as the sequential baseline improves, but its efficiency gains grow, reaching a 3.13× speedup and a 42% cost reduction on CodeProjectEval compared with 2.10× and 35% under GPT-5 Mini. With only two base models these trends are suggestive but indicate that structured coordination remains valuable as models improve.

Limitations. The cost model relies on proxies: interface size stands in for generation time and symbol similarity for communication volume. Both track the measured quantities (Appendix E), but Co-Coder does not optimize $T(P)$ against measured costs. The benchmarks fix the file structure, which keeps the comparison controlled but leaves open-ended settings, where agents must also plan the file layout, untested; since planning the layout is orthogonal to partitioning, Co-Coder can run on the output of any planner. When a project offers no exploitable parallelism, Co-Coder uses a single agent; this happens on 4 of the 28 projects, all with one or two source files. Finally, the main

evaluation covers only Python; a study on one C++ and one Java project (Appendix F) points the same way but is too small to establish generality across languages.

7 Related Works

7.1 Repository-level Coding

LLMs handle small, self-contained programming tasks well [8] but degrade sharply on real-world repository-scale software engineering, where modifications must be coordinated across many interdependent files [9, 19]. Recent agent systems mitigate this with structured agent-computer interfaces and staged pipelines: SWE-agent [33] and OpenHands [29] expose controlled tool environments for iterative patch refinement, while Agentless [32] decomposes bug resolution into localization, repair, and validation stages with explicit checkpoints. A complementary line tackles context management through retrieval-augmented generation [30], though selecting informative context over a large codebase in long-horizon settings remains an open problem [33, 29].

7.2 Multi-agent Systems Orchestration

LLM-based multi-agent orchestration has largely prioritized task quality through role specialization and structured communication. Organizational-style frameworks assign agents to specialized roles connected by predefined workflows [7, 23]; conversational frameworks generalize coordination to free-form dialogue between agents and arbitrary topologies [16, 31]; and graph-structured approaches encode agent interactions as optimizable graphs or DAG-based collaboration networks to improve scaling [35, 24]. Across these designs, orchestration is performance-oriented and relies on sequential handoffs or runtime communication protocols, so wall-clock latency scales with task complexity rather than with available agents. Recent diagnostic studies trace this to context mismanagement, shallow communication, and a spatial-semantic coordination asymmetry in which agents agree on *where* to edit but fail to coordinate *what* to implement [2, 20, 13], indicating that the bottleneck is structural rather than communicative. Co-Coder addresses this by reframing orchestration as a graph partitioning problem, assigning each agent a structurally cohesive group of files and eliminating most cross-agent dependencies before execution begins.

7.3 Efficient Planning

Efficient decomposition and scheduling of dependent tasks have long been studied in classical settings, from industrial workflow allocation [10] to heterogeneous distributed computing [27], where DAG-based execution and critical-path methods are standard tools for minimizing wall-clock latency under dependency constraints. Recent LLM-based systems adopt similar structures: graph-enhanced asynchronous plan decomposition [18] and temporal constraint-based planning [4] for general reasoning, CodePlan [1] for single-agent repository editing via DAG-ordered file edits, and LLMCompiler [14] for parallel function-call execution within a single agent. Co-Coder extends these paradigms to the multi-agent setup where the dependency graph simultaneously drives task partitioning across agents and the execution order, coupling allocation and scheduling into a single optimization problem.

8 Conclusion

We identify task partitioning as a key bottleneck in multi-agent LLM coordination and cast it as a graph optimization problem capturing the communication-to-computation trade-off. Our method, Co-Coder, achieves up to $3.13\times$ wall-clock speedup and improved pass rate on DevEval and CodeProjectEval across two base models, with the largest gains on the most dependency-dense projects. This suggests cohesion-based orchestration as a broader design principle for multi-agent LLM systems.

Acknowledgments and Disclosure of Funding

This research was supported by the National Science Foundation (NSF) under awards III-2505097, CCF-2212559, and CCF-1918651. The authors declare no competing interests.

References

[1] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram K. Rajamani, Balasubramanyan Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *CoRR*, abs/2309.12499, 2023.

[2] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya G. Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? *CoRR*, abs/2503.13657, 2025.

[3] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In Marina C. Chen and Robert Halstead, editors, *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, May 19-22, 1993*, pages 1–12. ACM, 1993.

[4] Zifeng Ding, Sikuan Yan, Moy Yuan, Xianglong Hu, Fangru Lin, and Andreas Vlachos. TCP: a benchmark for temporal constraint-based planning. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng, editors, *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 22452–22475. Association for Computational Linguistics, 2025.

[5] Pareesa Ameneh Golnari, Adarsh Kumarappan, Wen Wen, Xiaoyu Liu, Gabriel Ryan, Yuting Sun, Shengyu Fu, and Elsie Nallipogu. Devbench: A realistic, developer-informed benchmark for code generation models. *CoRR*, abs/2601.11895, 2026.

[6] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, 1966.

[7] 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. Metagpt: Meta programming for A multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[8] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[9] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[10] Josef Kallrath. Planning and scheduling in the process industry. *OR Spectr.*, 24(3):219–250, 2002.

[11] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[12] Brian W. Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.*, 49(2):291–307, 1970.

[13] Arpandeep Khatua, Hao Zhu, Peter Tran, Arya Prabhudesai, Frederic Sadrieh, Johann K. Lieberwirth, Xinkai Yu, Yicheng Fu, Michael J. Ryan, Jiaxin Pei, and Diyi Yang. Cooperbench: Why coding agents cannot be your teammates yet. *CoRR*, abs/2601.13295, 2026.

[14] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 24370–24391. PMLR / OpenReview.net, 2024.

[15] Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. Prompting large language models to tackle the full software development lifecycle: A case study. In Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert, editors, *Proceedings of the 31st International Conference on Computational Linguistics*, pages 7511–7531, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics.

[16] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: communicative agents for "mind" exploration of large language model society. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[17] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, Findings of ACL, pages 3603–3614. Association for Computational Linguistics, 2024.

[18] Fangru Lin, Emanuele La Malfa, Valentin Hofmann, Elle Michelle Yang, Anthony G. Cohn, and Janet B. Pierrehumbert. Graph-enhanced large language models in asynchronous plan reasoning. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 30108–30134. PMLR / OpenReview.net, 2024.

[19] Tianyang Liu, Canwen Xu, and Julian J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[20] Emanuele La Malfa, Gabriele La Malfa, Samuele Marro, Jie M. Zhang, Elizabeth Black, Michael Luck, Philip Torr, and Michael J. Wooldridge. Large language models miss the multi-agent mark. *CoRR*, abs/2505.21298, 2025.

[21] R.C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003.

[22] Elizabeth Mieczkowski, Katherine M. Collins, Ilia Sucholutsky, Natalia Vélez, and Thomas L. Griffiths. Language model teams as distributed systems. *CoRR*, abs/2603.12229, 2026.

[23] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 15174–15186. Association for Computational Linguistics, 2024.

[24] Chen Qian, Zihao Xie, Yifei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Scaling large language model-based multi-agent collaboration. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[25] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[26] Alvin Wei Ming Tan, Chunhua Yu, Bria Long, Wanjing Ma, Tonya Murray, Rebecca D. Silverman, Jason D. Yeatman, and Michael C. Frank. Devbench: A multimodal developmental

benchmark for language learning. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024.

[27] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Syst.*, 13(3):260–274, 2002.

[28] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[29] 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, Junyang Lin, and et al. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[30] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 53270–53290. PMLR / OpenReview.net, 2024.

[31] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *CoRR*, abs/2308.08155, 2023.

[32] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *CoRR*, abs/2407.01489, 2024.

[33] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024.

[34] Qianhui Zhao, Li Zhang, Fang Liu, Junhang Cheng, Chengru Wu, Junchen Ai, Qiaoyuanhe Meng, Lichen Zhang, Xiaoli Lian, Shubin Song, and Yuanping Guo. Towards realistic project-level code generation via multi-agent collaboration and semantic architecture modeling. *CoRR*, abs/2511.03404, 2025.

[35] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Gptswarm: Language agents as optimizable graphs. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 62743–62767. PMLR / OpenReview.net, 2024.

A Quality Control of Repository Interface Blueprint

The correctness of the Repository Interface Blueprint (RIB) is critical, as all subsequent steps—coupling weight estimation, graph partitioning, and parallel scheduling—depend on it. We employ an LLM-based judge to score the generated architecture on a 1–10 scale across four dimensions, namely requirement coverage, module partitioning, interface consistency, and dependency management. If the score falls below a threshold of 8, the RIB is refined by feeding the judge's critique back to the generation model, and re-judged. This loop iterates for up to three rounds.

Blueprint accuracy. We compare the generated RIBs of all 28 projects with the reference implementations using static analysis, with no LLM involved. The RIBs cover nearly all files to be generated (file coverage 0.994; the few misses are empty `__init__.py` files). Table 5 reports precision and recall for file dependencies and for interface descriptions (class, function, and constant names). Dependency errors are mostly omissions (about 9%). Interface errors include both omissions and extra entries (roughly 13–15%); this is an upper bound on the true error rate, since legitimate design differences such as naming or file placement also count as errors.

Table 5: Accuracy of the generated RIBs against the reference implementations, over all 28 projects.

Element	Precision	Recall
File dependencies	0.982	0.910
Interface descriptions	0.869	0.843

Sensitivity of the partition to dependency errors. Dependency errors affect the partitioning stage. Starting from a ground-truth RIB calibrated against the reference implementation, we delete file dependencies at controlled rates, recompute the partition, and compare it with the partition of the ground-truth RIB using the Adjusted Rand Index (ARI), which is 1 for identical partitions and 0 for chance-level agreement. Table 6 shows that at the measured omission rate (about 9%, between the 5% and 10% levels) the partition stays largely unchanged; it degrades markedly only at two to four times that rate.

Table 6: Partition agreement (mean ARI over projects) with the ground-truth partition after deleting file dependencies from the RIB.

Deletion rate	5%	10%	20%	40%
Mean ARI	0.898	0.848	0.751	0.515

Sensitivity of pass rate to interface errors. We perturb interface descriptions by deleting or corrupting entries, at rates beyond the measured error rate, and rerun Co-Coder end-to-end on CodeProjectEval with GPT-5 Mini (three runs). Pass rate drops from 34.1% to between 28.5% and 31.3% but remains above all baselines (Table 7). The RIB is not the agents' only source of information: the requirement documents remain in the workspace, and interface errors that surface as test failures can be fixed in the repair loop, at the cost of extra repair rounds.

Table 7: Average pass rate on CodeProjectEval (GPT-5 Mini) with injected interface errors in the RIB. Baseline values are from Table 2.

Co-Coder						
Original	+5%	+10%	+15%	Sequential	File-based Parallel	Claude Code w/ Agent Teams
34.1%	30.2%	31.3%	28.5%	20.1%	23.3%	16.3%

Judge validation. The judge does not produce the RIB; it only gates quality, sending RIBs below the threshold back for revision. On RIBs with errors injected at varying rates, judge scores track true quality (within-project average Spearman ρ=0.657, N=100). In addition, evaluators not involved in this work independently scored 50 RIBs under the same rubric without seeing the judge's scores; on the binary pass/revise decision, their agreement with the judge is 92.3%.

B Implementation Details

Table 8 lists the hyperparameters used throughout all experiments. All values are fixed across repositories and benchmarks; no per-project tuning is performed.

Software versions. Sequential, File-based Parallel, and Co-Coder are built on a consistent OpenHands SDK `v1.11.4`. Claude Code runs at `v2.1.119` with LiteLLM `v1.83.8` as the API proxy.

Table 8: Hyperparameters of Co-Coder. All values are held constant across the 28 evaluated repositories.

Component	Parameter	Value
Edge weight (Eq. 4)	Defined-symbol weight in $\mathbf{s}_i$	2
	Referenced-type weight in $\mathbf{s}_i$	1
	Cosine scale γ	10
	Default weight (no shared symbols)	0.1
Community detection (Sec. 3.2)	Markov time t	1.0
Hub isolation (Sec. 3.2)	Role threshold (fan-in/fan-out ratio)	0.4
RIB generation (Sec. 3.1)	Maximum refinement rounds	3
Test-driven repair (Sec. 3.3)	Maximum repair iterations	10

Edge weight construction. Each file's symbol vector $\mathbf{s}_i$ assigns weight 2 to symbols the file *defines* (class names, top-level functions, global constants) and weight 1 to types it *references* in parameter annotations and to method names defined inside its classes, after filtering builtins. The cosine similarity is scaled by $\gamma{=}10$ to place edge weights in the same numerical range as the trade-off coefficient α in Eq. (3). Edges between files that share no symbols receive a floor weight of 0.1 to preserve graph connectivity.

Files as the unit of assignment. Files are the unit of isolation rather than the unit of work: one agent may own several files, but no two agents edit the same file. Agent edits are file-scoped, so splitting a file across agents lets concurrent writes conflict. In a preliminary test, two agents each implemented half of the functions of the same file (12 files with at least eight functions, one per project, across both benchmarks): 67% of the 98 concurrent writes failed on conflicts, and the forced rewrites increased output tokens by $2.88\times$ and wall-clock latency by $2.14\times$ compared with a single writer per file.

Agent communication. Each group agent receives a fixed system prompt and an initial message stating its responsibility and the files of its group, and it queries interface specifications from the RIB through a tool. Agents coordinate through the shared task list, where they claim and complete tasks and downstream agents are woken once their prerequisite files are done, and through a message bus that supports direct and broadcast messages between the leader and group agents and among group agents. We add no custom prompt caching; caching is the provider's default behavior, identical for all methods. The complete prompt templates for RIB generation, the RIB judge, and the leader and group agents are available in our released code repository.

Role classification. A file is classified as an *in-hub* if its in-degree ratio $|\{v_j : (v_j, v_i) \in E\}| \,/\, (|V| - 1)$ exceeds 0.4, indicating it is widely depended upon; and as an *out-hub* (integration file) if its out-degree ratio $|\{v_j : (v_i, v_j) \in E\}| \,/\, (|V| - 1)$ exceeds the same threshold, indicating it aggregates many upstream modules. The threshold is intentionally conservative: raising it merges more utility files into community clusters, increasing intra-group serialization; lowering it isolates too many files into singletons, fragmenting cohesive modules.

Hyperparameter sensitivity. All values in Table 8 are set a priori (library defaults, the simplest option, or conservative values) and shared across all 28 projects; neither benchmark has a development split, and no project was held out for tuning. The coefficient α in Eq. (3) is not set directly: its trade-off is controlled through the Markov time t (Section 3.2). Table 9 summarizes offline sweeps

over all 28 RIBs with the same random seed as the main experiments. Only the hub threshold changes the partitions noticeably, and disabling hub isolation altogether corresponds to the ablation in Appendix D. The three edge-weight parameters have little effect on the partitions (Table 10).

Table 9: Hyperparameter sensitivity. Partition changes are counted against the default configuration over the 28 projects.

Parameter	Default	Range	Effect
Defined : referenced weight	2 : 1	1 : 1–4 : 1	Partition changes slightly on 3–4 projects
Cosine scale γ	10	0.05–1000	Partition changes on 4–5 projects for $\gamma \geq 1$ (up to 9 at $\gamma = 0.05$); only the ratio of γ to the floor weight matters
Floor weight	0.1	0.01–1.0	Partition changes on 2–5 projects
Hub threshold	0.4	0.2–0.8	Average group count falls from 6.00 to 4.07 as the threshold rises from 0.2 to 0.8
Markov time t	1.0	0.5–4.0	Doubling t changes the partition on 6 projects; rerunning them lowers average pass rate by 1.5 pp (DevEval) and 1.0 pp (CodeProjectEval)
Refinement / repair caps	3 / 10	from logs	97.2% of RIBs pass the judge in the first round; 8.4% of runs use the full repair budget

Table 10: Results at every swept value of Table 9: the number of projects (out of 28) whose partition is identical to the default configuration, and the average group count. Default values are in bold.

Parameter	Values	Identical partitions	Avg. #groups
Defined : referenced weight	1, 1.5, **2**, 3, 4 (: 1)	24, 24, 28, 24, 25	5.18, 5.18, 5.21, 5.25, 5.21
Cosine scale γ	0.05, 0.5, 1, **10**, 100, 1000	19, 22, 23, 28, 24, 24	5.21, 5.18, 5.18, 5.21, 5.21, 5.18
Floor weight	0.01, 0.05, **0.1**, 0.5, 1.0	24, 26, 28, 24, 23	5.21, 5.21, 5.21, 5.11, 5.18
Hub threshold	0.2, 0.3, **0.4**, 0.5, 0.6, 0.8	11, 15, 28, 20, 19, 17	6.00, 5.93, 5.21, 4.96, 4.32, 4.07
Markov time t	0.5, 0.8, **1.0**, 1.5, 2.0, 4.0	17, 21, 28, 22, 22, 18	5.14, 5.36, 5.21, 5.14, 5.11, 4.93

C Per-Repository Partition Statistics

Table 11 characterizes the dependency graph $G{=}(V, E)$ and the partition produced by Algorithm 1 for each repository. CodeProjectEval projects exhibit markedly higher edge density ($|E|/|V| \approx 1.5$) than DevEval ($|E|/|V| \approx 0.2$), explaining why cohesion-aware partitioning yields larger gains on CodeProjectEval (Section 5). Across all runs, Co-Coder uses 5.1 group agents per task on average (range 1–11). Four projects (`ArXiv_digest`, `Hybrid_Images`, `geotext`, and `lice`), each with one or two source files, form a single group and run as a single agent; splitting them further would only add communication. Partitioning is CPU-only graph computation with no LLM calls; its median wall-clock time in the execution logs is about 0.7 s, including framework overhead, and it is included in the reported latency.

Note on file and LOC counts. The *#Files* and *#LOC* columns report only source files that agents are tasked to generate. A small number of `.py` files in the original benchmarks are auto-generated artifacts (e.g., Django database migrations in `drf-simplejwt`, static data tables in `zxcvbn`) or packaging scripts (`setup.py`) that do not contain meaningful implementation logic; these are pre-populated in the agent workspace and excluded from the generation task. Consequently, the file counts for a few repositories differ slightly from those reported in the original benchmark papers.

D Component-Level Ablation

We ablate the components of Co-Coder. The File-based Parallel baseline in the main results already ablates the partitioning itself, with markedly worse pass rate and cost. We run the ablation on the projects with at least one structural hub, $\geq$5 files, and $\geq$4 groups (Table 13), keeping all other settings unchanged; Table 12 reports the results.

Table 11: Dependency graph and partition statistics. *#Files* and *#LOC* count only source files that agents must generate; auto-generated data files (e.g., Django migrations, static lookup tables) are pre-populated in the workspace and excluded from the generation task. #Edges, Density, #Hubs, #Groups, MaxGrp, and #Sing are computed on the RIB. Its file list matches the reference implementation except on seven projects, where it adds or omits auxiliary files (`__init__.py`, `__main__.py`, `setup.py`, or a test script) or splits one module into several (stocktrends); the required functionality is the same. *#Edges*: directed dependency edges $|E|$; *Density*: edge density $|E|/|V|$; *#Hubs*: structural hubs isolated per Section 3.2; *#Groups*: total partitions K; *MaxGrp*: largest cohesive cluster size; *#Sing*: singleton partitions (isolated hubs, lifted files, and files that form their own community).

Task	#Files	#LOC	#Edges	Density	#Hubs	#Groups	MaxGrp	#Sing
DevEval								
ArXiv_digest	2	198	0	0.00	0	1	1	1
chakin	2	62	1	0.33	0	2	2	1
geotext	2	134	0	0.00	0	1	2	0
hone	6	274	1	0.14	0	4	4	3
Hybrid_Images	1	144	0	0.00	0	1	1	1
lice	2	376	1	0.50	0	1	2	0
pso	3	168	0	0.00	0	2	2	1
readtime	5	284	2	0.40	0	3	3	2
stocktrends	2	384	0	0.00	0	8	2	7
TextCNN	6	403	4	0.80	0	5	1	5
CodeProjectEval								
bplustree	8	1,509	15	1.88	1	5	3	3
cookiecutter	19	2,805	36	2.00	2	10	4	6
csvs-to-sqlite	3	816	1	0.33	1	2	2	1
deprecated	3	597	2	0.67	1	2	2	1
drf-simplejwt	18	1,712	29	1.61	2	11	5	7
flask	24	9,314	65	2.71	1	8	8	3
imapclient	17	3,531	25	1.47	0	10	5	6
parsel	5	1,128	5	1.00	0	4	2	3
portalocker	9	1,958	16	1.78	2	7	3	6
pyjwt	12	2,690	27	2.25	1	6	4	3
python-hl7	11	2,434	16	1.45	0	4	5	2
rsa	14	2,949	25	1.79	1	10	3	7
simpy	12	2,184	20	2.00	1	7	4	6
tinydb	10	2,170	11	1.10	0	6	4	4
trailscraper	13	890	16	1.23	0	8	2	3
voluptuous	6	3,100	12	2.00	2	4	3	3
xmnlp	24	1,504	23	0.96	0	9	5	0
zxcvbn	6	1,386	6	1.00	1	5	2	4

Removing hub isolation drops the pass rate by 7.2 percentage points with no improvement in latency or cost. Without it, Infomap pulls hub files and their dependents into one giant community, and the average group count falls from 7.3 to 4.5, so a single agent takes on non-cohesive files and its context becomes cluttered. Removing the repair loop drops the pass rate by 11.4 percentage points while saving about 417 seconds and $0.16 per project; generation errors lose their chance to be corrected, and the savings correspond exactly to the repair overhead. Table 13 lists, for each partition-stage component, the number of projects on which it is activated and the corresponding tasks. Latent-parallelism lifting is activated on only 6 of the 28 projects, so its effect on the aggregate results is limited.

E Validation of the Cost Model

Both terms of $T(P)$ are estimated from the RIB rather than measured, so we check each proxy against measurements.

Table 12: Component-level ablation on the projects with at least one structural hub, $\geq$5 files, and $\geq$4 groups (Table 13). All variants are rerun for this ablation. Pass rate, wall-clock latency, and API cost are averaged over projects.

Variant	Pass rate	Latency (s)	Cost ($)
Co-Coder (full)	**36.4%**	1285	0.603
w/o hub isolation	29.2%	1353	0.608
w/o repair loop	25.0%	**868**	**0.440**

Table 13: Activation of the partition-stage components across the 28 evaluated projects. *# Activation*: number of projects on which the component changes the partition; *Activated tasks*: the corresponding projects.

Component	# Activation	Activated tasks
Structural hub isolation	12 / 28	bplustree, cookiecutter, csvs-to-sqlite, deprecated, drf-simplejwt, flask, portalocker, pyjwt, rsa, simpy, voluptuous, zxcvbn
Latent-parallelism lifting	6 / 28	drf-simplejwt, flask, rsa, simpy, trailscraper, xmnlp

Communication weights. The coupling weight c_{ij} in Eq. (4) is a cosine similarity, which captures the relative coupling strength between two files rather than the absolute volume of context they exchange. Because Infomap normalizes edge weights into transition probabilities (Eq. (5)), the partition depends mainly on relative weights. Replacing the cosine with the unnormalized inner product, which preserves volume, and rerunning the partitioning on all 28 RIBs leaves the partitions largely unchanged (mean ARI 0.95).

Computation weights. Over the execution logs, the Spearman rank correlation between the total interface size of each agent's group and the group's measured implementation time, from claiming its first task to completing its last, is ρ=0.66 (N=424 groups).

F Results on Non-Python Projects

Co-Coder's partitioning and scheduling operate on the RIB and use no Python-specific features; only the test execution script depends on the language. We run one C++ and one Java project with dense cross-file dependencies and at least eight source files, taken from the non-Python portion of the original DevEval release [15], using GPT-5 Mini and three runs. Table 14 shows that Co-Coder attains the highest average pass rate at the lowest latency and cost, consistent with the Python results. With two projects, this is a sanity check rather than evidence of generality across languages.

Table 14: Results on non-Python projects (GPT-5 Mini). Each cell reports #Pass / Time (s) / Cost (USD); the Avg row reports pass rate and the latency and cost ratios to Sequential.

Project	Lang	#Files	#Tests	Sequential	File-based Parallel	CC w/ Agent Teams	Co-Coder
logistic_system	C++	8	17	10.0 / 1381 / 0.60	8.0 / 498 / 0.88	8.0 / 510 / 0.55	**11.7** / 471 / 0.42
redis-cache	Java	10	17	7.7 / 1827 / 0.56	**14.0** / 1138 / 1.01	7.7 / 750 / 0.55	13.7 / 668 / 0.51
Avg				52.1% / 1× / 1×	64.7% / 0.49× / 1.64×	46.2% / 0.39× / 0.95×	**74.7%** / 0.35× / 0.81×

G Clarification on Benchmark Naming

This paper evaluates on two benchmarks: DevEval [15] and CodeProjectEval [34]. Note that DevEval was formerly named "DevBench" in its earlier arXiv version. Readers should be aware that other benchmarks [17, 26, 5] share the same names.

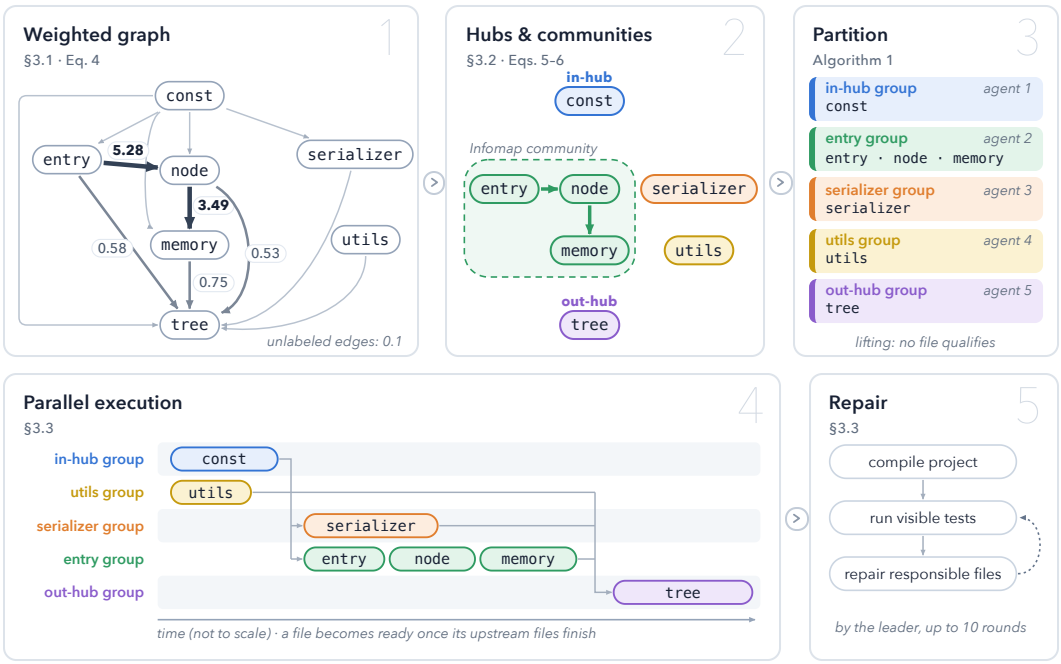


Figure 3: Worked example on `bplustree`, the project shown in Figure 2. (1) Edges from the RIB are heavy when one file defines symbols the other uses (`entry`–`node`, `node`–`memory`) and light when the two files share few or no symbols (Eq. (4)). (2) `const`, imported by most files, is an in-hub, and `tree`, which imports most files, is an out-hub; each gets its own group. Infomap puts the strongly linked `entry`, `node`, and `memory` together, so the two heaviest edges stay uncut (Eqs. (2), (5)–(6)). (3) No file meets the lifting conditions, so the five groups are final, one agent each (Algorithm 1). (4) The groups run in parallel, and a file can start once its upstream files finish; `tree`, which imports all other files, runs last. (5) The leader then runs the visible tests and fixes the files responsible for failures, for up to ten rounds.

H Worked Example

Figure 3 traces `bplustree` through each stage of Co-Coder.

I Repository Interface Blueprint Example

Figure 4 shows a condensed excerpt of the Repository Interface Blueprint (RIB) produced for the `zxcvbn` project (6 files, 6 dependency edges). The RIB specifies, for each file: (i) the files it depends on, (ii) a natural-language description of its role, and (iii) the function signatures it defines. This structured representation provides the inputs to graph construction (Section 3.1): dependencies yield the edge set E, and shared symbols across function signatures yield the coupling weights c_{ij} via Eq. (4).

RIB excerpt — `zxcvbn` (6 files, 6 dependency edges)


```
[{
  "name": "zxcvbn",
  "files": [
    {
      "path": "zxcvbn/__init__.py",
      "dependencies": [
        "zxcvbn/feedback.py", "zxcvbn/matching.py",
        "zxcvbn/scoring.py", "zxcvbn/time_estimates.py"
      ],
      "description": "Package initializer exposing the public API.",
      "functions": [{
        "name": "zxcvbn",
        "parameters": [
          {"name": "password", "type": "str"},
          {"name": "user_inputs", "type": "list[str] or None"},
          {"name": "max_length", "type": "int", "default": "72"}
        ]
      }]
    },
    {
      "path": "zxcvbn/scoring.py",
      "dependencies": [],
      "description": "Guess estimation and match sequence optimization.",
      "functions": [
        {"name": "most_guessable_match_sequence", "parameters": ["..."]},
        {"name": "estimate_guesses", "parameters": ["..."]}
      ]
    },
    {
      "path": "zxcvbn/matching.py",
      "dependencies": ["zxcvbn/scoring.py"],
      "description": "Pattern detection (dictionary, spatial, sequence, etc.).",
      "functions": [
        {"name": "omnimatch", "parameters": ["..."]},
        {"name": "dictionary_match", "parameters": ["..."]}
      ]
    }
  ]
}]
```


Figure 4: Condensed RIB for the `zxcvbn` project. Three of six files shown; remaining files (`feedback.py`, `time_estimates.py`, `__main__.py`) omitted for space.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Contributions stated in the abstract and introduction are directly supported by experimental results in Sections 5–6.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 6 discusses limitations including scope of evaluation and failure cases.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: The paper presents a formal optimization objective and algorithmic instantiation but does not claim theoretical guarantees such as approximation bounds or convergence proofs.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Section 4 specifies benchmarks, base models, baselines, and metrics. Appendix B lists all hyperparameters and software versions. The algorithm is fully described with pseudocode in Algorithm 1.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Our code is released at `https://github.com/Flitternie/CoCoder` with instructions for running Co-Coder, the baselines, and the evaluation. The benchmarks (DevEval, CodeProjectEval) are publicly available.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 4 describes benchmarks, base models, baselines, and evaluation protocol. Appendix B provides a complete hyperparameter table with all values fixed across repositories.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Tables 1–4 report mean $\pm$ standard deviation over three independent runs for every method and repository.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: All inference uses commercial APIs (GPT-5 Mini and Claude Sonnet 4.6); no local GPU is required. Per-repository wall-clock latency and API cost are reported in the results tables.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research uses publicly available benchmarks and commercial APIs. No human subjects, private data, or dual-use concerns are involved.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

Justification: Co-Coder is an orchestration layer for existing LLM APIs and introduces no new model capabilities or tool access. Its positive impact is lower cost and latency for repository-level code generation, making such tooling more accessible to resource-constrained developers. A potential negative impact is that faster automated generation can reduce opportunities for human review; Co-Coder's test-driven repair loop verifies generated code against tests, and we recommend human review of generated repositories before production use.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release pre-trained models or scraped datasets. Co-Coder is an orchestration framework that coordinates existing LLM APIs and poses no additional misuse risk.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

Justification: Both benchmarks (DevEval, CodeProjectEval) are cited with their original papers; DevEval is released under the Apache-2.0 license. We use the OpenHands SDK (MIT license) and Infomap (GPL-3.0 license), and access GPT-5 Mini, Claude Sonnet 4.6, and Claude Code under their providers' terms of service.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: The only new asset is our code, released under the Apache-2.0 license with a README covering installation, configuration, and how to run Co-Coder, the baselines, and the evaluation. We release no new datasets or models.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing or human subjects research.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve human subjects research.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are central to the method: they generate the Repository Interface Blueprint (Section 3.1) and perform code generation (Section 3.3). The base models (GPT-5 Mini and Claude Sonnet 4.6) are stated in Section 4.3.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.