Multi-Agent Orchestration for Autonomous Regression Management

Sangwoo Noh[1], Jin Choi[2], Seonghee Yim[3], Youngsik Kim[4]
([1]sangwoo.noh; [2]jjin.choi; [3]sh.yim; [4]ys31.kim@samsung.com)

***Abstract* - The growing complexity of designs and limited compute resources have made regression management a critical bottleneck in modern verification. While regression automation pipelines reduce manual execution, they remain largely linear and execution-centric: failure and coverage signals are often siloed, so a human manager must interpret and relay them instead of having them directly drive the next triage and follow-up decisions. This manager-centric flow increases turnaround time and creates persistent bottlenecks when designs or verification environments change. To address this, we propose an AI-driven closed-loop regression architecture in which each run's outcomes can be fed back into the next decision cycle, enabling adaptive updates to triage actions and, in the full system, to scope, priorities, and scheduling. We realize this flow through multi-agent orchestration: a central Orchestrator coordinates role-specialized LLM agents that execute the triage controller—ownership routing, evidence-backed root-cause explanation, and exact duplicate linkage—under shared, schema-checked evidence bundles and strict guardrails. In a controlled benchmark, we evaluate the triage controller—ownership routing, root-cause explanation, and duplicate linkage—under identical tool access and fixed budgets to isolate architectural effects. The results suggest that role specialization and structured evidence handling can improve triage reliability with modest overhead, providing a practical path toward more adaptive regression management.**

I. INTRODUCTION

As modern digital designs grow increasingly complex, regression management has become the key constraint on verification throughput. Conventional automation pipelines improve repeatability but remain scripted, developer-defined, and effectively linear: regression data is routed through a manager who monitor progress and continuously adjust test scope, configurations, and schedules. This manager-centric workflow creates a throughput bottleneck: root-cause information propagates slowly across owners, delaying issue resolution and triggering repeated reruns to rediscover and validate fixes. Even with automation in place (e.g., Jenkins), most regression flows remain execution-centric: the system can trigger and monitor runs, but it cannot interpret failures, route ownership, or translate outcomes into updated scope, priorities, and schedules.

In regression triage, even small mistakes in reading evidence can cause big delays—for example, missing the key line in a stack trace for blame, extracting the wrong error signature, or linking a failure to the wrong "duplicate" bug ID. These errors often lead to back-and-forth between teams and repeated reruns. Role decomposition helps prevent this by splitting triage into clear steps—evidence extraction, ownership attribution, and exact duplicate-ID linking—so each step can be checked independently with limited, role-specific tool access.

This paper makes four contributions to design verification regression automation. First, we frame regression management as a closed-loop control problem rather than a linear scripted pipeline. Second, we propose a role-decomposed multi-agent orchestration architecture, comprising Planning, Execution, Monitoring, Resource, Debugging, and Notification agents coordinated by a central Orchestrator. Third, we specify an end-to-end design blueprint—including agent I/O definition, tool adapters, and safety guardrails—that can support operation in dynamic regression environments. Fourth, as a controlled and interpretable evaluation, we benchmark the triage controller against procedural and single-agent baselines under identical tool access and fixed budgets, reporting owner assignment accuracy, root-cause quality (similarity and Match@0.5), duplicate linkage micro-F1, and cost metrics.

II. BACKGROUNDS

Closed-loop control system

A closed-loop control system has a few basic parts: goal, state, action and feedback. The **goal** specifies the desired target the system should achieve or maintain. The **state** denotes the system's currently observed condition, typically represented by measured outputs or telemetry signals. The **action** is the intervention applied to the system to move it closer to the goal. The **feedback** path returns the measured state to the controller so it can compare the current condition against the goal and update future actions accordingly.

Design verification regression maps onto the basic parts of a closed-loop system. The goal is defined by sign-off objectives—e.g., schedule milestones, achieving target coverage levels, and meeting quality thresholds such as failure-rate limits. The state is the current observed condition of the regression—coverage closure status, open failure backlog, runtime trends, and resource availability. The feedback is the stream of results returned by each run—failure logs and

signatures, coverage deltas, pass/fail statistics, and queue/resource metrics—informing how far the current state deviates from the goal. The action is the set of control moves taken for the next cycle: selecting or pruning tests, adjusting priorities and scheduling, reallocating compute and licenses, triggering targeted reruns and routing issues to the appropriate owners. Together, these parts form a repeating loop in which each regression iteration updates the next plan based on observed outcomes.

Typically, human regression managers operate this loop as a sequence of scripted steps within a largely linear pipeline. They observe the status of ongoing regression processes, assess progress against the stated goals, triage failing cases, and then adjust test selection, constraints, priorities, and resource usage before launching the next run. However, this reveals a mismatch between the problem structure and the prevailing architecture. Closed-loop regression requires continuous re-planning under non-stationary conditions—frequent design changes, coverage gaps, and fluctuating queue and license availability—yet a static, linear pipeline can only react through manual edits and sequential handoffs. A better way to run this control loop is to make the "controller" distributed and flexible rather than centralized in one manager or one fixed script. In practice, different signals require different expertise—failure triage, coverage closure, and queue/resource management—so specialized agents can each track their own part of the state, share what they learn, and coordinate the next step. Our multi-agent orchestration replaces manual handoffs with tool-driven actions, so test scope, priorities, and scheduling can be updated immediately when new failures, coverage results, or resource constraints appear.

Agentic AI

An agent is an autonomous software entity that is responsible for a specific function, such as planning, execution, monitoring, debugging, or reporting. It considers only the information required for that function and interacts with the rest of the agents through structured inputs and outputs, under explicit policies and guardrails that constrain its actions. An agent consists of: (i) a role and responsibility specification that defines its objectives and decision scope; (ii) optional sub-agents that decompose internal skills (e.g., a log-summarization sub-agent within debugging and reporting); (iii) structured inputs and outputs that enable consistent communication among agents; (iv) tool adapters that safely invoke external systems; (v) an explicit communication topology that defines which upstream and downstream agents may exchange information; and (vi) guardrails that constrain actions and support auditability.

Single-agent systems rely on one generalized agent to interpret requests, maintain state, plan actions, invoke tools, and enforce policies end-to-end. While this design is simple to deploy, it becomes difficult to scale as task variety and system complexity grow. A single agent must accumulate and manage an increasingly large internal memory and context to remain consistent across decisions, which raises maintenance cost and can lead to unstable behavior, including inconsistent responses and hallucinated outputs as the prompt, tools, and policies evolve. Because decisions and actions are centralized, throughput is constrained by sequential processing and the agent becomes a single point of failure. Moreover, combining reasoning, tool execution, and policy enforcement in one component blurs responsibility boundaries, complicating least-privilege control, auditing, and debugging when behavior needs to be explained or corrected.

Multi-agent orchestration fits design verification regression because regression is inherently a closed-loop process that must repeatedly convert outcomes into the next control actions under changing conditions. The loop must handle heterogeneous signals and time scales while adapting to frequent changes in RTL, testbenches, and toolchains, and operating under hard constraints such as limited compute and EDA licenses.

A multi-agent system matches these requirements by assigning each function to a specialized agent with clear interfaces, coordinating their decisions through an orchestrator, and feeding results back to update scope, priorities, scheduling, and resource allocation each cycle.

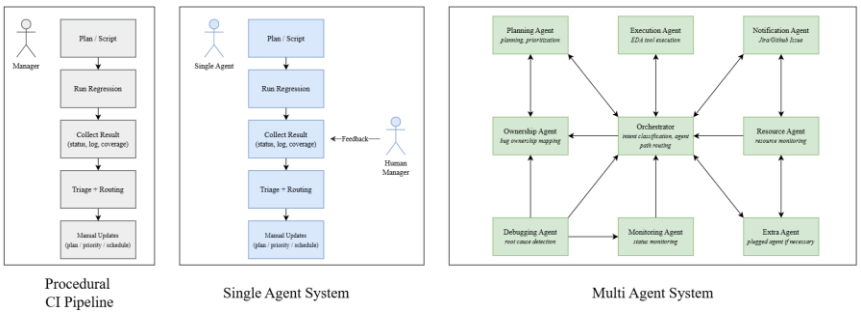

Figure 1. Linear regression pipeline vs Single Agent system vs MAS-based closed-loop regression control system

III. PROPOSED METHODS

In this paper, we propose a multi-agent regression management system to address the challenges of design verification regression. To make this design concrete, we decompose regression management into a small set of role-specific agents, each with a well-defined responsibility and interface. The **Planning Agent** transforms project schedule, live resource status, and history into a deadline-feasible, prioritized test plan; the **Debugging Agent** converts failures into auditable triage outcomes by assigning each case to the most likely owner, identifying and summarizing the root cause based on concrete evidence, and detecting whether the failure is a duplicate of a known issue by linking it to the correct prior identifier; the **Execution Agent** turns that plan into least-privilege, environment-aware runs across heterogeneous tools; the **Monitoring Agent** keep tracks of live states of tests, emitting failure evidence and replan triggers; the **Resource Agent** reports measured capacity signals—such as queue load, license usage, and available compute—to keep plans feasible under changing constraints; and the **Notification Agent** disseminates outcomes to Jira/mail/messaging with access control and default t manager visibility. Coordination arises from typed, schema-checked messages and a shared trace record with logs, commit hashes, and queue/license snapshots; the next sections describe each agent in the same simple format—its sub-agents, inputs, outputs, and strict guardrails.

Orchestration (Orchestrator): goal, inputs/outputs, tools and guardrails

The Orchestrator is the coordination layer that turns a user request and live regression signals into a consistent multi-agent workflow. Its primary functions are intent classification, agent-path selection, and routing. First, it classifies the incoming trigger (e.g., "run a smoke regression," "triage new failures," "re-run duplicates," "coverage gap closure," "resource shock") into a small set of supported intents and determines the minimal set of agents required to satisfy that intent. Second, based on the classified intent and the current system state, it selects an execution path—i.e., an ordered sequence of agent calls with explicit data dependencies (for example, Monitoring → Debugging → Notification for triage-only flows, or Resource → Planning → Execution → Monitoring for run-launch flows). Third, it routes structured messages among agents using typed schemas, ensuring that each downstream agent receives only the fields it needs (least privilege) and that all intermediate artifacts are tracked in a shared trace record. Its outputs are (i) the chosen workflow—i.e., which agents to call and in what order and (ii) the packaged evidence and clear instructions sent to each agent at each step.

Planning Agent: goal, inputs/outputs, tools and guardrails

The Planning Agent serves as the system's decision-making component before a regression run begins, translating open-ended user requests and changing project conditions into a deadline-feasible, value-maximizing regression plan. It jointly considers tape-out/MTO constraints, expected marginal coverage gain and execution cost so that each chosen testcase, seed, and configuration is justified against both schedule and budget. Inputs include the user request and project constraints (milestones, freeze dates, tape-out/MTO), the testcase corpus and history (past outcomes, coverage deltas, recent RTL/UVM changes), plus feedback from (i) a Resource Agent that reports current server/queue and license availability and (ii) the Monitoring Agent that reports per-test status and final results as runs progress. The Planning Agent's main output is a timestamped, prioritized plan—i.e., the planned testcase order with seeds/configs and intended time windows—which it sends to the Execution Agent. Execution turns the plan into jobs, Monitoring publishes live state and outcomes back to Planning, and Resource continuously updates feasibility constraints, enabling Planning to re-order or adjust scope when conditions change. The Planning Agent must not produce an infeasible plan—every plan must fit deadlines and resource/license limits, or it must return a clear refusal with reasons.

Debugging Agent: goal, sub-agents, inputs/outputs, tools and guardrails

The Debugging Agent performs regression triage by turning each failing case into two key outputs: a clear root-cause explanation and a duplicate link (if the failure is already known). To keep the logic simple and reliable, the Debugging Agent is split into two sub-agents: a Root-Cause Agent and a Duplicate Agent. Its inputs include a bounded log snippet, the main error message, and testcase metadata (e.g., test ID, seed/config, feature tags, and related block/IP). In addition, it can pull live status and failure evidence from the Monitoring Agent, including the current testcase state and pointers (URIs) to run artifacts such as full logs or waveforms. The Debugging Agent's outputs are a structured triage record that includes (1) a short root-cause text, (2) either no-duplicate or duplicate_of=<ID>, and (3) a brief rationale that points to the exact evidence used. When the triage outcome implies that a targeted re-run is useful—for example, to confirm that a suspected duplicate has been resolved or that a mitigation changes the failure signature—the Debugging Agent also emits an actionable rerun request to the Execution Agent (e.g., testcase ID, recommended seed/config/flags, and the evidence-based reason), enabling the system to validate the diagnosis and update status accordingly.

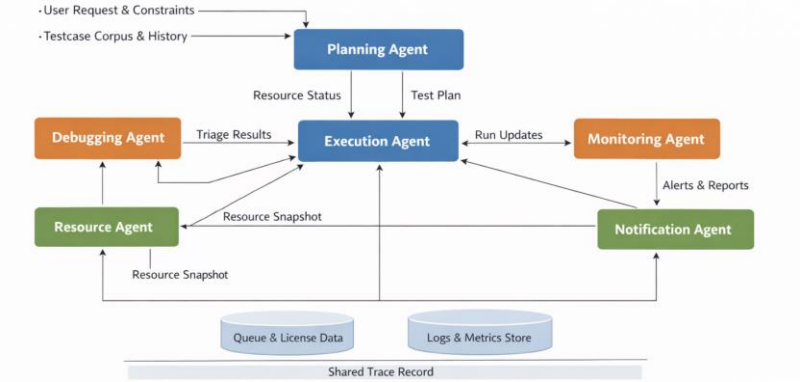

Figure 2. Architecture of Multi-Agent Regression Management System

The Root-Cause Agent writes a short diagnosis based on the evidence, while keeping important identifiers like error codes, file paths, and config keys so the result stays reproducible. The Duplicate Agent checks whether the case matches a past failure and, if so, links it to the correct historical ID. It does this by querying a failure history database and comparing key signatures (error code, file path, test name) rather than relying only on "similar meaning."

The Debugging Agent then merges the two sub-agent outputs into one final answer, making sure the duplicate link and the root-cause text do not contradict each other. The Debugging Agent uses a small set of safe tools: a history search tool (e.g., db_query) to find similar past failures and candidate IDs, and light search/parsing tools (e.g., grep-style pattern search) to extract key signatures from logs. Guardrails are strict: it must not guess. Every diagnosis or duplicate link must cite concrete artifacts (tool results or specific log lines). If evidence is weak, the agent can abstain and return "not enough evidence" instead of making up an answer, and it always leaves an audit trail so a human can trace how the decision was made. The Debugging Agent must not guess—every owner/root-cause/duplicate decision must be backed by concrete evidence (tool outputs or specific log lines), otherwise it abstains.

Execution Agent: goal, inputs/outputs, tools and guardrails

The Execution Agent turns the plan into real runs on different backends, while keeping strict safety limits. Its inputs are (1) the ordered testcase list from the Planning Agent (with seeds/configs and intended time windows) and (2) targeted rerun requests from the Debugging Agent (for example, rerun a testcase with a specific seed or flag to confirm a suspected duplicate or check a fix). Using these inputs, the agent builds the exact run commands by reading approved tool guides and environment descriptions. It standardizes common details such as flags, paths, license use, and seed handling, and it builds commands differently depending on the run mode (simulation vs. emulation). The key point is that the tool guide acts like a "how-to-run" rulebook for each backend, so the agent can create correct commands across different regression tools without being tied to one specific vendor workflow. Its guardrail is strict design verification scope: unauthorized commands, privileged shell actions, or ad-hoc system modifications are refused and logged, and all invocations are validated against tool guides prior to dispatch.

Monitoring Agent: goal, tools and guardrails

Monitoring Agent maintains the live status of regression. It continuously updates the state of each testcase—queued, running, passed, failed, or error-prone—and detects when an error has occurred. When a failure is observed, it emits a standardized failure event with minimal evidence and forwards it to the Debugging Agent. When a previously failing testcase turns green after a re-run, it marks the status accordingly and notifies the Planning Agent so the case can be re-scheduled if needed. It also aggregates simple progress views at suite and milestone levels to show how far the run is from plan.

The functionality of tools used for Monitoring Agent includes reading from scheduler and workflow status APIs, consumes execution telemetry streams, and tails bounded log excerpts for first-error signals. It writes structured status records to a metrics store and exposes read APIs and dashboards for humans and peer agents. All heavy artifacts remain in external storage and are referenced by stable URIs.

Monitoring Agent is strictly read-only with respect to artifacts and execution state; it does not mutate logs, change job configuration, or infer remedies. It emits signals rather than prescriptions, attaches timestamps and provenance to every update, and rate-limits duplicate events. If upstream data are incomplete, it degrades to best-effort reporting without fabricating status.

Resource Agent: goal, inputs/outputs tools and guardrails

The Resource Agent provides the Planning Agent with a reliable view of current capacity. It takes as input the live status and run signals collected by the Monitoring Agent, including queue state, license usage, and lightweight run metadata that reflects how the simulator or emulator is behaving. When the Planning Agent requests an update, the Resource Agent outputs a resource snapshot that summarizes what is available now—such as free compute slots, expected queue latency, and remaining license capacity—so Planning can decide whether to start, delay, or scale down parts of the regression plan. Its guardrail is to report only measured state: it must not guess future availability, and it must attach a timestamp (or snapshot ID) so the Planning Agent can avoid making decisions based on stale resource information.

Notification Agent: goal, inputs/outputs, tools and guardrails

Notification Agent ensures that results reach the right stakeholders, on the right channels, with the right policy. It packages execution summaries, debug diagnoses, and status changes for delivery via JIRA, email, and messaging sub-agents, selecting recipients from ownership maps and watch lists and enforcing access control with manager visibility by default. Its inputs are reportable payloads emitted by Debug and Execution, augmented with audience context; its outputs are delivery receipts, message IDs, and links that are written back to the system's decision traceability record for end-to-end traceability. The agent's tooling consists of mail sender, messenger sender, and Jira post/update interfaces. It depends on Debugging Agent and Execution Agent and closes the loop by confirming dissemination. The guardrails are clear: distribute design verification -relevant content only, respect authorization rigorously, and avoid overexposure by scoping messages to need-to-know audiences while ensuring managerial oversight.

IV. EXPERIMENT SETUP

A. Scenarios

The proposed architecture includes Planning, Debugging, Execution, Monitoring, Resource and Notification agents as an end-to-end regression management system. However, for the purpose of a controlled and interpretable simulation benchmark, we restrict the experimental scope to the triage sub-system, because it directly determines the three outcomes reported in our evaluation and can be measured under fixed evidence and tool access. In practical regression flows, triage is often the recurring bottleneck, and in particular **ownership routing**, **root-cause diagnosis**, and **duplicate detection** act as high-leverage decision gates: misrouting causes cross-team ping-pong, weak diagnoses slow convergence, and missed duplicates create redundant work and bug noise. Planning, Execution, Monitoring, Resource and Notification remain part of the overall system design, but in this benchmark, they are treated as out-of-scope infrastructure—either held constant or absorbed into dataset generation and evidence packaging—so that measured differences reflect architectural coordination and evidence handling rather than environment dynamics. Accordingly, even under an agent-reduced benchmark, we retain an explicit Ownership Agent and a Debugging Agent (Root-Cause + Duplicate) to isolate these bottleneck decisions under fixed evidence and bounded tool access. The benchmark instantiates two role-specific agents: an Ownership Agent, which is responsible for assigning each failure to the most likely owner, and a Debugging Agent, which decomposes into a Root-Cause sub-agent and a Duplicate sub-agent to produce an evidence-backed diagnosis and a case-level duplicate linkage decision.

Scenario 1. Owner Assignment

Scenario 1 models the ownership-routing stage of design verification regression triage, in which the system receives a failure case (log snippet plus metadata such as file paths, error codes, and test identifiers) and must determine the most appropriate responsible owner. The scenario assumes access to standard engineering evidence sources—pattern search over relevant code or log artifacts (e.g., grep), code-attribution context (e.g., git blame for recently modified lines), and a historical failure database query interface (e.g., db_query)—and emphasizes that ownership decisions should be traceable to concrete evidence rather than implicit intuition. While the scenario highlights grep, git blame, and db_query as representative evidence sources, these tools are illustrative examples; in practice, additional regression-flow-specific tools can be plugged in to enrich evidence collection and traceability, such as Excel-format files (e.g., .xlsx) and document/wiki repositories. The setting mirrors real triage situations where the same failure pattern can reasonably point to more than one module or team. In this case, the system must turn unclear or noisy clues into one responsible owner, while staying within limits on tool calls and runtime.

Scenario 2. Root-Cause Explanation

Scenario 2 models the diagnosis step of regression triage. The goal is not only to route the failure, but to write a short and clear root-cause explanation that helps a verifier understand what likely went wrong and what to check next. Each case contains mixed and incomplete clues—such as stack traces, assertion messages, compile errors,

configuration warnings, and sometimes intermittent (flaky) behavior—so the system must form a plausible explanation from the available evidence. The explanation must be easy to compare with the expert ground truth, so it should keep key identifiers unchanged (e.g., file paths, error codes, config keys, and test names) and avoid vague wording. It is scored using Jaccard-based similarity and a Match@0.5 threshold, so the system should state the suspected cause in a consistent, concrete form and point to which retrieved artifacts support it, within limited tool calls and runtime.

Scenario 3. Duplicate Detection

Scenario 3 models the duplicate-handling step of triage. For each new failure, the system must decide whether it is a genuinely new issue or a repeat of a previously seen one. If it is a duplicate, the system must link the case to the correct canonical reference (e.g., a duplicate_of identifier); if it is not a duplicate, it should explicitly output "no duplicate" (or an equivalent null link). The scenario assumes access to a historical case store (e.g., a database or files) that contains past failure signatures, short summaries, and canonical identifiers that can be retrieved and compared, including common recurring patterns such as repeated assertion failures (e.g., the same ASSERT_* message or check name), frequent compile/elaboration errors (e.g., missing module/symbol), and known configuration-related failures (e.g., a recurring missing key or mismatched parameter). Duplicates are relatively rare, and the link must match the correct prior ID exactly. We evaluate performance with duplicate-detection micro-F1, so the system must avoid both missed duplicates (which cause repeated triage and bug noise) and wrong links (which can hide new issues that only look similar). As with other scenarios, the system runs under limited tool calls and time, and should base its decision on retrieved evidence (e.g., the closest past cases and the specific features that support the link).

B. Dataset and Ground Truth

We use a synthetic triage dataset of 500 design verification failure cases that mimic common regression categories such as RTL functional failures (e.g., assertion/scoreboard mismatches), UVM/infrastructure issues (e.g., sequence/agent configuration or factory override problems), build/compile failures (e.g., missing package, type errors, or undefined symbols), and flaky/intermittent tests (e.g., seed-dependent timeouts or nondeterministic ordering). Each case contains (i) a log snippet, (ii) metadata that typically appears in triage (e.g., test identifier, error code, file path/module hints, and run context), and expert-labeled ground truth for (a) responsible owner, (b) a short root-cause summary, and (c) duplicate linkage (duplicate_of ID when applicable).

We split the 500 cases into a history pool (n=350) and a held-out evaluation set (n=150). The history pool is used only to populate the simulated "past cases" store that tools like db_query can retrieve from, so Agents can search prior signatures/summaries/IDs without ever seeing evaluation labels. The evaluation set is kept out of this store and is used exclusively for scoring after execution. In the evaluation set, 39/150 cases (26.0%) are duplicates (duplicate_of $\neq$ null) and 111/150 (74.0%) are non-duplicates. This split prevents information leakage: systems must infer owner/root cause/duplicate links from evidence and retrieved history matches, not from access to the answer key.

For transparency and interpretability, we also report dataset statistics that directly affect task difficulty and the meaning of the metrics. These include the number of distinct owner labels, the number of canonical duplicate IDs, and the duplicate cluster-size distribution (e.g., how many issues recur 2 times versus 3+ times). We further report the category breakdown (RTL, UVM infrastructure, build/compile, flaky) and evidence-field coverage rates (the fraction of cases that contain file paths, error tokens, and configuration keys). Together, these statistics clarify how hard owner routing and exact duplicate linking are, and they help interpret root-cause scoring based on token-overlap metrics such as Jaccard similarity and Match@0.5.

C. Tooling and Resource Budgets

All systems access an identical deterministic MockToolRegistry that emulates common triage tools: grep(pattern, file), git_blame(file, line), and db_query(query). Tools use fixed random seeds so identical inputs yield identical outputs, removing tool nondeterminism as a confounder. Agent-based systems are constrained to the same per-case budgets: up to 50 simulated Agent invocations, up to 100,000 tokens, up to 300 seconds wall time, and up to 100 tool invocations. The legacy CI baseline consumes no Agent invocations.

D. Evaluation Metrics

We report five primary metrics: Owner Assignment Accuracy (exact match), Root Cause Similarity (Jaccard similarity between token sets), Root Cause Match@0.5 (binary: similarity >= 0.5), Duplicate Detection micro-F1 (true positive only if the system both flags a case as duplicate and names the correct duplicate_of identifier), and cost metrics (average time per case and average simulated Agent invocations per case). The evaluation set contains 26.0% positives for duplicate linkage (39/150), so F1 is preferable to accuracy.

Root-cause text normalization is deterministic: we lowercase, tokenize by whitespace, perform no stemming or stopword removal, and keep digits, file paths, and error codes intact to preserve rare identifiers. Because Jaccard similarity is bounded in [0,1] and can be strongly non-normal (mass near 0 or 1), we report mean and standard deviation as descriptive dispersion summaries rather than assuming a Gaussian error model.

Abstention handling. If a system abstains for owner or root-cause prediction under guardrails, we score that output as incorrect for the corresponding metric. For duplicates, abstention is treated as no duplicate link (negative prediction).

E. Statistical Analysis

Because all systems are evaluated on the same 150 cases, we use paired statistical tests throughout. For paired binary outcomes (Owner Accuracy and Root Cause Match@0.5), we apply a two-sided McNemar's test based on cases where the two systems disagree. For continuous outcomes (Time per Case), we use paired t-tests on per-case time differences. We report 95% confidence intervals for paired deltas using case-level paired bootstrap resampling over the evaluation set. To control the family-wise error rate across seven pre-specified comparisons, we apply a Bonferroni correction, setting the significance threshold to 0.0071 (0.05/7). Duplicate micro-F1 is reported descriptively in this version; future work will add paired bootstrap confidence intervals and duplicate confusion counts (TP/FP/FN).

V. Results

We benchmarked three approaches to regression triage in design verification: a legacy procedural CI script, a monolithic single-agent architecture, and a multi-agent architecture. Using a triage dataset (N=500; evaluation set n=150) with expert-annotated ground truth for (i) owner assignment, (ii) root-cause explanation, and (iii) duplicate detection, we compared accuracy, diagnostic similarity, duplicate merging quality, and resource cost under identical tool access and execution budgets. On the evaluation set, the multi-agent system improved owner assignment accuracy from 63.3% to 76.7% (+13.3 percentage points) and root-cause match@0.5 from 62.7% to 77.3% (+14.7pp). The largest gain was in duplicate detection (micro-F1 40.8% to 96.0%). Multi-agent triage incurred modest additional cost (4.04s and 4.0 simulated Agent invocations per case vs. 3.05s and 3.0 invocations for the single-agent). Paired significance testing showed improvements significant at alpha=0.05, but not all survived Bonferroni correction (alpha=0.0071) for multi-agent vs. single-agent comparisons.

Table 1 summarizes results on the held-out evaluation set (n=150). Both Agent-based systems substantially outperform the legacy procedural CI baseline across triage quality metrics: owner accuracy rises from 28.7% (legacy) to 63.3% (single-agent) and 76.7% (multi-agent), and root-cause quality improves from near-zero overlap for legacy (1.9% mean similarity; Match@0.5 = 0.0%) to 62.7%/62.7% (single-agent) and 76.9%/77.3% (multi-agent). The largest gain is in duplicate detection, where micro-F1 increases from 22.7% (legacy) to 40.8% (single-agent) and then to 96.0% (multi-agent), suggesting that role specialization and structured evidence aggregation are particularly effective for exact duplicate linking. These accuracy improvements come with moderate cost: compared to the single-agent, the multi-agent adds +0.99s per case on average (3.05s → 4.04s, ~+32%) and +1.0 simulated Agent invocation (3.0 → 4.0), while the legacy baseline remains fast (0.10s) largely because it applies lightweight procedural heuristics rather than evidence-driven diagnosis.

The legacy CI performs particularly poorly on root-cause scoring (Root Cause Match@0.5 = 0.0%), primarily because it does not produce a structured, human-readable root-cause summary like the ground-truth explanations; it mostly prints raw error lines or coarse tags. Instead, the procedural baseline typically surfaces raw failure artifacts (e.g., a first error line, a short log excerpt, or a coarse failure tag such as "compile failed"), which may contain a few overlapping tokens (yielding a small non-zero mean similarity) but rarely share enough explanatory content to exceed the Match@0.5 threshold. More broadly, the legacy baseline relies on fixed pattern matching and shallow heuristics without structured evidence aggregation across sources, making it brittle under ambiguous signatures and strict duplicate-ID linking, and leading to consistently lower scores across triage metrics.

Metric	Legacy CI	Single-Agent	Multi-Agent
Owner Accuracy	28.7%	63.3%	76.7%
Root Cause Similarity (mean ± SD)	1.9% ± 5.8	62.7% ± 48.5	76.9% ± 41.7
Root Cause Match@0.5	0.0%	62.7%	77.3%
Duplicate Detection micro-F1	22.7%	40.8%	96.0%
Avg Time per Case	0.10s	3.05s	4.04s
Avg simulated Agent invocations	0.0	3.0	4.0

Table 1. Overall metrics on the held-out evaluation set (n=150)

Table 2 reports paired statistical comparisons on the same 150 evaluation cases. Because each system is evaluated on identical cases, we use paired tests to assess whether observed differences are consistent beyond sampling noise. For binary outcomes (Owner Accuracy and Root Cause Match@0.5), we apply a two-sided McNemar test; for per-case runtime, we use a paired t-test. We report 95% confidence intervals for deltas using case-level paired bootstrap resampling, and we apply Bonferroni correction over the seven pre-specified comparisons (α=0.05/7=0.0071). Under α=0.05, the multi-agent system shows significant gains over the single-agent baseline in Owner Accuracy (Δ=+13.3pp, p=0.0142) and Root Cause Match@0.5 (Δ=+14.7pp, p=0.0077), but they do not meet the conservative Bonferroni-adjusted threshold under the current sample size. We therefore treat multi-agent gains over single-agent as suggestive evidence and prioritize larger-scale and pre-registered evaluations in future work. Future releases will report the full duplicate confusion counts (TP/FP/FN) and paired bootstrap confidence intervals for micro-F1 to attribute gains to precision vs. recall and to enable statistically grounded comparisons.

In contrast, both Agent-based approaches remain strongly significant relative to the legacy baseline after correction. Runtime differences are also statistically reliable: the multi-agent system is slower than the single-agent system by +0.983s per case on average (p<0.0001), reflecting a measurable overhead associated with additional role-specialized processing.

Comparison / Metric	System 1 → System 2	p-value	Sig. @0.0071
Multi vs Single - Owner Accuracy	63.3% (95/150) → 76.7% (115/150)	0.0142	No
Multi vs Single - Root Cause Match@0.5	62.7% (94/150) → 77.3% (116/150)	0.0077	No
Multi vs Single - Time per Case	3.053s → 4.037s	<0.0001	Yes
Single vs Legacy - Owner Accuracy	28.7% (43/150) → 63.3% (95/150)	<0.0001	Yes
Single vs Legacy - Root Cause Match@0.5	0.0% (0/150) → 62.7% (94/150)	<0.0001	Yes
Multi vs Legacy - Owner Accuracy	28.7% (43/150) → 76.7% (115/150)	<0.0001	Yes
Multi vs Legacy - Root Cause Match@0.5	0.0% (0/150) → 77.3% (116/150)	<0.0001	Yes

Table 2. Paired statistical tests. Bonferroni-corrected threshold is alpha=0.0071 (7 comparisons).

Tables 1–2 show that the multi-agent design improves triage quality by handling evidence extraction and history matching more reliably, with the largest gain in duplicate detection (micro-F1 40.8% → 96.0%). Owner accuracy and root-cause Match@0.5 also increase (63.3% → 76.7%, 62.7% → 77.3%), but these gains are weaker under strict Bonferroni correction at the current sample size, while the runtime overhead is clear (+0.99s per case and +1 simulated agent invocation). We treat this as a quality–cost trade-off: the extra time may be justified if improved duplicate linking reduces repeated triage work and bug-noise at scale. Because duplicate micro-F1 requires exact canonical-ID linkage, future releases will additionally report duplicate confusion counts (TP/FP/FN) and paired bootstrap confidence intervals to separate precision and recall effects. This benchmark is a controlled simulation, and we limit claims to held-out evaluation performance under fixed tool access, fixed budgets, and no access to ground truth during execution, so observed differences reflect architectural coordination rather than additional information or tuning.

VI. Conclusion

In conclusion, this paper proposes a role-decomposed multi-agent architecture for design verification regression management and evaluates its triage core in a controlled benchmark designed to isolate architectural effects from operational noise. Although the full system includes Planning, Execution, Monitoring, Resource, and Notification components, our study focuses on the triage controller because it produces the most decision-critical outputs for practitioners: ownership routing, root-cause explanation, and duplicate linkage. Across these tasks, the results support the central hypothesis that specialization and structured coordination—through typed message interfaces, shared evidence bundles, and evidence-based guardrails—can improve triage reliability even when tool access, budgets, and base agent behavior are held constant. At the same time, the evaluation is a deterministic simulation with a restricted scope, so the findings should be interpreted as controlled evidence of architectural benefit rather than a direct claim about absolute performance in live regressions.

Future work will (i) reintegrate the triage controller into the full closed-loop regression system so that monitoring signals, adaptive re-runs, and resource-aware planning can jointly drive outcomes; (ii) validate the approach on real regression logs and workflows under realistic nondeterminism, tool failures, and organizational constraints; and (iii) expand the benchmark methodology by increasing evaluation-set size, pre-registering primary endpoints, and publishing full duplicate confusion counts (TP/FP/FN) to remove ambiguity in micro-F1 interpretation. We also plan to evaluate end-to-end impact measures that matter operationally—time-to-resolution, duplicate suppression and bug-noise reduction, and the overall cost–benefit trade-off under practical compute and latency limits.

REFERENCES

[1] Tang, Shuo, et al. "Decentralized and Lifelong-Adaptive Multi-Agent Collaborative Learning." arXiv preprint arXiv:2403.06535 (2024).
[2] Liu, Wei, et al. "Autonomous agents for collaborative task under information asymmetry." Advances in Neural Information Processing Systems 37 (2024): 2734-2765.
[3] Wu, Qingyun, et al. "Autogen: Enabling next-gen Agent applications via multi-agent conversations." First Conference on Language Modeling. 2024.
[4] Guo, Taicheng, et al. "Large language model based multi-agents: A survey of progress and challenges." arXiv preprint arXiv:2402.01680 (2024).