Spec2Vision: Contract-Guided Delivery of AI-Generated Computer Vision Pipelines

Ghfran Jabour[1], Sergey Ivanov[1]

[1]ITMO University

Abstract

Generated computer-vision code can be runnable without satisfying the task contract enforced by a downstream evaluator. We study that gap with Spec2Vision, an experimental framework for producing and evaluating specification-grounded CV pipeline bundles through a staged runtime that keeps the task contract explicit across synthesis, screening, testing, and bounded repair. The benchmark evaluates 17 CV tasks, 10 executable conditions, and 5 repeats per task-condition cell, for 850 primary runs. In the primary 850-run evaluation, Spec2Vision reaches 81/85 evaluator-test passes; removing structural repair drops to 55/85, compatibility scaffolding to 58/85, and generator preflight to 39/85. The executable single-agent baselines expose progressively richer task specifications to the model, culminating in direct source-spec exposure, yet remain much weaker overall, from 17/85 for lightweight task grounding to 35/85 evaluator-test passes. The lightweight baseline nevertheless remains core-runnable in 85/85 runs but reaches only 17/85 evaluator-test passes and 6/85 strict-delivery successes, showing that runnability is not equivalent to delivery. Across this benchmark, the strongest evidence comes from keeping the task contract explicit across staged generation, checking, and repair. Artifacts are provided to support audit of run bundles, model-visible inputs, and derived tables.

Keywords

computer vision pipelines, code generation, structured task specifications, software testing, multi-agent systems

1. Introduction

Large language models now synthesize non-trivial code and increasingly tackle repository-level software tasks [1, 2, 3]. Yet runnable generated code is still not the same as delivered software. In computer vision, a generated bundle must satisfy downstream evaluator requirements over data layout, outputs, metrics, and interfaces; surviving a dry run is often not enough.

This paper studies that gap for generated CV pipelines. The question is bounded: within a fixed benchmark, does carrying an explicit task contract through staged generation, testing, and repair improve downstream delivery over weaker variants and single-agent baselines?

Spec2Vision is an experimental framework for producing and evaluating specification-grounded CV pipeline bundles; its executable component takes a structured CV task specification and produces a runnable bundle with persisted artifacts for audit. The key design choice is to treat the specification as an operational contract that remains visible through context assembly, architect-to-generator handoff, deterministic preflight checks, testing, and bounded repair.

In the primary 850-run evaluation, Spec2Vision reaches 81/85 evaluator-test passes. Some nearby variants remain competitive, with only small differences among the strongest conditions, including the no-history variant at 83/85 and the no-architect-guidance variant at 78/85. The stronger evidence comes from the larger drops to 58/85 without compatibility scaffolding, 55/85 without structural repair, and 39/85 without generator preflight. Even progressively richer task-specification exposure raises the executable single-agent baselines only from 17/85 to 35/85 evaluator-test passes; the lightweight baseline remains core-runnable in 85/85 runs but satisfies strict delivery in only 6/85. The central result is therefore that runnability is an insufficient proxy for downstream delivery.

We organize the study around three research questions: **RQ1.** Does contract-carrying staged generation improve downstream delivery success for generated CV pipelines compared with executable

GeCoIn 2026: Generative Code Intelligence Workshop, co-located with the 35th International Joint Conference on Artificial Intelligence (IJCAI-ECAI 2026), August 15–17, 2026 — Bremen, Germany

ghfranj@itmo.ru (G. Jabour); svivanov@itmo.ru (S. Ivanov)

published 2026-08-14

single-agent baselines? **RQ2.** Which Spec2Vision mechanisms contribute most to delivery success: structural repair, generator preflight, compatibility scaffolding, history context, or architect guidance? **RQ3.** Where do generated CV pipelines fail when they are runnable but not delivered?

The paper makes three contributions:

- A specification-grounded staged architecture for CV pipeline synthesis that carries an explicit task contract through architect-to-generator handoff, deterministic screening, testing, and bounded repair.
- A fixed 17-task contract-diverse benchmark and delivery-state evaluation framework for generated CV pipelines, with persisted artifacts for auditing evaluator outcomes, model-visible inputs, and shallow-contract signals.
- Evidence that the largest stable gains in this benchmark come from contract-carrying repair and deterministic screening, while the executable single-agent baselines define a progressively richer task-specification ladder that still does not close the delivery gap, even with direct source-spec exposure.

2. Related Work

Work on LLM code generation has largely been evaluated through executable correctness on self-contained programming tasks. HumanEval/Codex [1], MBPP [4], APPS [5], MultiPL-E [6], and AlphaCode [7] each measure important parts of synthesis quality, but they mostly test whether a generated program satisfies hidden tests for a bounded programming task. Those settings differ from ours in two ways. First, we evaluate a generated CV pipeline bundle rather than a short standalone program. Second, success is defined by whether that bundle satisfies the downstream task contract: the required files, interfaces, outputs, metrics, and dataset-layout assumptions. This differs from generic functional-correctness benchmarks for self-contained programming tasks.

Repository-level benchmarks and coding agents move closer to realistic software work by evaluating issue resolution in existing codebases. SWE-bench [2] frames the task as fixing real GitHub issues, and systems such as SWE-agent [3], AutoCodeRover [8], Agentless [9], and OpenHands/OpenDevin [10] study tool use, code editing, execution feedback, sandboxed environments, and benchmarked software-agent behavior. CodeAct [11] further studies executable code actions as an agent action space. Recent evaluation work also stresses that agent success is sensitive to benchmark design: SWE-bench Multimodal extends issue resolution to visual, user-facing JavaScript software [12], while UTBoost shows that patches can pass available SWE-bench tests without fully resolving the issue [13]. These works are closest to our concern with execution feedback and agentic evaluation, but our scope is different: Spec2Vision does not repair an existing repository issue or evaluate a general coding agent. It starts from a structured CV task specification and evaluates whether the generated bundle satisfies the downstream evaluator's task contract.

The design of Spec2Vision also draws on work in iterative refinement, reasoning-and-acting, automated program repair, and multi-agent software generation. ReAct [14] couples reasoning with environment interaction; Self-Refine [15] and Reflexion [16] use feedback across iterations; automated program repair studies broader repair mechanisms and their limits [17]. AutoGen [18], MetaGPT [19], and ChatDev [20] show how role separation can organize planning, implementation, and review across interacting agents. We adopt the broad intuition that staged interaction can help, but the paper does not argue for universal multi-agent superiority. Our use of bounded repair and role separation is narrower: it is inserted into a delivery-oriented pipeline whose goal is to preserve task-contract alignment rather than to maximize generic debugging, autonomous tool use, or open-ended issue resolution.

Finally, Spec2Vision sits alongside ML systems, testing, reproducibility, and AI requirements work. Hidden technical debt [21], production-readiness criteria [22], ML software engineering practice [23, 24], and reproducibility efforts [25] all emphasize that data-dependent systems fail through interfaces, data assumptions, and process gaps rather than model quality alone. Requirements-engineering studies for ML and AI-based systems likewise stress traceability, testable requirements, and specification

errors [26, 27, 28]. Our contribution in that landscape is not another general coding-agent scaffold, but a delivery-state benchmark and staged contract-carrying pipeline for generated CV bundles, with comparators that separate specification exposure, role separation, deterministic screening, bounded repair, and downstream task-contract satisfaction.

3. Spec2Vision and the Benchmark

Spec2Vision takes a structured CV task specification as input and produces a runnable project bundle together with persisted traces, diagnostics, and evaluator results. The pipeline is staged: an architect plans the interfaces, a generator materializes the project, deterministic preflight screens the bundle shape, a validator/tester runs execution-facing checks, and later stages record feedback and verification evidence. The task specification is therefore carried forward as an operational contract rather than left as prompt background.

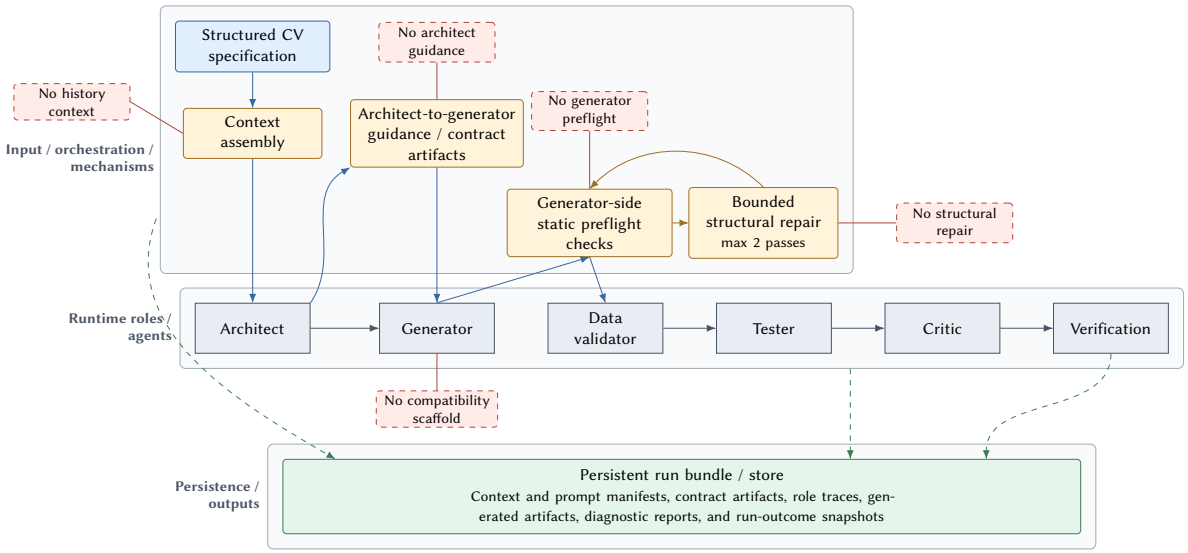

Figure 1: Primary Spec2Vision runtime and main ablation points. Rectangles denote staged roles, rounded boxes denote orchestration mechanisms, and dashed boxes mark removed or weakened components. The CV specification is converted into model-visible context and persisted contract artifacts; generator preflight and bounded repair form a local control path before validation. The run bundle stores artifacts, traces, and reports. External baselines are omitted.

Figure 1 summarizes the primary execution path. The architect stage converts the task description into a generator-facing contract over expected data interfaces, outputs, and metrics, and preflight checks whether the produced bundle is executable in the required shape before downstream testing proceeds.

In the evaluated implementation, the architect-to-generator handoff is persisted as a compact artifact bundle: a natural-language architecture plan describing the intended modules and data flow, a JSON module graph, a YAML per-module interface contract, and a compiled JSON generator contract mapping modules to canonical files that the generator is expected to produce. Preflight then checks syntax, imports, undeclared dependencies, the required runtime entry point, and dry-run execution, plus lightweight heuristics for dry-run evidence and model-task drift. If fatal structural findings remain, the generator may attempt targeted structural repair, but this repair loop is bounded to two passes; after that, the run either proceeds with the repaired bundle or is recorded as failed.

The benchmark is designed to test delivery across materially different CV pipeline structures under a common specification language and downstream evaluator. It contains a fixed set of 17 tasks chosen for contract diversity across detection, segmentation, classification, tracking, pose, scene text, depth, optical flow, video segmentation, lane detection, stereo, LiDAR 3D detection, and scene parsing, with pinned local dataset layouts. The goal is evaluator-facing delivery against known task requirements rather than external data acquisition or leaderboard performance.

These tasks are contract-diverse in precisely the dimensions that determine whether a generated bundle satisfies the evaluator. Expected outputs range from class labels and scalar reports to masks, trajectories, polylines, dense maps, bounding boxes, and structured 3D detections; expected metrics likewise vary across accuracy-style scores, overlap measures, tracking statistics, and task-specific latency or reporting requirements. The tasks also encode different dataset and directory assumptions, creating opportunities for family mismatch or layout mismatch even when code dry-runs. A generated bundle can therefore be executable while still failing the evaluator because it delivers the wrong artifact shape, metric schema, or data contract.

We use this task set as a fixed grid for comparing executable conditions. Each executable condition is run five times on every task, yielding 85 runs per condition and 850 runs in the primary evaluation; two comparable reruns are used only for stability checks. All conditions share the same task grid, repeat structure, artifact store, and downstream evaluator.

Across executable conditions, all LLM calls use a common client interface and the same environment-resolved provider/model configuration. In the audited primary evaluation, this resolved to `provider=gemma, model=gemma-4-26B-A4B-it` for all executable conditions, with per-call identifiers persisted. We treat this model as a fixed generation substrate rather than a model-ranking target: several weaker conditions reach 85/85 core runnability, showing that the main contrasts are not explained by an inability to emit executable code. The artifact-derived timing summary reconstructs wall-clock spans for the primary run and two comparable reruns, but we do not claim hardware-normalized throughput because exact worker counts and GPU-hours were not recorded in the original run bundles.

The outcome definitions are delivery-oriented. *Artifact-complete* means that the generated bundle contains the required project files, configuration files, and evaluator-facing artifacts. *Core-runnable* means the package imports and satisfies the required dry-run checks. *Evaluator-test pass* is the main outcome: a persisted harness result that the bundle satisfies task-facing tests. For Spec2Vision-family conditions the tester stage instantiates that harness from the task specification and upstream artifacts; for single-agent comparators an equivalent condition-appropriate harness is attached deterministically after generation. *Strict delivery* adds a shallow check over required outputs, metrics, and obvious placeholder/mock substitution. These outcomes measure contract satisfaction rather than scientific CV quality.

The downstream evaluator is designed to be condition-agnostic: it operates on the delivered bundle and task-facing outputs rather than on Spec2Vision-specific internal artifacts. Evaluator-test and shallow-contract checks do not inspect architect plans, role traces, context manifests, critic outputs, or whether a bundle was produced by staged orchestration or by a single-agent runner. The executable single-agent baselines are therefore assessed through the same delivered-file, configuration, runtime, output, and metric-facing checks as the Spec2Vision-family conditions. Table 1 summarizes these checks and the Spec2Vision-specific internal artifacts excluded from them.

Table 1
Evaluator-facing checks and excluded Spec2Vision-specific internal artifacts.

Check family	What is inspected	What is not inspected
Runtime checks	Generated package, imports, required entrypoint, dry-run behavior	Architect plan, role traces, context manifests, critic artifacts
Evaluator tests	Delivered task-facing files, outputs, metrics, and execution results	Whether the bundle came from staged roles or a single-agent runner
Shallow contract	Required configuration, declared outputs, expected metrics, placeholder/mock signals	Spec2Vision-specific handoff artifacts or internal role structure
Bundle reporting	Persisted run outcomes, failure summaries, and checker outputs	Manual judgments or condition-specific success rules

Persisted run bundles retain model-visible inputs, intermediate artifacts, harness outputs, and shallow-contract signals, so the reported outcomes can be audited from concrete artifacts rather than narrative summaries alone. Table 2 summarizes what each condition receives, what machinery it excludes, and

what comparison it supports.

Table 2
Condition groups, model-visible input, and comparison purpose. The executable single-agent baselines form an input-exposure ladder under the same bounded single-agent runner.

Condition / group	Class	Model-visible input	Excluded machinery / purpose
Spec2Vision primary	Staged	Normalized spec and layered context; generator also sees architect handoff artifacts; later roles consume upstream artifacts and execution evidence	Reference system for contract-carrying staged generation, deterministic preflight, compatibility scaffolding, and bounded repair.
Spec2Vision-family ablations	Staged variants	Same staged inputs as the primary system, with one local mechanism removed or weakened	Mechanism tests inside the same runtime: compatibility scaffold, architect guidance, structural repair, history context, or generator preflight.
Lightweight baseline	Single-agent	Task id, objective, domain, modalities, and fixed file/runtime instructions; no raw spec or normalized CV-Spec	Compact grounding-only executable baseline.
Contract-spec baseline	Single-agent	Normalized CV-Spec plus a derived evaluator-facing contract; no layered context or staged handoff	Tests whether structured contract exposure helps without staged orchestration.
Source-spec baseline	Single-agent	Complete catalog-resolved source specification plus the same fixed file/runtime instructions; no layered context or staged handoff	Tests whether direct source-spec exposure alone is enough for a single agent.
Full-context single-agent control	Single-agent control	Normalized full specification plus layered task/domain/data/history/failure-pattern context, but synthesis collapsed into one agent	Reuses richer context construction while removing staged role separation.

The three executable single-agent baselines share the same bounded runner: 3 synthesis attempts, 2 repair attempts, and 16 generated files, together with the same fixed file/runtime envelope. The comparison against Spec2Vision is therefore a shared-harness system comparison rather than a prompt-identical ablation. We report the full-context single-agent control separately because it reuses Spec2Vision's richer context construction while collapsing synthesis into one agent, allowing us to distinguish role separation from increased specification exposure.

Lane detection example. Lane detection illustrates why the benchmark separates runnability from delivery. The task contract requires structured lane polylines and lane-specific precision, recall, F1, and latency reporting. In the primary evaluation, the lightweight single-agent baseline remained dry-run runnable in all five repeats but fell back to generic outputs or metrics. The corresponding Spec2Vision runs preserved the lane-specific contract and passed all five evaluator tests. This example shows why runnability alone is not sufficient evidence of task-contract delivery.

4. Results

Table 3 reports the primary 850-run evaluation together with the range observed across that evaluation and two comparable reruns. The main outcome is evaluator-test pass, with strict delivery shown separately because it reveals an additional failure mode in the weaker baselines. The results answer RQ1 by comparing Spec2Vision with the executable single-agent baselines, RQ2 by analyzing the mechanism-removal conditions, and RQ3 by separating core runnability from evaluator-test and strict-delivery failures.

For RQ1, staged contract-carrying generation substantially outperforms the executable single-agent baselines under the shared benchmark and downstream evaluator. Read as an input-exposure ladder, the single-agent baselines improve from 17/85 to 27/85 to 35/85 evaluator-test passes as more specification material becomes visible, but even direct source-spec exposure remains 41–46 evaluator-test passes below Spec2Vision across the primary evaluation and comparable reruns. The full-context internal single-agent control reaches 18/85, so richer context alone does not replace staged role separation.

For RQ2, small differences among the strongest Spec2Vision-family variants should be interpreted conservatively. The no-history condition reaches 83/85 evaluator-test passes in the primary evaluation

Table 3
Main delivery outcomes. Evaluator-test pass is the paper's primary full-test outcome; strict delivery adds a shallow deliverable contract. Primary counts come from the 850-run primary evaluation, and the final column shows the range across that wave and two comparable reruns.

Condition	Artifact complete	Core runnable	Evaluator test pass	Strict delivery	Eval. range
Spec2Vision primary	82/85	82/85	81/85	81/85	81–81
No compatibility scaffold	73/85	73/85	58/85	58/85	57–59
No architect guidance	79/85	79/85	78/85	78/85	77–78
No structural repair	55/85	55/85	55/85	55/85	55–59
No history context	83/85	83/85	83/85	83/85	82–85
No generator preflight	82/85	82/85	39/85	39/85	35–40
Lightweight baseline	85/85	85/85	17/85	6/85	11–17
Contract-spec baseline	85/85	85/85	27/85	27/85	27–38
Source-spec baseline	85/85	85/85	35/85	35/85	35–40
Full-context single-agent control	82/85	73/85	18/85	18/85	17–19

and ranges from 82/85 to 85/85 across the primary evaluation and comparable reruns, while the no-architect-guidance condition remains at 77/85–78/85. We therefore treat these differences descriptively rather than as strong mechanism evidence. The stronger evidence comes from the larger stable gaps: removing compatibility scaffolding lowers evaluator-test pass to 57/85–59/85 across the evaluations, removing structural repair to 55/85–59/85, and removing generator preflight to 35/85–40/85 despite 82/85 core runnability in the primary evaluation.

For RQ3, the lightweight baseline makes the delivery gap concrete: it is core-runnable in 85/85 runs but reaches only 17/85 evaluator-test passes and 6/85 strict-delivery successes. Strict delivery is especially informative for weaker baselines because it separates evaluator-test failure from more immediate deliverable mismatches, such as generic outputs, missing task-specific metrics, or placeholder behavior.

We assign each strict-delivery non-success to one dominant delivery-layer class using persisted run-outcome, failure-summary, and shallow-contract signals: integration/interface failure, evaluator-test failure, task-contract mismatch, metric/output-contract failure, placeholder/mock-substitution failure, data/layout failure, or other/unknown.

Table 4
Delivery-layer failure distribution in the primary evaluation. Counts are over strict-delivery non-success runs in `command-paper-experiment-20260511T035415Z-95443b`; each run is assigned one dominant class from persisted artifact signals.

Condition	Non-succ.	Int.	Eval.	Task	Metric	Mock	Data
Spec2Vision primary	4	3	1	0	0	0	0
No compatibility scaffold	27	12	15	0	0	0	0
No architect guidance	7	6	1	0	0	0	0
No structural repair	30	30	0	0	0	0	0
No history context	2	2	0	0	0	0	0
No generator preflight	46	3	2	0	0	39	2
Lightweight baseline	79	0	1	28	10	15	25
Contract-spec baseline	58	0	8	12	4	13	21
Source-spec baseline	50	0	15	11	1	22	1
Full-context single-agent	67	12	15	8	1	31	0

Table 4 sharpens RQ3 into distinct failure regimes. The executable single-agent baselines are usually runnable before failing on task-contract, metric/output, placeholder/mock, data/layout, or downstream evaluator signals. By contrast, removing structural repair concentrates failures almost entirely in integration/interface breakdowns, while removing generator preflight shifts many failures to late placeholder/mock and data issues. The artifact-derived taxonomy therefore supports a more specific claim than "runnable is not enough": reliable delivery depends on preserving the task contract through

generation, checking, and bounded recovery.

5. Discussion

The main interpretation is that specification exposure helps, but it is not sufficient for delivery. The single-agent ladder improves as more task material becomes visible, yet even direct source-spec exposure remains far below Spec2Vision. This suggests that the delivery bottleneck is not only informational but also structural: the task contract must be preserved across planning, file generation, checking, and bounded recovery, not merely exposed to a collapsed single-agent runner.

The mechanism results support this interpretation. Structural repair is the cleanest principal contrast because removing it sharply reduces evaluator-test pass while leaving the broader staged pipeline intact. Generator preflight provides strong supporting evidence for deterministic screening: without it, many bundles remain core-runnable but fail downstream delivery. Compatibility scaffolding contributes more moderately. By contrast, the no-history and no-architect-guidance results should be interpreted conservatively; in this benchmark, their effects are small and may be partly redundant with the normalized specification, generator contract, deterministic checks, and downstream evaluator.

The failure taxonomy further shows that the mechanisms address different layers of the delivery problem. Single-agent baselines often remain runnable before failing through task-contract mismatch, metric/output mismatch, placeholder behavior, or data-layout drift. Removing structural repair instead concentrates failures in integration and interface breakdowns, while removing generator preflight shifts many failures toward later placeholder and data issues. These patterns support a more specific claim than "runnable is not enough": reliable delivery depends on preserving the task contract through generation, checking, and repair.

The full-context single-agent control also shows that richer context is not equivalent to staged role separation. It reuses the layered context construction but collapses synthesis into one agent, removing the explicit architect-to-generator handoff and intermediate checkpoints. We treat this as bounded evidence rather than causal proof: the benchmark shows that this collapsed control does not replace the staged pipeline here, not that any multi-agent decomposition will dominate any single-agent alternative.

Taken together, the results support the paper's central distinction between runnable code and delivered pipelines. Within this benchmark, contract-carrying staged generation produces more evaluator-compliant CV bundles than runnable code generation alone. The evidence is intentionally scoped to task-contract delivery, not to scientific CV quality or production readiness.

6. Threats to Validity

The benchmark is intentionally bounded but execution-heavy: 17 fixed CV tasks, 10 executable conditions, 5 repeats per task-condition cell, and two comparable reruns. The tasks were selected for contract diversity across CV families, output shapes, metrics, and dataset-layout assumptions, allowing the benchmark to characterize system behavior across different delivery contracts. They still share one specification language and one evaluator-facing contract, so the results should be interpreted as evidence about the represented task-contract space rather than all possible CV pipeline-generation settings.

The downstream evaluator is condition-agnostic but not an external third-party benchmark. Because the benchmark is specification-grounded, it may appear favorable to a contract-carrying system; we mitigate this by applying success checks across all conditions to delivered files, runtime behavior, outputs, metrics, and shallow-contract signals rather than to Spec2Vision-specific internal artifacts. This supports comparison across the evaluated conditions, but the benchmark still measures preservation of the provided task contract rather than independent CV model quality.

The comparison is not an equal-internal-machinery comparison: Spec2Vision has staged roles and deterministic control mechanisms by design. The single-agent baselines test whether comparable

delivery can be achieved under the same task grid, model path, execution envelope, and evaluator without that machinery.

Statistical and execution uncertainty remain. Five repeats per task-condition cell and two comparable reruns expose some stochastic variation, but they do not remove dependence on the particular task set, model outputs, serving configuration, and local environment. We therefore treat large stable gaps as stronger evidence than small differences among the strongest variants, and interpret history-context and architect-guidance effects descriptively. The delivery-layer taxonomy is artifact-derived and assigns one dominant class per non-success run, so it supports descriptive comparison rather than exhaustive causal attribution.

7. Conclusion

Within this benchmark, the answers to the three research questions are direct. For RQ1, Spec2Vision substantially outperforms the executable single-agent baselines under the same benchmark and downstream evaluator. For RQ2, the strongest mechanism-level evidence comes from structural repair and generator preflight, with compatibility scaffolding providing a smaller supporting contribution and history/architect-guidance effects remaining descriptive. For RQ3, many generated CV bundles are runnable but not delivered because they fail task-contract, metric/output, placeholder/mock, data-layout, or evaluator-test checks.

The central result is that runnable generated code is not the same as delivered CV pipeline behavior. Within the represented task-contract space, carrying the task contract through staged generation, deterministic checks, testing, and bounded repair yields far more evaluator-tested deliveries than runnable code generation alone.

Future work should test how far this result extends beyond the current `cv-spec` setting by increasing the number and diversity of tasks, covering additional CV families and specification styles, evaluating additional model families and serving settings, studying more external evaluator designs, and refining the failure taxonomy to capture multi-causal delivery failures.

Acknowledgment

This work was supported by the Ministry of Economic Development of the Russian Federation (IGK 000000C313925P4C0002), agreement No. 139-15-2025-010.

Artifact Availability

The anonymized repository is available at https://github.com/annony-ml-llm/Spec2Vision.git. The artifact archive, `spec2vision_artifacts.tar.gz`, is available from the repository's GitHub Releases page and contains the task catalog, specifications, frozen experiment configuration, runtime and harness code, persisted run bundles, prompt/input manifests, shallow-contract reports, failure summaries, and derived analysis tables used in this paper. Full reruns require compatible model serving, local dataset layout, and environment configuration. A post-hoc execution-server snapshot records Ubuntu 22.04.5, 2× AMD EPYC 9124 CPUs, 1.0 TiB RAM, and 4× NVIDIA RTX 6000 Ada GPUs with driver 565.57.01 and CUDA 12.7.

Declaration on Generative AI

We used generative AI tools for language editing and restructuring. We reviewed the final manuscript and take responsibility for its content.

References

[1] M. Chen, et al., Evaluating large language models trained on code, 2021. URL: https://arxiv.org/abs/2107.03374. doi:10.48550/arXiv.2107.03374. arXiv:2107.03374.

[2] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, K. Narasimhan, SWE-bench: Can language models resolve real-world GitHub issues?, 2023. URL: https://arxiv.org/abs/2310.06770. doi:10.48550/arXiv.2310.06770. arXiv:2310.06770.

[3] J. Yang, et al., SWE-agent: Agent-computer interfaces enable automated software engineering, 2024. URL: https://arxiv.org/abs/2405.15793. doi:10.48550/arXiv.2405.15793. arXiv:2405.15793.

[4] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, C. Sutton, Program synthesis with large language models, 2021. URL: https://arxiv.org/abs/2108.07732. doi:10.48550/arXiv.2108.07732. arXiv:2108.07732.

[5] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, J. Steinhardt, Measuring coding challenge competence with APPS, 2021. URL: https://arxiv.org/abs/2105.09938. doi:10.48550/arXiv.2105.09938. arXiv:2105.09938.

[6] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, A. Jangda, MultiPL-E: A scalable and extensible approach to benchmarking neural code generation, 2022. URL: https://arxiv.org/abs/2208.08227. doi:10.48550/arXiv.2208.08227. arXiv:2208.08227.

[7] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, et al., Competition-level code generation with AlphaCode, Science 378 (2022) 1092–1097. URL: https://www.science.org/doi/10.1126/science.abq1158. doi:10.1126/science.abq1158.

[8] Y. Zhang, H. Ruan, Z. Fan, A. Roychoudhury, AutoCodeRover: Autonomous program improvement, 2024. URL: https://arxiv.org/abs/2404.05427. doi:10.48550/arXiv.2404.05427. arXiv:2404.05427.

[9] C. S. Xia, Y. Deng, S. Dunn, L. Zhang, Agentless: Demystifying LLM-based software engineering agents, 2024. URL: https://arxiv.org/abs/2407.01489. doi:10.48550/arXiv.2407.01489. arXiv:2407.01489.

[10] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, G. Neubig, OpenHands: An open platform for AI software developers as generalist agents, 2024. URL: https://arxiv.org/abs/2407.16741. doi:10.48550/arXiv.2407.16741. arXiv:2407.16741.

[11] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, H. Ji, Executable code actions elicit better LLM agents, 2024. URL: https://arxiv.org/abs/2402.01030. doi:10.48550/arXiv.2402.01030. arXiv:2402.01030.

[12] J. Yang, C. E. Jimenez, A. L. Zhang, K. Lieret, J. Yang, X. Wu, O. Press, N. Muennighoff, G. Synnaeve, K. R. Narasimhan, D. Yang, S. I. Wang, O. Press, SWE-bench multimodal: Do AI systems generalize to visual software domains?, in: The Thirteenth International Conference on Learning Representations, 2025. URL: https://arxiv.org/abs/2410.03859.

[13] B. Yu, Y. Zhu, P. He, D. Kang, UTBoost: Rigorous evaluation of coding agents on SWE-bench, in: Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Vienna, Austria, 2025, pp. 3762–3774. URL: https://aclanthology.org/2025.acl-long.189/. doi:10.18653/v1/2025.acl-long.189.

[14] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, Y. Cao, ReAct: Synergizing reasoning and acting in language models, 2023. URL: https://openreview.net/forum?id=WE_vluYUL-X. doi:10.48550/arXiv.2210.03629. arXiv:2210.03629.

[15] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, P. Clark, Self-refine: Iterative refinement with self-feedback, 2023. URL: https://arxiv.org/abs/2303.17651. doi:10.48550/arXiv.2303.17651. arXiv:2303.17651.

[16] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, S. Yao, Reflexion: Language agents with verbal reinforcement learning, 2023. URL: https://arxiv.org/abs/2303.11366. doi:10.48550/arXiv.2303.11366. arXiv:2303.11366.

[17] M. Monperrus, Automatic software repair, ACM Computing Surveys 51 (2018) 1–24. URL: https://doi.org/10.1145/3105906. doi:10.1145/3105906.

[18] Q. Wu, et al., AutoGen: Enabling next-gen LLM applications via multi-agent conversation, 2023. URL: https://arxiv.org/abs/2308.08155. doi:10.48550/arXiv.2308.08155. arXiv:2308.08155.

[19] S. Hong, et al., MetaGPT: Meta programming for a multi-agent collaborative framework, 2023. URL: https://arxiv.org/abs/2308.00352. doi:10.48550/arXiv.2308.00352. arXiv:2308.00352.

[20] C. Qian, et al., ChatDev: Communicative agents for software development, in: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Bangkok, Thailand, 2024, pp. 15174–15186. URL: https://aclanthology.org/2024.acl-long.810/. doi:10.18653/v1/2024.acl-long.810.

[21] D. Sculley, et al., Hidden technical debt in machine learning systems, in: Advances in Neural Information Processing Systems 28 (NIPS 2015), 2015, pp. 2503–2511. URL: https://proceedings.neurips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf.

[22] E. Breck, S. Cai, E. Nielsen, M. Salib, D. Sculley, The ML test score: A rubric for ML production readiness and technical debt reduction, in: 2017 IEEE International Conference on Big Data (Big Data), 2017, pp. 1123–1132. URL: https://research.google/pubs/the-ml-test-score-a-rubric-for-ml-production-readiness-and-technical-debt-reduction/. doi:10.1109/BigData.2017.8258038.

[23] S. Amershi, A. Begel, C. Bird, R. DeLine, H. C. Gall, E. Kamar, N. Nagappan, B. Nushi, T. Zimmermann, Software engineering for machine learning: A case study, in: Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, 2019, pp. 291–300. doi:10.1109/ICSE-SEIP.2019.00042.

[24] L. E. Lwakatare, A. Raj, I. Crnkovic, J. Bosch, H. H. Olsson, Large-scale machine learning systems in real-world industrial settings: A review of challenges and solutions, Information and Software Technology 127 (2020) 106368. doi:10.1016/j.infsof.2020.106368.

[25] J. Pineau, P. Vincent-Lamarre, K. Sinha, V. Larivière, A. Beygelzimer, F. d'Alché Buc, E. Fox, H. Larochelle, Improving reproducibility in machine learning research (A report from the NeurIPS 2019 reproducibility program), Journal of Machine Learning Research 22 (2021) 1–20. URL: https://www.jmlr.org/papers/v22/20-303.html.

[26] H. Villamizar, T. Escovedo, M. Kalinowski, Requirements engineering for machine learning: A systematic mapping study, in: 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2021, pp. 29–36. doi:10.1109/SEAA53835.2021.00013.

[27] H. K. M. Haug, et al., How mature is requirements engineering for AI-based systems? a systematic mapping study on practices, challenges, and future research directions, Requirements Engineering (2024). doi:10.1007/s00766-024-00432-3.

[28] G. S. Walia, J. C. Carver, A systematic literature review to identify and classify software requirement errors, Information and Software Technology 51 (2009) 1087–1109. URL: https://doi.org/10.1016/j.infsof.2009.01.004. doi:10.1016/j.infsof.2009.01.004.