REVIBE: QUANTITATIVELY EVALUATING MULTI-AGENT VIBE CODING ON REAL-WORLD SOFTWARE TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Vibe coding is reshaping software development, and multi-agent teams promise to compress delivery time by dividing labor across parallel workers. Yet existing multi-agent benchmarks rely on simulated settings that entangle reasoning with communication, only reward coarse task completion, and ignore the time and monetary costs that matter most in practice. We introduce REVIBE, a benchmark that quantitatively evaluates multi-agent vibe coding on real-world software tasks. REVIBE crosses 10 full-stack projects, each given to agents only as a natural-language requirement document and graded by a weighted 100-point rubric, with 10 explicitly defined *collaboration modes*: complete organizational policies that fix how a four-agent team allocates ownership, concurrency, decision authority, and handoff artifacts, while the model, team size, tools, round budget, and evaluator stay fixed. Two components make the grid runnable: LEGOGENT executes each mode through periodic *sync* communication and a native CI/CD pipeline, and TAGENT discovers and probes each unseen deployment to report a functional score together with latency and prefix-cached token cost. Across 100 project–mode configurations and more than 300 runs, the collaboration mode moves the average best score about as much as the choice among three models does, and far more on a given task: with the project and model fixed, switching the mode shifts the score by up to 40 points and the time to the best round by up to 4.1×. No mode dominates: six modes lie within two points of each other across projects, and the winning mode depends on the project's requirement structure. REVIBE makes these speed, cost, and quality trade-offs measurable and reproducible. Code and benchmark are available at `https://anonymous.4open.science/r/revibe_repo-8828/`.

1 INTRODUCTION

Frontier coding agents have moved from toy function synthesis to long-horizon, execution-grounded software work (Jimenez et al., 2024; Huang et al., 2026; Lu et al., 2026). Yet most evaluations still ask whether one agent produces an acceptable patch, not how a team of agents divides labor, builds a project from scratch, spends time and tokens, and improves after concrete feedback.

Multi-agent systems promise faster delivery by assigning specialized workers to parallel subtasks, but the promise is fragile: poorly bounded communication amplifies semantic drift, heavy orchestration becomes a bottleneck, and self-reported completion drifts from runnable behavior. Systems such as ChatDev and MetaGPT each hard-code one staged workflow (Qian et al., 2024; Hong et al., 2024), and multi-agent benchmarks expose communication patterns (Zhu et al., 2025) but are often simulated, detached from continuous integration and delivery (CI/CD), and too coarse to explain why one way of organizing a team succeeds where another fails on the same project and model.

We introduce REVIBE, a benchmark for quantitatively evaluating multi-agent vibe coding on real-world software tasks. By *vibe coding* we mean building software by delegating implementation to LLM agents through natural-language requirements, with no human writing or reviewing code; in REVIBE the loop is fully autonomous and the only feedback is the evaluator's report. *Real-world* tasks are university software-engineering capstones, which teams of three to four students

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

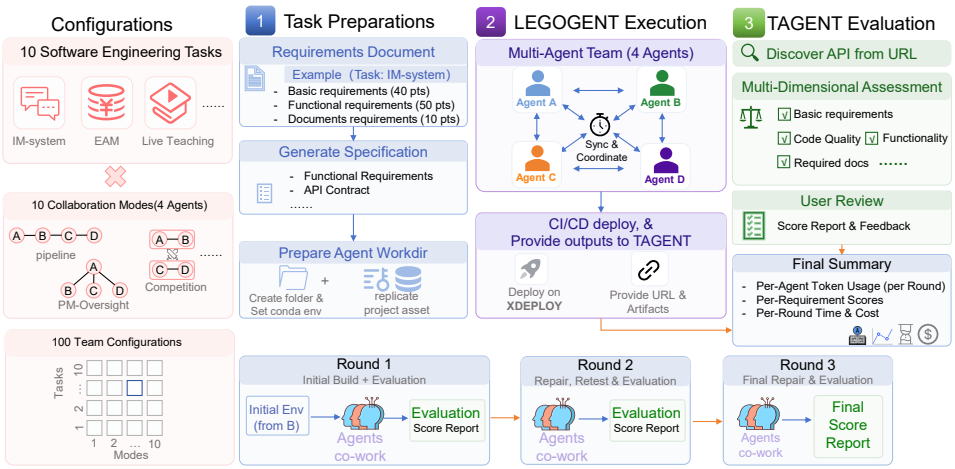


Figure 1: REVIBE overview. REVIBE builds a web app from a requirement document without API specifications, deploys it, and evaluates functionality, time, and cost over up to three rounds.

build iteratively against a weighted requirement document with teaching-assistant (TA) feedback. *From scratch* means that agents start from an empty repository and that document alone, with no API specification, schema, or starter code; the API specification in Figure 1 is derived from the document inside LEGOGENT, so every interface is designed within the run. The object of study is the *collaboration mode*: a complete organizational policy that fixes, for a four-agent team, who owns which artifact, which agents run concurrently, who resolves conflicts, and which handoff artifact must exist before a round is accepted (Table 2). We compare modes as whole packages, the unit a practitioner adopts, with the model, team size, tools, budget, and evaluator held fixed.

REVIBE combines four properties that prior coding-agent evaluations lack in combination: **diverse collaboration**, running the same project under ten modes, from feature squads and pipeline handoff to open-source (pull-request) review and competing teams; **deployment-grounded evaluation** through live user interface (UI), application programming interface (API), and code checks rather than final-text inspection; **iterative feedback** over up to three rounds, exposing convergence and regression; and **first-class cost metrics**, logging wall-clock latency and prefix-cached token usage alongside the functional score. Our contributions are:

- REVIBE, a benchmark crossing 10 real-world projects with 10 explicitly defined collaboration modes (100 configurations), released with project specifications, weighted rubrics, mode definitions, source code, metrics, and analysis scripts;
- two enabling components: LEGOGENT, a multi-agent runtime with bounded synchronization, peer messaging, CI/CD deployment, and behavioral completion checks; and TAGENT, a document-driven grader that probes the UI, API, and code of unseen implementations and returns weighted, actionable feedback;
- an empirical study over 300+ runs showing that the collaboration mode, model, and project jointly shape the speed–cost–quality trade-off, with no mode best everywhere.

2 BACKGROUND AND RELATED WORK

2.1 CODING AGENTS

Coding-agent evaluation has moved from function-level generation (HumanEval, MBPP) to multi-file context and real GitHub issues (RepoBench, SWE-bench) (Chen et al., 2021; Austin et al., 2021; Liu et al., 2024; Jimenez et al., 2024). Agent systems such as SWE-agent, Agentless, AutoCodeRover, and OpenHands navigate repositories, run tools, and repair defects (Yang et al., 2024; Xia et al., 2025; Zhang et al., 2024; Wang et al., 2025), and DeepSWE and FrontierCode push toward longer-horizon engineering (Huang et al., 2026; Lu et al., 2026). RealDevWorld asks agents

Table 1: Comparison of REVIBE against existing benchmarks. REVIBE is the first benchmark that examines the impact of multi-agent coordination on from-scratch coding. ✓,✓̸, and ✗denote full, partial, and no support.

Work	Code	From Scratch	Multi Agent	Collab. Focus	Cost Aware
SWE-bench	✓	✗	✗	✗	✗
MultiAgentBench	✗	✗	✓	✓	✓̸
CloudDevBench	✓	✓̸	✓	✗	✗
DeepSWE	✓	✗	✗	✗	✗
FrontierCode	✓	✗	✗	✗	✗
TheAgentCompany	✓̸	✗	✓	✗	✗
REVIBE (ours)	✓	✓	✓	✓	✓

to generate production-ready repositories from scratch and judges them by clicking through their graphical interfaces with an agent-as-a-judge (Bian et al., 2025). These benchmarks still evaluate a single agent's patch, repair, or artifact. They do not ask whether a team of agents can start from requirements, divide work, deploy a full-stack system, and improve after feedback while spending time and tokens; REVIBE measures this whole delivery loop rather than a model's coding skill alone.

2.2 MULTI-AGENT BENCHMARKS

ChatDev and MetaGPT generate software from requirements through role-specialized, staged collaboration (Qian et al., 2024; Hong et al., 2024), but each fixes one waterfall-style workflow (close to our pipeline and PM-oversight modes) and is evaluated as a system rather than used to compare coordination policies. REVIBE therefore does not claim multi-agent from-scratch coding as new; its contribution is a controlled, deployment-grounded comparison of collaboration modes. AgentsNet and MultiAgentBench study collaboration and competition across communication structures, but on tasks more abstract than deployed software, so they cannot show whether a policy yields a runnable application (Grötschla et al., 2025; Zhu et al., 2025). CloudDevBench evaluates the multi-agent assistant CloudMAS on 50 repository-development cases (Guan, 2026), but with a fixed collaboration mode and detailed specifications with pre-defined APIs, so it is only partially from scratch. TheAgentCompany broadens the task mix but does not control collaboration policy inside a software-delivery loop (Xu et al., 2025). None of these designs separates weak coding ability from duplicated work, stale context, bad handoffs, deployment friction, or inefficient evaluation.

REVIBE fills this gap by combining realistic from-scratch projects, multiple collaboration modes, deployment-grounded scoring, and cost-aware feedback in one controlled setup (Table 1). It holds the project, model, deployment path, and rubric fixed while varying the mode, so a low score can be read together with the run's time, tokens, deployment behavior, and feedback trajectory. Because each artifact typically requires 2000+ lines of newly generated Python code, REVIBE is also less sensitive to data contamination than benchmarks with fixed reference solutions.

3 METHODOLOGY

3.1 BENCHMARK FRAMEWORK OVERVIEW

REVIBE measures how agent teams build deployable software in three stages (Figure 1).

Data foundation. We curate 10 full-scale projects from a corpus of university software-engineering capstones developed since 2018, covering real-time messaging, enterprise asset management, crowdsourcing, requirement tracking, AI image restoration, e-commerce, influencer analytics, role-based access control, news search, and online live teaching. Each requirement document decomposes into six to eight modules and 30–45 weighted line items, normalized to a fixed 100-point rubric.

Execution and metrics. LEGOGENT instantiates a mode, launches agents with role-specific prompts, monitors progress, and sends the repository to XDEPLOY, which builds and stages it

Table 2: The ten collaboration modes as instances of the mode template; later tables use the abbreviations in parentheses. All modes use four agents with the same model, tools, round budget, time cap, and feedback. Only competitive mode builds redundant implementations, and only pipeline is nearly serial. Appendix C details each mode.

Mode	Ownership map	Activation schedule	Decision rule	Required handoff artifact
Serialized handoff (planned coordination)				
Feature squad (Feat.)	One vertical module (UI+API+state) per agent.	All four in parallel.	Module owner decides.	A working slice per module.
Layer specialists (Layer)	Two frontend, two backend agents.	Parallel; sync at the layer boundary.	The API contract arbitrates.	API contract and both layers.
Pipeline (Pipe.)	Architect → backend → frontend → QA.	Nearly serial, stage by stage.	Upstream output is frozen.	A clean stage artifact.
QA-first (QA)	QA owns tests; others implement.	Tests first, then code.	Acceptance tests are the truth.	An executable acceptance suite.
Rotation (Rot.)	Roles rotate every round.	All active; roles permuted.	Current role governs.	A handoff note for the next role.
Parallel contention (independent-then-reconcile)				
PM oversight (PM)	Non-coding manager; three developers.	Manager gates each round.	Manager adjudicates.	Reviewed plan and sign-off.
Swarming (Swarm)	No owner; tasks claimed from a list.	All four in parallel.	First claimant owns a task.	A claimed-task ledger.
Open-source review (OSS)	Maintainer, two contributors, reviewer.	Parallel proposals, then review.	Changes land on approval.	Approved change sets.
Adversarial (Adv.)	Builders vs. a breaker.	Parallel build and attack.	Reproduced defects veto.	A defect report and its fix.
Competitive (Comp.)	Two redundant two-agent teams.	Independent, in parallel.	Higher score wins.	Two deployable candidates.

through a native CI/CD path with GitLab and SonarQube (Campbell & Papapetrou, 2013), so deployment failure is part of the measured outcome. TAGENT scores the staged artifact from 0 to 100, and LEGOGENT records wall-clock time and tokens, joining quality, speed, and cost per round.

Mode definition. A collaboration mode is a complete organizational policy for a four-agent team, specified by four template objects (Table 2): an *ownership map* (roles, and who owns which part of the repository), an *activation schedule* (which agents run concurrently), a *decision rule* (who resolves conflicts and may reject a completion claim), and a *required handoff artifact* (what must exist before a round is accepted); TAGENT feedback goes to whoever the ownership map makes responsible. The objects co-vary by construction, because each mode is one coherent way to organize work: the ten modes mirror practices software teams actually adopt, such as feature squads, a layered front/back-end split, staged (waterfall) handoff, test-driven acceptance, and pull-request review (Table 2); each is taken as a package a team commits to, not a menu of independent knobs. Pulling the objects apart breaks the practice: a pipeline whose stages run concurrently is no longer a pipeline, and competitive mode cannot exist without redundant implementations. Differences in role allocation, parallelism, decision authority, and redundancy are therefore native properties of the modes, not confounds added by the benchmark, and a result about a mode transfers to a practice a team could adopt.

Controlled factors. For a given project and model, every run uses the same backbone model for all agents, four agents, the same requirement document, tools, and prepared workspace, the same sync period and mailbox, the same three-round budget and per-round time cap, the same deployment path, and the same TAGENT rubric and feedback format. We impose no token budget: tokens, cost, and time are outcomes of a mode, not controlled inputs. We therefore compare modes as whole policies, which is what separates a weak model from a weak way of organizing it, and do not attribute an effect to any single component; isolating components (e.g., varying concurrency at fixed ownership) needs ablations that the template makes expressible and we leave to future work.

3.2 MULTI-AGENT EXECUTION VIA LEGOGENT

LEGOGENT turns a collaboration mode into an executable team. Each agent runs as an isolated, resumable process with file and shell tools in a shared workspace, and framework-side validators check ports, HTTP behavior, and artifacts before accepting a completion claim, so a handoff counts only as a validated artifact (a schema, route contract, test report, or deployable repository state). Rather than inferring a fixed dependency graph, LEGOGENT runs a periodic sync loop (Figure 2):

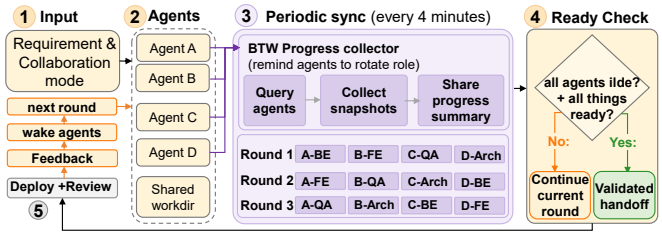


Figure 2: An illustration of LEGOGENT execution under the rotation collaboration mode, where the agents switch their roles in each round. BE and FE mean backend and frontend.

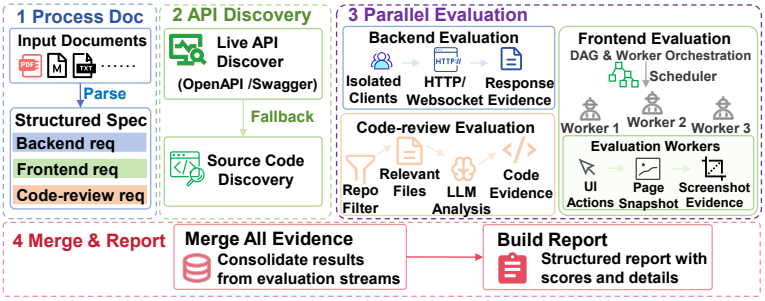


Figure 3: TAGENT parses a requirement document into weighted checks, discovers the implementation, and merges parallel UI, API, and code evidence into a score.

agents work concurrently while a progress collector polls them and broadcasts a shared snapshot, and a completion gate accepts the round once all agents are idle and the mode's required artifacts exist.

Coordination uses two bounded channels in every mode: a passive *sync* round every four minutes, where each agent reports progress, test evidence, and blockers and the runtime broadcasts a consolidated snapshot; and an active mailbox for targeted peer questions between syncs. Parallel workers thus share a view without collapsing into one context, and communication overhead shows up in the time and token logs.

Modes as templates. A mode fills in the four objects of Table 2, while tools, validators, and channels stay shared, and LEGOGENT routes each failure-report item to its responsible owner. Because all modes write into one repository and pass the same deployment validator, the benchmark can see whether parallelism reduces wall-clock time or merely creates duplicate context and merge repair.

3.3 AUTOMATED EVALUATION VIA TAGENT

TAGENT acts as an automated TA for heterogeneous agent-generated projects (Figure 3). It parses each requirement document into a structured rubric of weighted scenarios with pass/partial/fail criteria, receives the staged URL and repository, and discovers the implementation surface before probing it; like agent-as-a-judge GUI testing (Bian et al., 2025), it exercises the running application rather than trusting self-reports. Each requirement is routed to one or more of three evaluator classes. **UI** checks use Playwright to drive the live site, read the accessibility tree or DOM summary, and judge captured states against the rubric. **API** checks probe OpenAPI/Swagger routes when available, otherwise reconstruct them from source, and issue authenticated HTTP sequences. **Code** checks use directory filtering, targeted search, and LLM inspection to verify constraints that are not externally visible, such as password hashing or CI/CD configuration. The rubric is compiled once per project, checked by human experts (none required changes), and reused unchanged for every run of that project, so all runs are judged against the same checks; as an editable file, it can also be extended with new checks. Individual verdicts still involve LLM judgment: an LLM (DeepSeek v4 Pro) drives the browser and judges observed UI states, declares blocking dependencies among UI checks, and inspects code. Stateful UI checks (e.g., register, log in, befriend, then enter a group

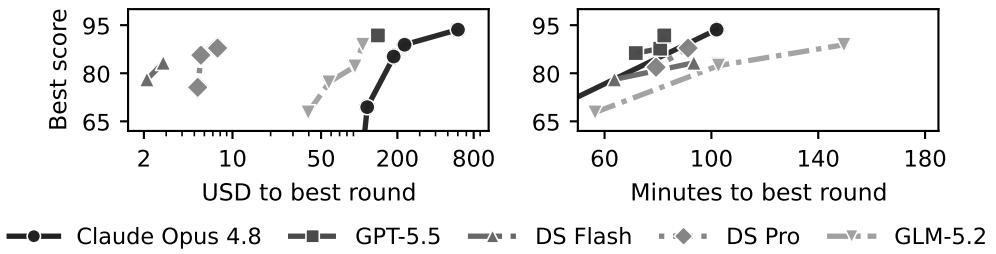


Figure 4: Pareto-frontier modes on instant messaging: best score vs. USD cost (log scale) and minutes to the best round. Each marker is a frontier mode of one model.

chat) are scheduled over this dependency graph so that shared setup is built once per browser worker and unreachable prerequisites cascade-skip their dependents (Appendix F).

Conservative parallelism. TAGENT parallelizes probes only after discovering dependencies, since a worker that deletes a friend or changes profile data can poison another check on the same account (Appendix F). The same principle motivates the benchmark itself: parallel agents help when their side effects are isolated and hurt when they touch shared state without a merge protocol.

Feedback as repair guidance. A final scalar does not tell agents where a dependency chain broke, so TAGENT emits item-level evidence: the observed UI state, API response, code location, failed prerequisite, and affected rubric weight. LEGOGENT returns it as a ranked repair agenda that puts high-weight root causes before low-weight symptoms: fixing registration unlocks downstream friend, chat, and group checks, while a cosmetic chat bubble cannot compensate for a missing message API. Weighted scores sum to a 100-point Functional Completion Score. As the report is both repair signal and next-round grade, refinement gains are gains on this rubric (Section 5).

4 EVALUATION

Isolated port ranges keep concurrent deployments, tests, and evaluations apart (Appendix J). A *configuration* is a project–mode pair, and a *run* executes one configuration with one model for up to three rounds; a round builds or patches the project, deploys it, runs TAGENT, and returns scored feedback. The full grid runs all 100 configurations once with DeepSeek v4 Flash, DeepSeek v4 Pro, and GLM-5.2 (300 runs, hence 600 round-to-round transitions); Claude Opus 4.8, GPT-5.5, and Qwen3.6-Flash run the ten instant-messaging configurations. On instant messaging, every model runs each configuration three times to measure run-to-run variance. We report the Functional Completion Score (0–100) with wall-clock minutes, USD cost, and raw tokens to the best round; cost applies provider prices to new-input, cache-read, cache-create, and output tokens (Appendix H) (Anthropic, 2026b; OpenAI, 2026b; DeepSeek, 2026; Z.AI, 2026b; Alibaba Cloud, 2026).

4.1 RESULTS BY MODEL

Table 3 reports instant messaging under all ten modes and five models as a quality–latency–budget frontier rather than a leaderboard. Claude Opus 4.8 (Anthropic, 2026a) tops this project, GPT-5.5 (OpenAI, 2026a) stays close at much lower cost, DeepSeek v4 Pro (DeepSeek-AI et al., 2026) is the stronger full-grid baseline, and GLM-5.2 (Z.AI, 2026a) has the highest full-grid average but is by far the slowest. Qwen3.6-Flash (Qwen Team, 2026) scores 0 in every mode even though its runs last 39–270 minutes and cost $9.04–$61.25. We report this negative result in the text rather than as an all-zero block: a model can spend a full budget without earning any rubric credit.

Single-agent reference. To test whether organizing a model as a team helps, we run one Claude Opus 4.8 agent under the teams' conditions (last row of Table 3): the same requirement document, tools, three-round budget, time cap, deployment path, and TAGENT feedback. It reaches only 64, against 94 for the best team (feature squad) and 84.4 over the ten modes; all ten collaboration modes beat it, and the other four models' single-agent runs appear in the last row of Table 3. For a capable model a team therefore pays off, and the question is which mode, not whether to form one.

Table 3: Metrics of the instant-messaging project. R1/R2/R3 scores with time to the best round (T, minutes), cost (C, USD), and total tokens (Tok, millions); bold marks the best round in each cell and the best-round metrics of the row-best model. The last row runs one single agent per model (no team) under the same conditions. Qwen3.6-Flash (0 in every mode) is omitted.

Mode	Claude Opus 4.8				GPT-5.5				DeepSeek v4 Flash				DeepSeek v4 Pro				GLM-5.2			
	Score	T	C	Tok	Score	T	C	Tok	Score	T	C	Tok	Score	T	C	Tok	Score	T	C	Tok
Feat.	73/88/**94**	**111**	**617**	**112**	76/**86**/81	88	146	29	0/53/**68**	96	3	448	48/60/**82**	81	7	578	38/**68**/64	55	43	119
Layer	68/74/**85**	93	194	169	72/88/**92**	**84**	**149**	**29**	57/22/**66**	117	3	655	32/70/**86**	92	5	390	46/66/**73**	138	77	216
Pipe.	61/78/**89**	137	208	167	68/81/**90**	**101**	**155**	**33**	0/37/**78**	61	2	329	44/79/**88**	102	8	758	66/26/**73**	157	87	265
PM	67/86/**91**	**179**	**914**	**152**	74/80/**85**	122	278	55	51/**83**/77	87	3	535	73/**76**/71	97	5	338	50/72/**81**	151	155	384
QA	19/19/**69**	94	127	135	78/81/**90**	**90**	**233**	**50**	66/72/**83**	99	3	454	65/75/**88**	83	8	624	55/**59**/58	99	66	159
Swarm	70/81/**85**	153	1077	234	78/82/**86**	**73**	**246**	**47**	0/62/**70**	127	3	548	45/72/**85**	95	10	702	0/76/**82**	104	111	291
Rot.	70/79/**84**	134	943	187	59/**88**/84	**81**	**287**	**61**	43/42/**52**	105	3	488	61/76/**85**	105	10	711	40/67/**77**	165	65	166
OSS	0/70/**83**	261	173	35	52/80/**88**	**292**	**274**	**61**	43/68/**71**	63	2	507	0/30/**48**	137	11	679	0/65/**66**	168	92	302
Adv.	57/76/**85**	176	213	32	58/90/**90**	**71**	**141**	**42**	39/67/**80**	114	4	620	64/63/**72**	98	8	502	57/78/**89**	167	94	253
Comp.	58/71/**78**	177	201	33	0/72/**85**	**157**	**261**	**46**	0/72/**82**	119	4	655	63/70/**78**	132	11	635	0/69/**82**	168	95	237
Single	40/37/**64**	78	92	5	24/**40**/40	22	7	1	53/52/**53**	46	1	41	33/35/**67**	111	3	3	73/76/**82**	75	9	10

Table 4: Average metrics over all 10 projects. Score gives the per-round means (R1/R2/R3); T, C, and Tok are defined as in Table 3. Bold marks each model's top mode by R3 mean. Best is the mean best-round score over the three models, which can exceed R3 means because some runs regress.

Mode	DeepSeek v4 Flash				DeepSeek v4 Pro				GLM-5.2				Best
	Score	T	C	Tok	Score	T	C	Tok	Score	T	C	Tok	
Feat.	23.6/51.9/71.9	79	3	460	35.2/66.6/82.5	78	8	542	35.4/65.8/83.4	231	46	128	82.2
Layer	20.0/48.7/67.2	101	3	429	40.0/67.0/80.5	94	8	473	38.5/60.7/76.6	254	48	135	77.3
Pipe.	34.7/58.0/72.2	78	2	393	57.7/77.5/83.7	108	7	509	57.6/76.0/**85.8**	**248**	**57**	**177**	81.9
PM	29.3/61.1/76.2	99	3	444	40.9/71.9/78.8	93	6	432	30.4/64.8/82.7	254	73	201	81.4
QA	48.7/67.5/**80.8**	**96**	**3**	**473**	49.8/74.5/84.4	93	7	539	53.8/69.4/80.1	234	55	181	83.1
Swarm	24.1/45.7/64.3	85	3	523	59.7/77.4/82.9	76	8	565	34.8/75.6/84.2	247	63	169	79.5
Rot.	34.5/66.8/74.8	82	3	483	48.8/72.2/**85.4**	**100**	**9**	**575**	55.6/74.3/85.3	258	78	189	83.2
OSS	19.9/49.2/72.9	77	3	434	35.9/56.8/69.1	93	9	615	8.4/39.2/67.5	248	59	173	73.5
Adv.	27.8/45.8/68.6	95	3	492	40.0/65.5/71.1	88	8	487	37.7/68.6/80.9	235	95	189	77.3
Comp.	38.5/68.0/79.9	110	4	578	43.9/60.7/78.6	107	8	542	7.7/59.7/82.7	269	65	183	81.3

Figure 4 makes the comparison cost-aware. Opus 4.8 reaches the highest score (94, feature squad) but needs 111 minutes and about $617, while GPT-5.5 is close (92, layer specialists) at 84 minutes and $149; on the full grid, GLM-5.2 edges DeepSeek v4 Pro at 2.7× the time and 8× the cost.

4.2 Results by Collaboration Mode

Table 4 tests whether modes transfer beyond instant messaging by averaging over the 10 projects. We read small gaps as ties: each model's top mode leads its runner-up by at most 1.1 points.

Figure 5 profiles each mode as an execution signature on ten 0–10 axes (higher is better): seven are measured from the logs (best and R1 score, R1-to-R3 gain, R2-to-R3 stability, cross-run consistency, inverse time, inverse tokens) and three are rated from each mode's protocol in Table 2 (ownership clarity, handoff inspectability, useful parallelism); Appendix D defines each axis.

The hardest projects favor modes that turn long requirement chains into explicit checks. Live teaching and image restoration are the two lowest-scoring projects (average best 67.8 and 69.6) as many user-visible functions must compose. QA-first scores near the top on image restoration (78.1, +8.5 over the project average) and above average on live teaching (69.7), consistent with its protocol of turning requirements into acceptance targets before implementation spreads across layers.

Ownership helps most when several agents share one requirement. Feature squad is a top-three mode on five projects (e.g., 80.5 on enterprise asset management and 92.1 on e-commerce). A chain such as approving a user or publishing an exhibit lives in neither frontend nor backend alone; when agents split only by layer, no one owns the scenario after TAGENT reports it unmet, whereas feature squad gives one agent the whole vertical path.

Large repair gains can hide weak first rounds. QA-first and pipeline start higher, with tests or staged deliverables exposed early. Competitive and open-source review post large R1-to-R3 gains,

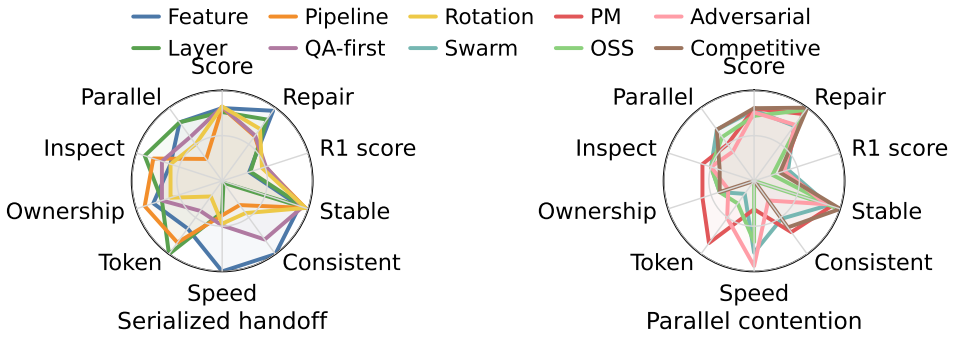


Figure 5: Collaboration-mode radar on ten normalized axes; higher is better (Appendix D).

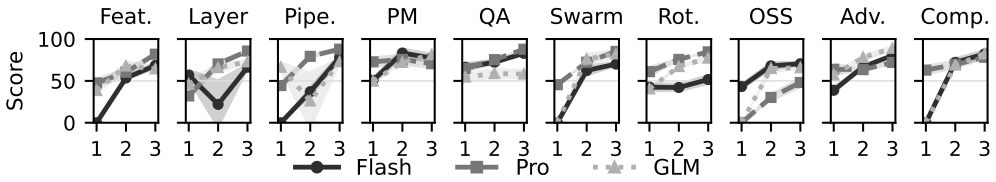


Figure 6: Per-round IM scores (R1–R3); shaded bands show standard deviation over three trials.

but their weaker score or cost shows that a big gain can mean a weak first round, not a better policy. Figure 6 plots these per-round trajectories with ± 1 s.d. bands over trials; the best-round standard deviation averages 2.6 points across configurations (Figure 7 in Appendix B), far below the mode-to-mode spread, so differences between modes are real, not trial noise.

4.3 Refinement under TAgent Feedback

Feedback-conditioned rounds usually improve. Of the 600 full-grid transitions, 82.0% improve, and 94.7% of runs finish R3 above R1, so R1 is one-shot and R3 feedback-conditioned. Because every round gets feedback, this does not separate item-level feedback from extra rounds (Section 5).

Three rounds capture most of the gain. The largest jump is R1-to-R2, when one-shot drafts first meet TAgent's report; by R2-to-R3 the steps are small, with 73.3% gaining under 20 points (median +7.8). Further rounds mostly add tail recoveries (a late round rescuing a run that stalled early), most often for GLM-5.2, whose capped first rounds leave the most to repair. Three rounds thus sit near the knee of the refinement curve.

Regression is a distinct failure mode. Two mechanisms explain the low stability axis in Figure 5 and the round-level drops in Table 3. First, the per-round time cap can truncate implementation before the repository is coherent, most visibly for GLM-5.2, many of whose first rounds hit the 90-minute cap with unfinished handlers or missing routes. Second, a repair can fix one requirement while breaking another: in DeepSeek v4 Pro's PM-oversight run on instant messaging, round 3 fixes profile editing but breaks search, friend requests, messaging, and group flows (80.9 to 66.0).

4.4 A General Failure Taxonomy

Regressions explain lost *stability*, but not why even the best round rarely reaches 100. Labeling every unmet check by root cause yields 1,440 agent-attributable failures (after excluding 80 grader-side errors; Table 10 in Appendix G). **Functional shortfall dominates, not crashes.** Missing features and incomplete logic are 56% of failing checks, and 86% of failed items keep partial credit, so teams ship running but incomplete systems. **One broken prerequisite poisons many checks.** Authentication and prerequisite failures are 23% of failures, and one broken prerequisite auto-skipped up to 14 downstream checks in a run, so a contract broken by one owner silently zeroes another owner's correct work. **Part of the ceiling is a deployment artifact.** Security and transport items (8.5%)

recur in nearly 80% of runs because they depend on integration with the TLS layer of XDEPLOY. The bottleneck is integration and cross-agent contracts, not raw code generation.

4.5 KEY FINDINGS AND INSIGHTS

The mode matters about as much as the model on average, and far more on a given task. Mean best scores span 73.5 (open-source review) to 83.2 (rotation) across modes (Table 4, Best column), comparable to the 75.5–82.4 spread across full-grid models (Table 6). With the project and model fixed the spread is much larger: DeepSeek v4 Pro scores 88 under pipeline but 48 under open-source review on instant messaging. These are whole-mode, not per-component, effects.

No mode dominates, so the mode must fit the project. Six modes (QA-first, rotation, feature squad, pipeline, PM oversight, and competitive) lie within two points of each other across projects, which we do not read as a ranking. The consistent signal is at the bottom: open-source review has the lowest mean best score across projects. Within a project, modes differ by up to 34.2 points (56.8–91.0 on requirement tracking), and the best mode follows the requirement structure (Table 8): QA-first wins requirement-heavy projects, pipeline wins projects with clean stage boundaries, and feature squad wins the hardest realtime project.

The score frontier is not the cost frontier. GLM-5.2 posts the highest full-grid average best score at far higher time and cost than the DeepSeek models, and Flash is cheapest in every Flash-vs-Pro pair despite using more raw tokens in 34 of 100 configurations, so price, not token volume, sets the bill. Within a model, the mode moves cost independently of score: on instant messaging GLM-5.2's costliest mode (PM oversight, $155) scores below its cheaper adversarial mode (81 vs. 89 at $94), and Opus 4.8's swarm mode costs $1077 yet scores nine points below feature squad (94) at $617.

The mode should be a runtime decision. Since no mode is best everywhere, future agentic systems should choose per task when to parallelize, when to serialize, and when to stop a branch before it costs more than its expected gain. Our runs expose the needed signals: the requirement structure suggests the mode (long requirement chains favor up-front acceptance targets, clean stage boundaries favor a pipeline, cross-layer scenarios favor one owner per feature), the round history suggests when to stop (three rounds capture most of the gain, and late repairs can regress), and metric signatures suggest what failed. With REVIBE recording these metrics, it can benchmark such policies.

5 DISCUSSION AND FUTURE WORK

Scale. Each run builds, deploys, and grades a full-stack application over up to three rounds, taking hours and, for frontier models, hundreds of dollars (Table 3). The budget caps REVIBE at ten projects. The scope is nonetheless broad with ten domains from realtime messaging to analytics and AI-backed multimedia (Table 5), each proposed by industry and crossed with ten modes and six models for 100 configurations and more than 300 graded runs. Scaling up is cheap by design, since a project enters only as a requirement document with an automatically compiled rubric, so adding tasks (including repository maintenance or research-style tasks (Xiong et al., 2026; Du et al., 2025)) needs no framework change. We describe rather than predict which project favors which mode, leaving such predictors to future work.

TAGENT as source and judge. Refinement gains (Section 4.3) are gains on TAGENT's rubric and may partly reflect agents learning what it checks. TAGENT reaches a Spearman correlation of 0.87 with TA grades on 97 student projects, on par with the 0.85 expert agreement for GUI-testing judges (Bian et al., 2025); validating it on agent artifacts is a natural next step.

6 CONCLUSION

REVIBE benchmarks 100 project–mode configurations, measuring score, time, token cost, and feedback rounds together with collaboration modes defined as complete organizational policies for a fixed four-agent team. Modern models build substantial software but remain strongly shaped by how they are organized: no mode wins everywhere, and the best mode depends on the project. Mode, model, and stopping policy should therefore be runtime decisions, not fixed defaults.

AI USE STATEMENT

In this work, generative AI models serve only as participants of the benchmark: the models evaluated in Section 4 act as the coding agents that LEGOGENT instantiates, and TAGENT issues LLM calls (DeepSeek v4 Pro) inside its automated grading pipeline (Section 3.3). We did not use generative AI to raise, develop, or refine the main ideas of this work—the research questions, benchmark design, collaboration modes, methodology, and analysis are the authors' own—and used it only to assist with LaTeX formatting and text editing. We have reviewed all AI-assisted work.

ETHICS STATEMENT

All authors have read and adhere to the ICLR Code of Ethics. The ten REVIBE projects are curated from university software-engineering capstones; the released requirement documents and rubrics are task specifications rather than personal data, and human reviewers checked all released materials so that no student work or industry partners' commercial secrets are disclosed. We release no student submissions, grades, or graded repositories (Appendix A). Agent-generated applications can contain security defects—for example, the plaintext password transport surfaced by TAGENT in Figure 8—and should not be deployed without human review.

REPRODUCIBILITY STATEMENT

Section 3 and Appendices B–J specify the ten projects and their weighted 100-point rubrics, every collaboration mode, the LEGOGENT runtime and the TAGENT grading protocol, the token and cost fields behind every budget figure, and the compute environment. The anonymous repository at `https://anonymous.4open.science/r/revibe_repo-8828/` provides the source code, the requirement documents and compiled rubrics, the two bundled datasets, the price tables, and the scripts that regenerate every table and figure from the aggregated metrics (Appendix A).

REFERENCES

Alibaba Cloud. Alibaba cloud model studio: Model inference pricing. `https://www.alibabacloud.com/help/en/model-studio/model-pricing`, 2026. Accessed: 2026-07-17.

Anthropic. Introducing claude opus 4.8. `https://www.anthropic.com/news/claude-opus-4-8`, 2026a. Accessed: 2026-07-20.

Anthropic. Claude api pricing. `https://platform.claude.com/docs/en/about-claude/pricing`, 2026b. Accessed: 2026-07-20.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL `https://arxiv.org/abs/2108.07732`.

Yutong Bian, Xianhao Lin, Yupeng Xie, Tianyang Liu, Mingchen Zhuge, Siyuan Lu, Haoming Tang, Jinlin Wang, Jiayi Zhang, Jiaqi Chen, Xiangru Tang, Yongxin Ni, Sirui Hong, and Chenglin Wu. You don't know until you click: Automated GUI testing for production-ready software evaluation, 2025. URL `https://arxiv.org/abs/2508.14104`.

G Ann Campbell and Patroklos P Papapetrou. *SonarQube in action*. Manning Publications Co., 2013.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec

Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

DeepSeek. Models and pricing. https://api-docs.deepseek.com/quick_start/pricing/, 2026. Accessed: 2026-07-18.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Xiaorui Wang, and Zhendong Mao. Deepresearch bench: A comprehensive benchmark for deep research agents, 2025. URL https://arxiv.org/abs/2506.11763.

Florian Grötschla, Luis Müller, Jan Tönshoff, Mikhail Galkin, and Bryan Perozzi. Agentsnet: Coordination and collaborative reasoning in multi-agent llms, 2025. URL https://arxiv.org/abs/2507.08616.

Tian Guan. A multi-agent coding assistant for cloud-native development: From requirements to deployable microservices. In *Proceedings of the 2025 6th International Conference on Computer Science and Management Technology*, ICCSMT '25, pp. 1285–1290, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400719981. doi: 10.1145/3795154.3795362. URL https://doi.org/10.1145/3795154.3795362.

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*, 2024.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks, 2026. URL https://arxiv.org/abs/2607.07946.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 47832–47850, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/d191ba4c8923ed8fd8935b7c98658b5f-Paper-Conference.pdf.

Eric Lu, Ben Pan, Deniz Birlikci, Sam Lee, Ray Wang, Rohan Choudhury, Fermi Ma, TC Qin, Carlo Baronio, Silas Alberti, et al. Introducing frontiercode. https://cognition.com/blog/frontier-code, 2026. Accessed: 2026-07-13.

OpenAI. Introducing gpt-5.5. https://openai.com/index/introducing-gpt-5-5/, 2026a. Accessed: 2026-07-20.

OpenAI. Openai api pricing. https://developers.openai.com/api/docs/pricing, 2026b. Accessed: 2026-07-20.

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 *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, 2024.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

Lei Xiong, Kun Luo, Ziyi Xia, Wenbo Zhang, Jin-Ge Yao, Zheng Liu, Jingying Shao, Jianlyu Chen, Hongjin Qian, Xi Yang, Qian Yu, Hao Li, Chen Yue, Xiaan Du, Yuyang Wang, Yesheng Liu, Haiyu Xu, and Zhicheng Dou. Autoresearchbench: Benchmarking ai agents on complex scientific literature discovery, 2026. URL https://arxiv.org/abs/2604.25256.

Frank (Fangzheng) Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. Theagentcompany: Benchmarking llm agents on consequential real world tasks. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*,

volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-0315. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/0d744742f6fac4d1134c019b7cef3c8a-Paper-Datasets_and_Benchmarks_Track.pdf.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Z.AI. Glm-5.2: Built for long-horizon tasks. https://z.ai/blog/glm-5.2, 2026a. Accessed: 2026-07-15.

Z.AI. Z.AI pricing. https://docs.z.ai/guides/overview/pricing, 2026b. Accessed: 2026-07-15.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024.

Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Daisy Zhe Wang, Zhenhailong Wang, Cheng Qian, Robert Tang, Heng Ji, et al. Multiagentbench: Evaluating the collaboration and competition of llm agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8580–8622, 2025.

Table 5: The ten projects. "Scope" names the dominant difficulty; e.g. online live teaching has 24 functional items across six groups and AI image restoration chains upload/storage, ordered enhancement calls, progress, before/after comparison, exhibit layout, search, and moderation. Every project ships a hierarchical requirement document that compiles to a weighted rubric (P00 yields 56 checks: 7 system, 46 functional, 3 documentation).

ID	Project (domain)	Scope
P00	Instant messaging	56 checks
P01	Enterprise asset management	long-path
P02	Crowdsourcing platform	multi-role
P03	Requirement tracking	workflow
P04	AI image restoration	multimedia
P05	E-commerce trading	transactional
P06	Influencer analytics	data/vis
P07	Permission platform	auth-heavy
P08	News search	retrieval
P09	Online live teaching	realtime A/V

APPENDIX

Appendix A lists the released assets and how to regenerate the paper's numbers. Appendix B details each project and its rubric, and Appendix C describes each of the ten modes and its per-project results, and Appendix D defines the radar axes. The LEGOGENT synchronization runtime is described in Appendix E and the TAGENT grading protocol in Appendix F; Appendix G details the failure taxonomy, Appendix H defines the token and cost fields behind the budget metric, Appendix I follows one run across its refinement rounds, and Appendix J describes the compute environment. All examples are taken verbatim from real benchmark runs.

A BENCHMARK ASSETS AND REPRODUCIBILITY

REVIBE is a 10×10 grid—ten authentic full-stack projects crossed with ten collaboration modes—executed over six models: three run all 100 configurations (the 300-run full grid), and three more join them on the ten instant-messaging configurations. We release the material needed to reproduce every reported number in an anonymous repository, `https://anonymous.4open.science/r/revibe_repo-8828/`: the LEGOGENT and TAGENT source code, the ten requirement documents and their compiled rubrics, the two bundled datasets, the price tables, and the aggregation and figure scripts. To keep the release compact and to avoid redistributing model-generated source, the raw inter-agent handoff logs and the graded project repositories are not included.

Reproducing the tables and figures. The released scripts rebuild every reported number from the aggregated metrics. Joining the per-round records with the price table yields the best- and final-round columns of the main results tables directly, and `plot_eval_figures.py` renders the cost-frontier, per-round-trajectory, and collaboration-radar figures from the same metrics file. The price tables ship with the release so the rates in Table 11 can be adjusted and the grid recomputed.

B THE TEN PROJECTS

REVIBE draws its ten projects from real software-engineering course capstones spanning messaging, CRUD-and-workflow business systems, transactional commerce, retrieval, analytics, access control, and realtime multimedia. Each ships a hierarchical requirement document that compiles to a weighted 100-point rubric (typically 35–40 system, 50–55 functional, 10 documentation). Table 5 lists them; only the three full-grid models (DeepSeek v4 Flash/Pro, GLM-5.2) run all ten, while Claude Opus 4.8, GPT-5.5, and Qwen3.6-Flash run only the instant-messaging reference project P00 (Table 3).

Each project below is annotated with its full-grid difficulty – the mean best score over the three full-grid models (DeepSeek v4 Flash/Pro and GLM-5.2) across the ten collaboration modes – and the collaboration mode that tops it.

- **P00 Instant messaging** (mean 77.7; best PM oversight 82.9). An iMessage-style app for private and group chat with a session-list/conversation-pane UI, serving ordinary users and group owners/admins. Its 56-check rubric splits 40 system (usability, security, sub-200 ms realtime delivery, data integrity) / 50 functional / 10 docs, with friend- and group-prerequisite chains; it is the reference project run by every model.
- **P01 Enterprise asset management** (mean 70.6; best pipeline 80.8; GLM-5.2 leads at 75.5). Full asset lifecycle – intake, lending, transfer, retirement, maintenance – with multi-tenant business-entity isolation, hierarchical category trees, and approval workflows across four role tiers. A single operation must mutate persistent state, enforce tenant isolation, and append an audit log, so the layer split collapses here (59.9, its worst project).
- **P02 Crowdsourcing platform** (mean 75.2; best PM oversight 80.4). A data-labeling marketplace spanning publisher, worker, and reviewer roles over a long chain: create/configure task $\rightarrow$ upload data $\rightarrow$ dispatch $\rightarrow$ claim $\rightarrow$ annotate $\rightarrow$ review $\rightarrow$ settle $\rightarrow$ export. Anti-cheating and credit scoring ride on top of every stage, keeping the whole pipeline coupled.
- **P03 Requirement tracking** (mean 78.0; best QA-first 91.0; DeepSeek v4 Pro leads at 85.8). A Jira-like tool that decomposes initial requirements into functional requirements, plans iterations, and analyzes Git-commit traceability across several services. QA-first dominates because its acceptance targets map directly onto the traceability rubric.
- **P04 AI image restoration** (mean 69.6; best rotation 78.9; GLM-5.2 leads at 82.6). The second-hardest project: museum photo restoration via online AI APIs plus curated exhibition rooms with before/after comparison, browse/search, and likes/comments, serving curators and visitors. The multimedia chain – upload/storage $\rightarrow$ ordered enhancement calls $\rightarrow$ progress $\rightarrow$ comparison $\rightarrow$ exhibit – punishes weak models (Flash 59.4) and rewards GLM-5.2's long-horizon planning (82.6).
- **P05 E-commerce trading** (mean 89.4; best pipeline 94.9). A self-operated marketplace with buyer-side browse, balance top-up, and order flows plus merchant-side catalog, discount, and order management. Transactional consistency with clean stage boundaries makes the serial pipeline strongest.
- **P06 Influencer analytics** (mean 89.6; best competitive 94.7). A creator dashboard over a bundled dataset of per-day view/like/comment deltas, offering overview, per-work analysis, fan-activity trends, and background refresh. Being largely read-only over a fixed dataset places it in the top scoring tier.
- **P07 Permission platform** (mean 90.4; best pipeline 95.5; GLM-5.2 leads at 93.2). An RBAC platform covering apps, permission types, resources, roles, user authorization, and an external permission-query API with tenant isolation. Its clean split between a configuration UI and an auth API lets serialized handoff modes tie near the top (pipeline 95.5, layer specialists 94.6).
- **P08 News search** (mean 92.0; best QA-first 97.0; DeepSeek v4 Pro leads at 95.8). The easiest project: a full-text search engine – crawl, index, retrieve, rank, present – plus user interest tags. The well-bounded retrieval pipeline yields the highest scores in the grid.
- **P09 Online live teaching** (mean 67.8; best feature squad 75.1). The hardest project: a WebRTC live classroom for teacher, TA, and student roles with streaming, teaching resources, room management, in-class chat, and quizzes – 24 functional items across six groups. Only vertical ownership (feature squad) keeps the composed realtime features coherent.

Difficulty tracks requirement composition, not raw feature count. The two hardest projects – live teaching (67.8) and image restoration (69.6) – together with enterprise asset management (70.6) force many user-visible functions to compose over shared realtime or transactional state, whereas the easiest – news search (92.0), permission (90.4), and analytics (89.6) – decompose into well-bounded, largely independent requirements. Across Table 6, DeepSeek v4 Pro is the steadiest full-grid model, posting the highest score on six of ten projects; GLM-5.2 leads the other four, including the hardest

Table 6: Full-grid best-score results per project. **Flash/Pro/GLM** are each model's mean best score over the ten modes; **Avg** is the mean over the three models; **Best mode** is the top mode on that project with its mean best score; the bottom row averages each column over the ten projects. Claude Opus 4.8, GPT-5.5, and Qwen3.6-Flash run only P00 and appear in Table 3.

ID	Flash	Pro	GLM	Avg	Best mode
P00	76.0	79.5	77.6	**77.7**	PM (82.9)
P01	66.9	69.4	75.5	**70.6**	Pipe (80.8)
P02	71.1	75.8	78.8	**75.2**	PM (80.4)
P03	76.2	85.8	71.9	**78.0**	QA (91.0)
P04	59.4	66.9	82.6	**69.6**	Rot (78.9)
P05	87.0	91.8	89.4	**89.4**	Pipe (94.9)
P06	86.2	92.7	89.9	**89.6**	Comp (94.7)
P07	85.2	92.7	93.2	**90.4**	Pipe (95.5)
P08	87.3	95.8	93.0	**92.0**	QA (97.0)
P09	59.7	72.0	71.8	**67.8**	Feat (75.1)
Mean	75.5	82.3	82.4	**80.0**	–

Table 7: Single-agent baseline: one Claude Opus 4.8 agent per project under the team conditions (best score), against the best team mode on that project (mean best over the three full-grid models). The single agent trails the best team on eight of the ten projects and matches it on only two (e-commerce and live teaching).

	P00	P01	P02	P03	P04	P05	P06	P07	P08	P09
Single agent	64	64	70	73	59	95	92	94	88	75
Best team mode	83	81	80	91	79	95	95	96	97	75

multimedia and access-control projects (P04, P07); and DeepSeek v4 Flash keeps pace on the easy retrieval/CRUD projects but falls 10–25 points behind on multimedia and realtime (P04, P09).

Single-agent baseline. Table 7 runs one Claude Opus 4.8 agent per project under the same conditions as the teams. It trails the best team mode on eight of the ten projects, by up to 20 points where user-visible functions must compose (image restoration, 59 vs. 79; instant messaging, 64 vs. 83), and matches it on only two: e-commerce (P05) and live teaching (P09), the latter so hard that even the best team mode reaches only 75. The single agent is also far less stable: its instant-messaging best score ranges from 49 to 91 across runs, a standard deviation near 19 and an order of magnitude above the team's own 2.6-point run-to-run standard deviation. Organizing a capable model as a team therefore helps on all but the easiest tasks, and the open question is which mode to use, not whether to form a team.

Run-to-run variance. Repeating each instant-messaging configuration across trials and taking the best-round standard deviation (Figure 7) yields an average of 2.6 points, an order of magnitude below the mode-to-mode spread, so the differences in Table 4 reflect real effects rather than trial noise.

C The Ten Collaboration Modes

Every mode is an instance of the same template with four concrete objects—an ownership map, an activation schedule, a decision rule, and a required handoff artifact—and Table 2 in Section 3 gives the ten instances. Teams are four agents in every mode (`dev-1` … `dev-4`, or role names such as architect/backend/frontend/QA for staged modes), and the grouping matches the two radar panels of Figure 5 (serialized handoff vs. parallel contention).

Beyond the template objects in Table 2, the ten modes differ in how they split the repository, who holds decision authority, and what most distinguishes each from its nearest neighbor. These differences come as a package: a mode that changes the ownership map also changes who can act concurrently and who decides, which is why we compare modes as whole policies (Section 3).

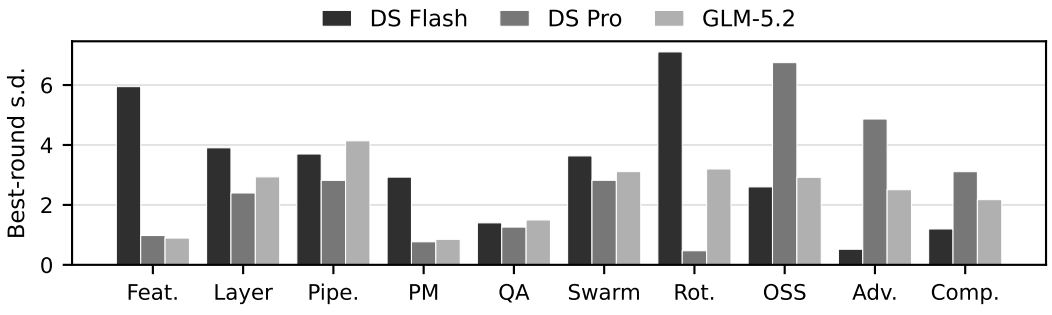


Figure 7: Run-to-run standard deviation of the best-round instant-messaging score, per mode and full-grid model, over the repeated trials behind Figure 6. The overall average is 2.6 points, an order of magnitude below the mode-to-mode spread, so the mode differences are real rather than trial noise.

- **Feature squad** splits the repository *vertically* by user-facing module; four full-stack agents each own one module end to end (UI, API, state, tests) and align only through interface contracts, so decision authority is local to each owner. Unlike swarming it fixes module boundaries rather than leaving them fluid. Its signature is the most balanced across projects, and it is the only mode that tops the hardest project (P09, 75.1).
- **Layer specialists** splits *horizontally*, with two frontend and two backend agents who never cross layers and reconcile through a single API contract that holds the shared authority. It is the only technical-layer split. It does best when that contract is clean (P07, 94.6, near the top) but collapses when one requirement spans layers (P01, 59.9, its worst project).
- **Pipeline** splits work *temporally*: architect → backend → frontend → QA, each stage frozen before the next begins and holding authority only during its window. It is the only nearly serial mode and marks the low-concurrency end of the space. It is strongest on transactional/CRUD projects with clean stage boundaries (P05, 94.9; P01, 80.8).
- **QA-first** is *quality-driven*: QA authors an executable acceptance suite first and holds final say while developers chase the tests. Unlike adversarial testing the QA role is collaborative rather than attacking. It ties for the best cross-project mean, wins the requirement-heavy projects (P08, 97.0; P03, 91.0), and scores near the top on image restoration (P04, 78.1).
- **Rotation** reassigns roles on a *timer*: every round agents swap backend/frontend/QA/architect and continue the predecessor's work via a handoff note, so authority travels with the role rather than the agent. It is the only forced-rotation mode and the heaviest in dialogue turns. It ties QA-first for the best cross-project mean (83.2).
- **PM oversight** is *hierarchical*: a non-coding manager plans, assigns, and gates each round's entry and exit while developers only execute, concentrating authority at the top. Against open-source review's peer approval it is top-down command rather than horizontal consent. It leads on messaging and crowdsourcing (P00, 82.9; P02, 80.4).
- **Swarming** applies *no split*: four identical agents pull tasks from a shared list, the first claimant owning a task until release. It differs from feature squad by having no fixed boundaries and from rotation by not rotating roles. It sits mid-pack (79.5) and spends late rounds reconciling overlap.
- **Open-source review** splits by *permission*: a non-coding maintainer, two contributors, and a reviewer, with change landing only on approval, so authority rests in the approval gate. Against PM oversight it is horizontal peer review rather than top-down command. Its approval and reconciliation overhead make it consistently the weakest mode (73.5).
- **Adversarial** runs *red/blue*: a breaker probes for defects and can veto completion, with disputes settled by reproducing the defect. It is the only mode with a hostile role. It lands mid-pack (77.3) and is strongest where correctness edge cases dominate.
- **Competitive** uses *redundant contention*: two independent teams build the whole spec and the better is selected by score, placing authority in the scorer. It is the only two-team

Table 8: Per-project × per-mode best score, averaged over the three full-grid models (DeepSeek v4 Flash/Pro, GLM-5.2). Columns are grouped as in the radar panels of Figure 5; the right column and bottom row give project and mode means. The best mode per project is discussed in the text.

	Serialized handoff					*Parallel contention*					
ID	**Feat**	**Layer**	**Pipe**	**QA**	**Rot**	**PM**	**Swarm**	**OSS**	**Adv**	**Comp**	**Avg**
P00	75.6	76.9	81.9	78.7	71.7	82.9	81.4	64.0	81.5	82.5	**77.7**
P01	80.5	59.9	80.8	74.2	74.2	70.4	68.5	67.9	59.1	70.4	**70.6**
P02	78.3	67.9	73.3	77.6	77.1	80.4	77.0	70.3	75.3	75.4	**75.2**
P03	73.0	68.9	71.4	91.0	89.2	79.8	88.7	56.8	74.7	86.3	**78.0**
P04	76.5	63.6	60.4	78.1	78.9	69.5	75.2	67.5	60.8	65.9	**69.6**
P05	92.1	86.4	94.9	91.0	93.5	88.8	89.4	81.7	87.6	89.0	**89.4**
P06	90.0	94.5	89.6	94.5	92.9	93.5	81.2	80.8	84.2	94.7	**89.6**
P07	93.3	94.6	95.5	78.9	95.1	93.1	80.8	89.2	91.1	92.1	**90.4**
P08	87.2	96.1	96.8	97.0	96.1	85.4	82.7	87.6	95.0	96.3	**92.0**
P09	75.1	64.1	74.2	69.7	63.0	69.9	69.9	68.7	63.4	60.5	**67.8**
Avg	**82.2**	**77.3**	**81.9**	**83.1**	**83.2**	**81.4**	**79.5**	**73.5**	**77.3**	**81.3**	**80.0**

mode. It wins where a fixed dataset makes selection cheap (P06, 94.7) but is weakest on the hardest realtime project (P09, 60.5).

No mode dominates; fit to the project is the effect. Averaged over projects (Table 8), the modes span only 73.5 (open-source review) to 83.2 (rotation), a range narrow enough that no single policy is globally best. Within a single project, however, the spread is far larger – e.g. 56.8–91.0 on requirement tracking and 59.1–80.5 on enterprise asset management – so the mode matters most when matched to the project's dependency structure. Serialized-handoff modes win seven of the ten projects. Among parallel-contention modes only PM oversight, whose manager gates every round, and competitive, when redundant candidates can be cheaply scored, win any project; swarming, open-source review, and adversarial win none, consistent with a reconciliation cost when parallel changes overlap.

D Collaboration-Mode Radar Axes

Figure 5 summarizes each mode on ten axes, each on a 0–10 scale where higher is better. The seven measured axes are computed per mode from its 30 full-grid runs (ten projects × three models), using each run's best round:

- **Score**: mean best-round score divided by 10.
- **R1 score**: mean first-round score divided by 10.
- **Repair**: mean R1-to-R3 gain divided by 5, clipped to $[0, 10]$.
- **Stable**: $10\times$ the fraction of runs whose R3 score is not below R2.
- **Consistent**: standard deviation of the best-round score across the 30 runs, min–max rescaled across the ten modes and inverted, so the least variable mode scores 10.
- **Speed** and **Token**: mean minutes and mean tokens to the best round, min–max rescaled across modes and inverted.

Because Consistent, Speed, and Token are rescaled across modes, they rank modes against each other rather than on an absolute scale.

The three protocol axes are not measured. We rated each mode from its definition in Table 2: **Ownership** is how clearly a failing requirement maps to one accountable agent, **Inspect** is how explicit and checkable the handoff artifact is, and **Parallel** is how much of the team can work at once without contending for the same files. Table 9 lists the ratings. They describe the protocols, not outcomes, and none of the results in Section 4 depends on them.

Table 9: Protocol ratings on the three unmeasured radar axes (0–10).

	Feat	Layer	Pipe	QA	Rot	PM	Swarm	OSS	Adv	Comp
Ownership	8	7	9	7	6	6	4	4	3	4
Inspect	6	9	8	7	6	6	5	5	5	4
Parallel	8	8	3	6	5	5	7	6	4	7

E LEGOGENT RUNTIME: BOUNDED SYNCHRONIZATION

LEGOGENT does not infer a fixed dependency graph. Instead it runs a periodic sync loop over concurrently working agents, with two bounded communication channels and a runtime-state completion gate.

Passive sync channel. Roughly every four minutes the runtime broadcasts a consolidated *progress-alignment snapshot* to every agent. Each agent contributes four fields—completed work, test evidence, blockers, and next steps—and receives the merged team view. Figure 8 (top) is a translated snapshot from the feature-squad / Claude Opus 4.8 / instant-messaging run (broadcast interval $\approx$244 s). Crucially, the snapshot is explicitly *not* a completion signal.

Completion gate. A round is accepted only when the runtime state shows all agents *idle and stable and* the required artifacts for the mode exist. This decouples "the team stopped talking" from "the team is done," and is what makes idle time and merge repair observable rather than hidden.

Active mailbox. Between sync rounds, an agent may send a targeted question or handoff directly to a peer. In this run (Figure 8), agents opened 52 such peer threads. This is the channel that lets, e.g., a backend owner confirm a route shape with a frontend owner without waiting for the next broadcast.

F TAGENT VERIFICATION METHODOLOGY

TAGENT grades an unseen deployment by discovering its implementation surface, routing each requirement to the appropriate evaluator, and aggregating weighted evidence into a 100-point Functional Completion Score.

Step 1: Rubric compilation. Each raw requirement document is parsed into a structured rubric of weighted scenarios with pass/partial/fail criteria. For P00 this yields 56 checks across three categories (7 system-level, 46 functional, 3 documentation).

Step 2: Surface discovery. Given only the staged URL and repository, TAGENT reconstructs the route surface (from OpenAPI when present, else from source). Each entry records method, path, parameters, and whether authentication is required (the discovery block of Figure 8), so probing uses the real contract rather than a guessed one.

Step 3: Three evaluator classes.

- **UI** (`frontend/`): Playwright drives the live site, reads the accessibility tree/DOM, and judges observed states. Stateful flows (register $\rightarrow$ login $\rightarrow$ befriend $\rightarrow$ enter room) are set up once and reused. Destructive flows get isolated browser sessions.
- **API** (`api_eval/`): authenticated HTTP sequences against discovered routes; each check stores a request/response `execution_log` and a scored `checklist`.
- **Code** (`code_review_eval/`): a directory filter first keeps relevant folders and drops noise (e.g. `.claude`, `.legogent`), then targeted search and LLM inspection verify constraints that are not externally visible, such as password hashing or transport security.

Step 4: Weighted checklists and evidence. Every requirement produces a checklist of weighted scenarios, each with a verdict and a human-readable justification, plus raw evidence (`api_response`, `frontend_test`, or `code_snippet`). The bottom of Figure 8 shows a passing API check and a failing code check from the same run—note that verdicts cite concrete behavior (measured latency, exact status codes, file:line).

```
=====  COLLABORATION: LegoGent periodic sync  ============================================
[BTW_PROGRESS_ALIGNMENT #5]  periodic sync broadcast, ~every 244s; 52 peer-mailbox threads
    this run
NOTE: alignment snapshot, NOT a completion signal. A round enters user-review ONLY when the
runtime state is all-idle-and-stable AND the mode's required artifacts exist.

Runtime states:  dev-1: idle 244s   dev-2: idle 244s   dev-3: idle 244s   dev-4: idle 244s

Per-agent snapshot (1 done / 2 evidence / 3 blockers / 4 next):
 [dev-1] 1: all backend APIs + full frontend integrated on single port 8002.
         2: curl all-green; WebSocket realtime ~9ms; Playwright e2e 9/10 steps pass.
         3: dev-2/3/4 crashed on a provider API error; dev-1 took over the remaining frontend
    work.
         4: fix group-chat view activation; rerun Playwright; write docs (api-spec, modules,
    report).
 [dev-2] 1: conversation/message endpoints + WebSocket push done.
         2: curl branches (401/403/400/404) green; WS 8.9ms (<200ms).
         3: waiting on shell App.tsx before frontend build.
         4: mount ChatPage; dual-browser Playwright verify.
 [dev-3] 1: reply-count / single-message / history-filter / soft-delete endpoints landed,
    integrated by dev-1.
         2: selftest 36/36 assertions green incl. 401/403/404/422.
         3: not hard-blocked.
         4: build message-ops UI; coordinate mount point with dev-1.
 [dev-4] --: crashed on the provider API error; its remaining frontend work was absorbed by
    dev-1 (above).

=====  EVALUATION: TAgent grades the same round, then feeds a ranked repair agenda to round N
    +1  ====
# discovery_result.json -- reconstructed route surface (excerpt: method, path, auth, params)
POST   /api/users/register   auth:no   {username,password,nickname?}
POST   /api/users/login      auth:no   {username,password}
DELETE /api/users/me         auth:yes  {}            # cancel account
GET    /api/users/search     auth:yes  {keyword}
POST   /api/friends/request  auth:yes  {to_id,message?}

# API evaluator -- requirement: chat_send_text
scenario "send text"          weight 3.0  PASS
  user1 sent a text in conversation 4; user2 received immediately; unread count updated;
  retrievable via GET /api/conversations/4/messages with full body + metadata.
scenario "empty & long text"  weight 1.0  PASS
  empty message rejected 400 ("content must not be empty"); ~650-char accepted 200, body
    preserved.

# Code evaluator -- requirement: sys_security_no_plaintext
dir_filter: keep=[backend,docs,frontend,tests] drop=[.claude,.legogent]
scenario "HTTPS enforced"       weight 2.0  FAIL
  backend/main.py serves plain HTTP on port 8002; no TLS/redirect/HSTS.
scenario "sensitive transport"  weight 1.0  FAIL
  passwords sent as plaintext JSON (api.ts:32); bcrypt is server-side only
  (security.py:28) -> raw plaintext over unencrypted HTTP.
```

Figure 8: One REVIBE round end to end, from the feature-squad / Claude Opus 4.8 / instant-messaging run. **Top (collaboration):** a periodic LEGOGENT sync broadcast—the runtime-state block and the "idle-and-stable + required-artifacts" gate are the actual basis for accepting a round, and each agent reports the same four fields every cycle; here dev-4 crashed on a provider API error and dev-1 absorbed its frontend work, a real multi-agent failure the round still had to reconcile. **Bottom (evaluation):** TAGENT then grades the deployed round—route discovery, a passing API check, and a failing security check—with every verdict citing concrete behavior (status codes, measured latency, `file:line`); this item-level evidence is returned to LEGOGENT as the ranked repair agenda that opens the next round.

Step 5: Aggregation and feedback. Scenario scores are summed by rubric weight to the 100-point score. A requirement graded by more than one evaluator (e.g. backend + frontend) is averaged per `requirement_id`, so a feature only counts as done when it holds end to end. The same item-level evidence—observed state, failing prerequisite, and affected weight—is returned to LEGOGENT as a ranked repair agenda for the next round.

Where LLM judgment enters. The rubric weights and the aggregation are deterministic once the rubric exists. All LLM judgment uses DeepSeek v4 Pro and enters at four points: compiling the requirement document into the rubric (Step 1, done once per project, checked by human experts, who found no changes necessary, and shared by every run of that project, so every run is judged

Table 10: General failure taxonomy over 1,440 failing TAGENT rubric checks. "% checks" is the share of all failing checks; "% runs" is the fraction of runs with at least one such failure.

Failure mode	% checks	% runs
Incomplete logic & edge cases	32.2	88
Missing / unimplemented feature	23.8	75
Auth, session & permission	11.9	46
Prerequisite cascade	11.0	39
Security & transport (deploy ceiling)	8.5	78
Other functional defect	6.1	38
Documentation gaps	3.9	26
Integration & UI wiring	2.6	22

against the same checks), driving the browser and judging observed UI states and inspecting code for constraints that are not externally visible (Step 3), and declaring blocking dependencies among UI checks for scheduling (below). Every verdict stores its evidence (Step 4), so a disputed verdict can be audited against the recorded API response, UI test record, or code excerpt. Because the rubric is an editable specification file, it can be extended with new checks without changing TAGENT.

Parallel UI scheduling. API and code checks are mostly stateless and fan out under a concurrency cap. UI checks are stateful: testing a group chat may require registration, login, friend creation, and room entry. Running every check independently repeats this setup, while naive parallelism duplicates it across workers. TAGENT therefore asks an LLM to declare only blocking dependencies among UI requirements, forms a task DAG, and partitions it across browser workers with an integer-programming objective that minimizes makespan plus duplicate setup. Prerequisites are co-located with dependents, shared setup is built once per worker, unreachable prerequisites cascade-skip their dependents, and duplicate probes are merged by taking the best verdict.

Evaluator parallelism is deliberately conservative. Because web applications are not pure functions, TAGENT parallelizes probes only after discovering dependencies: a worker that deletes a friend, submits a group application, or changes profile data can poison another check sharing the same account state. The scheduler thus prefers independent browser sessions for destructive flows and shared sessions for long setup chains, trading wall-clock time for verdicts that do not depend on a lucky execution order.

G FAILURE TAXONOMY DETAILS

Each unmet rubric check in Table 10 is labeled with one root cause; 86% of failed items still retain partial credit and only 14% score zero.

Functional shortfall. Missing features (23.8%) and incomplete logic or edge cases (32.2%) together account for 56% of failing checks and a similar share of lost weight. The typical defect is an endpoint that exists but skips a validation, a cleanup, or a secondary path—account deletion that invalidates the token but never cascades to messages, or a history filter that returns empty on a valid range—exactly the requirements a document-driven grader surfaces but a self-reported "done" would hide.

Prerequisite cascades. Authentication and permission failures (11.9%) and prerequisite cascades (11.0%) are together 23% of all failures. When login, token refresh, or a gating step such as friend-request breaks, every dependent scenario becomes untestable: a single broken prerequisite auto-skipped up to 14 downstream checks in one run, and cascades appear in 39% of runs. In a team, the broken prerequisite and the skipped checks often belong to different owners.

Deployment ceiling. Security and transport items (8.5% of checks) recur in nearly 80% of runs because they depend on successful integration with XDEPLOY, the deployment layer that supplies TLS, HTTPS redirects, and secure WebSockets; deployment is carried out through XDEPLOY rather than locally, so these are a near-constant deduction from unsuccessful integration rather than a coding error. Documentation (3.9%) and UI-wiring defects (2.6%) are minor.

Table 11: Provider list prices used for the budget metric, in USD per million tokens: new input, cache write, cache read, and output. "—" marks a field not billed as a separate line item (cache writes fold into new input). Anthropic and OpenAI publish in USD; DeepSeek, Z.AI (GLM-5.2), and Alibaba Cloud (Qwen) publish in CNY and are converted at a fixed 6.6 CNY/USD. Qwen3.6-Flash scored 0 across the grid and is listed only for completeness.

Model	**New-in**	**Cache-wr**	**Cache-rd**	**Out**
Claude Opus 4.8	5.00	6.25	0.50	25.00
GPT-5.5	5.00	—	0.50	30.00
DeepSeek v4 Pro	0.45	—	0.0038	0.91
DeepSeek v4 Flash	0.15	—	0.0030	0.30
GLM-5.2	1.21	—	0.30	4.24
Qwen3.6-Flash	0.18	0.23	0.018	1.09

Table 12: Real per-round token accounting for feature squad / Claude Opus 4.8 / instant messaging (four agents). Cache-creation dominates, which is why cache-aware pricing is required for a fair budget comparison.

Round	**New-in** (k)	**Cache-cr** (M)	**Cache-rd** (M)	**Out** (k)	**Total** (M)
R1	14.5	65.09	7.18	299.5	72.58
R2	0.4	18.61	0.65	65.8	19.33
R3	0.4	18.28	2.88	64.5	21.22
Sum	15.3	101.98	10.71	429.8	113.13

H Token and Cost Accounting

The budget metric is computed from four token fields logged per agent and per round: *new-input*, *cache-create*, *cache-read*, and *output*. Cost is the price-weighted sum over these fields,

$$\text{USD} \;=\; \sum_{a,r} \big(n^{\text{in}}_{a,r}\,\pi_{\text{in}} + n^{\text{cc}}_{a,r}\,\pi_{\text{cc}} + n^{\text{cr}}_{a,r}\,\pi_{\text{cr}} + n^{\text{out}}_{a,r}\,\pi_{\text{out}} \big),$$

where the per-model rates $\pi_\bullet$ are the published provider list prices in Table 11. Separating the four fields is what lets REVIBE distinguish raw token volume from actual budget pressure: cache reads are far cheaper than fresh input, so two runs with similar total tokens can differ several-fold in dollars. Table 12 gives the real per-round breakdown for the worked example; the totals (≈113 M tokens, 110 min, best score 96.95) match its row in Table 3. Table 11 lists the four per-model rates; applied to the token counts below they reproduce the USD cost column of every results table—for instance the $417.97 first-round cost of this worked example.

I A Worked Refinement Trace

Table 13 follows a subset of the 56 requirements across the three rounds of the worked example, illustrating three phenomena discussed in Section 4.3. First, mixed requirements are averaged across evaluators (backend `be` / frontend `fe`), so a feature that passes one side but not the other scores partially. Second, most R1 failures are repaired once TAGENT feedback localizes them (user-search, cancel-account, requirements-doc). Third, refinement is not monotone: at R3 "delete chat history" regresses on the backend (`be` $1 \to 0.5$) while other items are being fixed—the same class of cross-requirement regression reported in Section 4.3.

J Compute Environment

All runs execute on four Intel(R) Xeon(R) Gold 6430 processors. Deployment, testing, and evaluation use isolated port ranges so concurrent configurations do not interfere. Each configuration

Table 13: Real per-requirement trajectory (excerpt) for the worked example. Bold marks a backend regression at R3. The bottom row is the aggregate Functional Completion Score.

Requirement	**Cat.**	**R1**	**R2**	**R3**
Front–back communication	Sys	10/10	10/10	10/10
Message latency $<$200 ms	Sys	4/4	4/4	4/4
Cancel account	Fun	0/1	1/1	1/1
Login/logout	Fun	2.9/3	3/3	3/3
User search	Fun	1/2	2/2	2/2
Delete chat history	Fun	0.5/1	1/1	**0.8/1**
No-plaintext transport	Sys	0/4	0/4	2/4
Requirements doc	Doc	0/3	3/3	3/3
Total		**79.22**	**93.07**	**96.95**

allows up to three refinement rounds; a round builds or patches the project, deploys it through the XDEPLOY layer (which supplies TLS, HTTPS redirects, and secure WebSockets), runs TAGENT, and returns scored feedback. Per-round wall-clock is capped (e.g. 90 minutes), which is the source of some truncated first rounds analyzed in Section 4.3.