Computer Fraud and Security
ISSN (online): 1873-7056

Autonomous Evaluation Architectures: Multi-Agent LLM Pipelines, Browser-Grounded Testing: Programmatic Alignment via DSPy, and Adversarial Robustness in Production Orchestration Systems

Venkata Chandra Sekhar Sastry Chilkuri

Cloud-Native Data Engineering and AI Platform Architecture

Independent researcher, USA

Abstract

Evaluating multi-agent large language model systems requires fundamentally different approaches than evaluating single-model outputs. Conventional benchmarks assess isolated model capabilities in controlled conditions, but production multi-agent pipelines exhibit emergent failure modes that only manifest through agent interactions across pipeline stages. An individual agent may produce valid output that, when consumed by a downstream agent, leads to semantically incorrect or structurally broken final artifacts, a class of failures that per-agent evaluation cannot detect by design. This article introduces AgentForge-Eval, a closed-loop evaluation architecture that combines browser-grounded execution testing, multi-layer deterministic and semantic assertion frameworks, and programmatic prompt alignment to autonomously detect, diagnose, and remediate multi-agent failures. Unlike static benchmarks that assess what models produce, AgentForge-Eval tests what multi-agent outputs actually do by executing generated artifacts in headless browser environments and feeding runtime results back into an iterative fix loop with formal convergence guarantees. Deployment in a production multi-agent pipeline demonstrates substantial improvements in first-pass acceptance rates, significant reductions in iterations required before approval, and detection of a materially larger share of failures than semantic judge evaluation captures alone. Programmatic optimization using the full evaluation stack as its objective achieves additional composite metric gains through automated cross-stage prompt alignment. The framework contributes a formal taxonomy of multi-agent failure modes and empirical evidence that browser-grounded evaluation captures a failure class that proxy-metric assessment cannot reach.

Keywords: Multi-Agent Evaluation, Browser-Grounded Testing, Playwright, PromptFoo, DSPy, Programmatic Alignment, LLM-as-Judge, Adversarial Robustness, LangGraph, Closed-Loop Refinement

1. Why Multi-Agent Evaluation is Fundamentally Different

Measuring the quality of individual language model outputs and measuring the quality of multi-agent systems are two fundamentally different problems, and the distance between them has become one of the most consequential gaps in production artificial intelligence today. Conventional evaluation paradigms, designed for isolated model assessment, measure capabilities such as factual recall, code synthesis, and conversational fluency within tightly scoped test environments. These approaches produce reliable scores for single-model performance but offer no visibility into how quality degrades, propagates, or compounds once multiple agents begin exchanging state across pipeline boundaries [1].

Production multi-agent systems are qualitatively different from the sum of their individual components. A pipeline may route an incoming request through a classification agent, a planning agent, a synthesis agent, and a validation agent in sequence.

When the generated artifact is executed in a browser, it crashes because the planner encoded a relational pattern that the synthesizer's code generation strategy cannot accommodate. This class of failure exists only at the boundary between agents and is entirely invisible to per-agent evaluation strategies [2].

Multi-agent systems exhibit four distinct failure categories that have no equivalent in single-model evaluation. The first of these is the inter-agent semantic gap, a condition where outputs from one agent are structurally acceptable to the next agent in the pipeline yet carry semantic meaning that the receiving agent interprets incorrectly, with the resulting misalignment becoming visible only several stages later. A planning agent may produce a valid JSON specification while

embedding field naming conventions that a synthesis agent interprets differently, resulting in broken references that no individual agent evaluation would catch.

The third failure category involves model heterogeneity artifacts. Enterprise pipelines frequently route different stages through different language model providers as a cost optimization strategy, using more capable and expensive models for planning while routing synthesis to faster, lower-cost alternatives. When models with different output distributions interact across pipeline stages, subtle mismatches accumulate at agent boundaries, producing quality degradation patterns that are difficult to attribute to any single provider. The fourth category, temporal coherence decay, affects iterative refinement pipelines where human feedback triggers successive regeneration cycles. As context windows grow with each iteration, earlier context undergoes compression or truncation, and agents in later iterations produce outputs that are increasingly disconnected from the original intent [1][2].

Existing evaluation frameworks address none of these emergent patterns adequately. Static benchmark suites measure task completion in isolated scenarios and produce scores that do not predict pipeline-level quality. Per-agent unit testing confirms individual agent behavior but cannot detect interaction failures. Human evaluation is neither reproducible nor scalable enough to integrate into continuous deployment pipelines. Semantic evaluation using a language model as a judge assesses output quality against rubrics but cannot determine whether a generated artifact executes correctly when deployed in a real environment. The fundamental limitation shared by all these approaches is that they assess what agents produce rather than what those outputs actually do when consumed by downstream systems or deployed into production environments. AgentForge-Eval addresses these limitations through a closed-loop evaluation architecture that combines deterministic structural validation, semantic rubric-based assessment, browser-grounded execution testing, automated root cause diagnosis, and programmatic prompt alignment.

Failure Category	Characteristic
Inter-Agent Semantic Gap	Syntactically compatible outputs carry misaligned semantic meaning across pipeline boundaries
State Accumulation Drift	Minor per-agent deviations compound into structural inconsistencies across pipeline stages
Model Heterogeneity Artifacts	Provider switching mid-pipeline produces quality degradation patterns at agent boundaries
Temporal Coherence Decay	Growing context windows cause later refinement iterations to diverge from original specification

Table 1: Multi-Agent Failure Categories and Evaluation Gaps [1, 2]

2. Related Work

Language model evaluation has historically been built around static benchmark suites that isolate specific model capabilities and measure performance under controlled, reproducible conditions. Code generation benchmarks moved the field in a meaningful direction by shifting the evaluation criterion from surface-level text similarity to functional execution, measuring whether generated programs actually pass predefined test cases rather than whether they resemble reference solutions [3]. This was a genuine methodological advance, but its scope remained limited to single-model outputs evaluated against well-specified tasks. The software engineering tradition extended this execution-based logic further by testing whether language models could resolve real repository issues drawn from active codebases, demanding that models understand system context rather than just syntactic patterns [11]. Yet even this more demanding evaluation setting targets individual model performance on isolated tasks. Neither benchmark tradition accounts for what happens when multiple models coordinate across a shared pipeline, where the quality of the final output depends not just on each model's individual capability but on how their outputs interact across stage boundaries.

The theoretical literature on multi-agent collaboration has begun mapping the coordination dynamics that make pipeline-level quality so difficult to predict from per-agent evaluations alone [1]. Coordination protocols, shared state management strategies, and task decomposition approaches have been identified as the primary structural determinants of

multi-agent system quality, yet the same literature acknowledges that systematic evaluation frameworks for these properties remain largely absent from current practice. Agentic software engineering as a maturing discipline has approached this problem from the engineering side, cataloguing the methods by which language models are embedded into software development workflows and identifying test generation, code repair, and specification alignment as the core competencies that production-grade agent systems must reliably demonstrate [2]. These contributions establish the conceptual vocabulary and technical foundations that inform the evaluation architecture developed in this work, even as they stop short of providing an integrated framework for assessing multi-agent pipeline quality in production settings.

Knowledge-grounded evaluation offers a distinct perspective on what quality assessment should measure in language model systems [4]. Evaluating model outputs by measuring how closely they resemble training data distributions tells very little about whether those outputs are actually correct in a domain sense. Knowledge-based evaluation takes a different position, checking outputs against established conceptual frameworks for the domain in question rather than against statistical proximity to reference examples. For evaluation design, this distinction carries a direct consequence: rubrics built around generic quality indicators will consistently miss domain-specific correctness failures that only become visible when the evaluation criteria reflect genuine domain knowledge. The semantic rubric architecture in Section 3.3 is built on this principle, with each evaluation dimension tied to a domain-relevant correctness criterion rather than a surface-level quality signal. Automated program repair research has established a set of empirical findings that directly bear on the design of iterative remediation systems for language model pipelines. Language models demonstrate meaningful repair capability when provided with structured failure context that precisely identifies the failing assertion, the expected behavior, and the relevant code section, but their effectiveness drops substantially when failure information is incomplete or ambiguous [8]. The relationship between failure context specificity and repair success rate points to structured error enrichment as a prerequisite for effective automated remediation rather than an optional enhancement. Convergence behavior in iterative repair processes has also been characterized, with findings indicating that repair cycles either converge reliably within a small number of iterations or enter oscillation patterns where successive attempts cycle between different failure states without making progress [10]. Understanding these convergence dynamics is essential for designing repair loops with meaningful termination guarantees rather than arbitrary iteration limits.

Taken together, the existing literature leaves a specific and consequential gap unaddressed. No published framework combines browser-grounded execution testing for multi-agent pipeline outputs with a multi-layer assertion architecture spanning deterministic, semantic, and behavioral validation; integrated automated root cause diagnosis across pipeline stages; and programmatic prompt alignment that uses the complete evaluation stack as its optimization objective. AgentForge-Eval addresses this gap by treating evaluation not as a quality checkpoint applied after pipeline execution but as an active infrastructure component that continuously monitors behavior, diagnoses failure origins, and drives systematic improvement across the full pipeline.

3. AgentForge-Eval Architecture

3.1 System Overview

The implementation of AgentForge-Eval is in the form of a LangGraph subgraph that is meant to be composable with any multi-agent pipeline architecture. The framework takes as inputs the pipeline outputs in specific evaluation checkpoints and directs them to five successively more computationally expensive, semantically complete evaluation layers [1][2]. The ordering reflects a deliberate cost management strategy: cheaper deterministic checks run first, short-circuiting the pipeline before computationally intensive browser-grounded tests are invoked on artifacts that contain obvious structural violations

The complete evaluation stack is a typed LangGraph subgraph, where layers are conditionally mutually routed, long-running evaluations may be cancelled, and state is managed in a structured manner to maintain evaluation history throughout the entire pipeline execution.

Evaluation Layer	Primary Function
Layer 1: Deterministic Structural Validation	Thirty-plus JavaScript assertion functions checking schema compliance and referential integrity

Layer 2: LLM-as-Judge Semantic Evaluation	Five rubric dimensions assessed at temperature 0.0 for reproducible quality scoring
Layer 3: Browser-Grounded Execution Testing	Playwright headless Chromium executing generated artifacts under realistic deployment conditions
Layer 4: Root Cause Diagnosis	Cross-stage failure attribution mapping violations to specific pipeline agents

Table 2: AgentForge-Eval Five-Layer Evaluation Stack [3, 9]

3.2 Layer 1: Deterministic Structural Validation

The deterministic validation layer gives the quality gate of the evaluation stack, running more than thirty JavaScript assertion functions over pipeline output before any computationally expensive evaluation is run [3]. The evaluation suites have a different level of granularity to be able to support the spectrum of complexity of artifacts generated by the pipeline. The constrained suite of tests has fifteen test cases against artifacts that have a tight structure, such as a limited number of entities, shallow module structure, and strict naming conventions. The extended suite has sixteen test cases, which are complex artifacts with relaxed structural parameters, which allow more entities, multi-level relationships, and other cross-reference validation requirements.

The completeness of structures checks that the schema is compliant with JSON and that all the necessary fields are present by the specification of the artifact. Entity model validation checks include bound counting against limits, checks on attribute type correctness, and naming conventions using camelCase patterns across the board. Referential integrity validation determines circular references by depth-first graph traversal and dangling references in which entities refer to nonexistent targets. Consistency validation ensures that cross-module naming coincides and ensures that implementation structures are consistent with the planning specification. Domain-specific rules are used to deal with artifact-level issues such as form unification, where each entity has a single form definition, and priority field validation, where priority assignments are within anticipated ranges within the context of the domain of the artifact.

Each claim generates an encoded object of the result encoding pass or fail, the severity of a violation as an error, a warning, or an informational event, the exact position of the violation in the artifact tree, and a machine-readable violation code. These violation codes are processed as structured inputs to the root cause diagnosis layer to allow the systematic assignment of structural failures to specific pipeline stages without the necessity of manually inspecting raw assertion output.

3.3 Layer 2: LLM-as-Judge Semantic Evaluation

The semantic evaluation layer deals with qualitative aspects of the quality of the artifacts, which cannot be expressed by deterministic validators using rule-based assertions [4]. A language model used as a judge measures the output of pipelines on five dimensions of the rubric that integrate the key issues of quality of production and multi-agent systems. Data model normalization measures the existence of an appropriate decomposition of entities and the absence of redundancy between the artifact structure. Domain knowledge considers how well the data model is a true representation of the target domain and reveals instances where a superficially acceptable structure could not represent domain semantics appropriately. The structural reasoning coherence looks at whether the planning justification made by the planning agent makes sense in terms of the architectural decisions captured by the artifact that is generated. Design appropriateness evaluates the suitability of the artifact in terms of the interaction pattern and structures of the interfaces. Attribute type accuracy checks that field type assignments make sense in the domain scenario and prevents instances where a currency field is assigned as a string or a temporal field as an integer.

The output of each of the rubric dimensions is a numeric grade on a five-point scale with rubric-corresponding justification text that outlines the exact characteristics of the artifact that result in the allocated grade. The judge model is a temperature 0.0 model that minimizes evaluation-run variance, making repeated evaluations of the same artifact yield consistent scores. Provider configuration can be used to separate the generation and evaluation issues: the generation models use temperature 0.3 to enable creative latitude in the construction of artifacts, whereas the evaluation models use deterministic settings to provide quality assessment that is reproducible.

3.4 Layer 3: Browser-Grounded Execution Testing

The technical contribution of the evaluation architecture of AgentForge-Eval is browser-grounded execution testing. Whereas Layers 1 and 2 determine structural and semantic integrity of generated artifacts, Layer 3 is a test that actually determines what those artifacts themselves do when they are deployed and executed in a real browser environment [9][11]. This is essential to multi-agent systems producing executable results, in which syntactic correctness and semantic coherence are both necessary and not sufficient for functional quality.

The infrastructure of the test running has four consecutive steps. During the environment setup stage, an embedded HTTP server is started to serve the generated artifact and give a fully qualified URL to navigate by browsing as opposed to file-system access patterns, which might not accurately represent production deployment scenarios. The scenario generation stage involves a language model that examines the specification of the artifact and produces a complete code of test scenarios that detail happy path interaction flows, edge case inputs such as empty form submissions and boundary value conditions, error handling paths that ensure graceful failure behavior, and structural navigation completeness that ensures that all specified sections and interactions are reachable [9].

During the test implementation step, the language model is used to translate every generated scenario to executable Playwright tests in the form of browser automation primitives such as click interactions, form field population, element visibility assertion, and network request monitoring. The scripts generated will use known patterns of browser automation, which mimic realistic user interaction sequences and not artificial sequences that will pass with trivially high probability. During the execution and monitoring stage, tests are executed against an artifact hosted with adjustable timeouts. The test runner records pass/fail results of every test case, JavaScript console error output generated in the running test, and network request failures such as missing resources and API errors, and also captures a screenshot of the failure points, which can be viewed and debugged visually [8].

Error enrichment alters the raw test failure output into useful diagnostic data formatted into downstream root cause analysis. The names of failed method names are pulled out and matched with the associated artifact sections, the error message is analyzed to get runtime error types and identifiers of elements that are affected, and stack traces are remapped to particular line numbers in the generated code.

The visual evaluation is an extension of the functional testing capability, which is able to evaluate the visual quality dimensions. Whole-page screenshots are recorded at the most crucial points of the interaction, such as the first page load, the post-navigation states after menu or tab interaction, the post-submission states after filling in the form, etc. These screenshots are assessed using a multimodal language model that measures the layout consistency, responsiveness with different viewport configurations, visual completeness compared to the specification, and the accuracy of the interaction state in ensuring the visual feedback was correct in relation to the application state.

3.5 Cloud-Native Deployment Architecture

To implement the AgentForge-Eval framework on a production scale, it is necessary to have an infrastructure architecture that has the capacity to handle unpredictable evaluation load without affecting the response latency of the larger pipeline [5][6]. The evaluation stack is packaged as a containerized service composed of stateless evaluation workers, each capable of executing the full five-layer evaluation sequence independently.

Kubernetes-based container orchestration takes care of scheduling and provisioning of workers into the evaluation cluster. Horizontal pod autoscaling measures the evaluation queue depth and average worker utilization and sends a scale-out event when the queue depth is above the configured thresholds and a scale-in event when the demand is low to reduce the cost of infrastructure [5]. The worker scheduling policies are designed to give preference to evaluation jobs according to the priority of pipeline stages, with high-urgency evaluations like the ones that cause the delivery of user-facing responses taking precedence over the tasks associated with background quality monitoring. Multi-region deployment configurations spread evaluation capacity over geographic regions and reduce latency on pipelines serving geographically dispersed user populations, as well as providing resilience against failure of regional infrastructure [6].

Container image versioning guarantees reproducible evaluation environments in all deployment environments. Every branch of the evaluation stack is shipped with fixed dependency versions of the browser automation runtime, the assertion framework, and the language model client libraries, removing environment drift between development, staging, and production deployments. Provisioning of infrastructure is controlled by code-defined configurations, which allow

recreation of clusters and rollback to previous versions of evaluation stacks when it is discovered that quality is regressing in production.

3.6 Scalable Data Pipeline Integration

The evaluation telemetry generated by AgentForge-Eval is rich and can be a valuable data asset when added to the larger data engineering context of the multi-agent pipeline [7][13]. The results of the evaluation are organized as typed event records that adhere to a versioned schema defining the outcome of assertions, semantic scores, execution test results, and metadata of failure of each evaluated artifact. These event records are broadcasted to a message queue infrastructure that does not tie the production of evaluation results to downstream consumption, allowing analytics pipelines, monitoring systems, and data storage layers to consume evaluation data at their own processing rates without causing backpressure on the evaluation stack.

An evaluation pipeline ingests evaluation records into a data lake using a data ingestion pipeline that stores the records as partitioned datasets grouped by pipeline stage, evaluation layer, and time window to allow historical quality analysis and longitudinal trend monitoring. The same event stream is consumed by a streaming analytics layer to keep real-time quality dashboards visible that bring to the surface aggregate pass rates, failure distribution patterns, and convergence statistics of pipeline deployments that are running. Evaluation events are messages in history structured as a log, which can be assessed against an existing alerting and incident management process and sent to observability infrastructure [13].

Keeping evaluation schemas compatible with the surrounding data platform requires treating schema definitions as governed assets rather than informal conventions. Evaluation output schemas are registered in a central schema registry, giving downstream consumers a reliable reference point for validating incoming records and identifying schema evolution events before those changes propagate silently into consuming pipelines.

4. Iterative Fix Loop with Convergence Guarantees

Identifying a runtime failure is only half the problem. Reporting it for manual resolution introduces human latency into a pipeline that operates at machine speed, and in high-volume production environments that latency compounds into a substantial quality debt. When browser-grounded execution testing surfaces failures in generated artifacts, AgentForge-Eval moves directly into automated remediation through a structured fix loop rather than deferring correction to a human reviewer [8][10]. The ValidationLoopGenerator manages the complete remediation sequence, moving from artifact generation through execution test runs, failure extraction with enriched error context, targeted repair invocation, and test re-execution, cycling through each stage until every test reaches a passing state or the iteration ceiling is hit.

Bounding that cycle demands more than setting a counter. Three safeguards operate in combination to keep remediation from consuming resources without bound. A hard limit, defaulting to ten attempts and adjustable at deployment time, forces termination regardless of where repair progress stands at that point. A monotonic improvement constraint watches the total failing test count after each attempt and cuts the loop short when two consecutive iterations leave that count unchanged, recognizing that the repair process has settled into a local minimum from which additional attempts will not escape. A postfix validation check runs immediately after every repair to catch runtime errors introduced by the fix itself, closing off the pattern where correcting one failing assertion silently destabilizes behavior that was passing cleanly before [10].

Repair attempts operate at two distinct granularity levels, and the transition between them is governed by convergence progress rather than by a fixed schedule [8]. Fine-grained repair addresses individual failing test methods, supplying the language model with the specific failing assertion, the expected behavior, the actual observed outcome, and the precise code section associated with the failure. Keeping each repair attempt confined to the specific code paths tied to the identified failure reduces the chance of introducing regressions elsewhere, because a narrow change scope leaves unrelated functionality untouched. When fine-grained attempts have burned through more than half the iteration budget without the failure count reaching zero, the loop does not wait for the remaining attempts to play out the same way; it switches automatically to coarse-grained repair mode. At this level, entire modules are regenerated from their specifications rather than patched incrementally, reflecting the judgment that persistent non-convergence signals an architectural mismatch between the planning specification and the implementation strategy rather than a collection of isolated code errors.

Convergence patterns observed across production deployments confirm that the large majority of fixable failures resolve within three iterations, with nearly all remaining cases reaching a passing state by the fifth iteration. The ten-iteration ceiling serves as a practical accommodation for cases involving more complex structural mismatches between specification constraints and implementation patterns. A small residual fraction of cases fails to converge within the full limit, and these consistently share a common characteristic: the root incompatibility lies between a planning decision made several stages earlier and the synthesis strategy applied at generation time, a mismatch that code-level repair cannot resolve. These non-converging cases exit the fix loop with structured failure reports identifying the specific planning decisions that contributed to implementation infeasibility, giving the root cause diagnosis layer the precise upstream attribution it needs to trigger re-planning rather than continued repair.

4.1 Analytics Platform Integration

Fix loop operational metrics carry diagnostic value well beyond their immediate role in tracking remediation progress. The fix loop generates a continuous stream of operational metrics that carry significant diagnostic value beyond their immediate role in remediation tracking [6][7]. When average iterations-to-convergence begins climbing over successive evaluation windows, that trend surfaces as an early signal of prompt quality degradation or model behavior drift, often weeks before the degradation becomes visible through end-user quality feedback or manual review processes.

Predictive models trained on accumulated fix loop telemetry can extend this diagnostic capability into proactive territory, identifying artifact characteristics that correlate with high remediation costs before the evaluation cycle begins [6]. When failure type distributions, per-complexity convergence rates, and repair success rates broken down by failure category are combined, the resulting feature set captures pipeline health at a level of detail that top-level pass rate figures simply cannot provide. Directing artifacts that these predictive models flag as high-risk toward more rigorous initial generation reduces how many enter the fix loop already carrying deep structural problems, moving the cost of correction to an earlier stage where it is considerably cheaper to absorb. Feeding that same telemetry into a real-time streaming analytics layer means anomalous fix loop behavior does not wait for a post-hoc review to be discovered; it becomes visible as it is happening, during the production windows where catching it earliest matters most [7].

5. PromptFoo as Multi-Agent Regression Harness

Of the many ways multi-agent pipelines degrade in production, silent regression is among the hardest to catch because it leaves no immediate trace in pass/fail test outcomes [2]. A prompt adjustment made to sharpen one agent's output can quietly shift its output distribution in ways that invalidate assumptions baked into the prompts of agents further down the pipeline. Provider version updates alter generation behavior at a level of granularity that may appear inconsequential when any single stage is examined in isolation, yet across a ten-stage pipeline those small shifts accumulate into quality changes that users notice before engineers do. Configuration and template modifications carry the same risk, rerouting classification-to-generation pathways in ways that produce measurable quality changes in the final artifact without triggering a single explicit test failure at any intermediate stage [12].

Multi-agent quality regressions do not flip individual test cases from pass to fail; they shift the distribution of quality scores across a test suite. A degradation that drops average structural assertion pass rates by eight percentage points will go undetected by a framework that only reports whether individual test cases crossed their thresholds, because some cases will still pass even as the overall quality envelope contracts. Detecting this class of regression requires a framework that tracks quality distributions over time and compares each new evaluation run against an established statistical baseline rather than against a binary pass criterion.

PromptFoo serves as that framework within AgentForge-Eval, operating through three coordinated mechanisms that together provide continuous regression coverage for the full pipeline [9]. Every incoming evaluation run is checked against the rolling baseline, and when any assertion category drops by more than five percentage points, an investigation workflow is opened immediately. That threshold is intentionally set at a level that catches genuine quality shifts without being sensitive enough to fire on the natural run-to-run variance that language model evaluation produces under stable conditions. When a regression signal is confirmed, a root cause attribution process examines evaluation history and correlates the quality change with specific pipeline modification events, narrowing the responsible change down to a prompt version update, a model configuration adjustment, or an infrastructure modification rather than leaving engineers to reconstruct the causal chain manually [12].

Evaluation reproducibility across deployment contexts depends on consistent provider configuration. Token lifecycle management handles authentication with configurable expiry parameters and automatic retry handling for transient failures, preventing authentication issues from introducing noise into evaluation results. Each evaluation run produces a structured JSON report stored with diff-based comparison metadata, making it straightforward to place any two pipeline versions side by side and quantify the quality difference between them. The regression harness connects directly to continuous deployment pipelines as a blocking quality gate, preventing promotion of pipeline changes that produce regression signals during pre-production evaluation before those changes reach production traffic [9]. Health benchmark ranges define the expected operating envelope for pipeline quality under normal production conditions. Browser-grounded scenario tests hold a threshold above eighty percent. Semantic judge scores carry natural run-to-run variance that stems from language model stochasticity rather than genuine quality shifts, and regression detection thresholds are calibrated to account for that variance so that normal score fluctuation does not generate false regression alerts.

5.1 Data Engineering Workflow Integration

Approaching evaluation as a data engineering practice rather than a testing formality opens the door to applying the same quality governance standards used for primary pipeline data to the evaluation layer itself [7][13]. Evaluation results produced by PromptFoo regression runs are stored as versioned datasets partitioned by pipeline version, evaluation date, and assertion category, creating an archived quality record that makes it possible to reconstruct the complete quality trajectory of any pipeline version across its operational lifetime. That record supports both retrospective debugging, where engineers trace when a specific quality dimension began declining, and forward-looking trend analysis that informs decisions about when prompt optimization or model upgrades are warranted.

Regression reports from the harness are published as structured data assets that downstream quality analytics pipelines consume to aggregate signals across pipeline components, artifact types, and time windows [13]. The aggregated view these pipelines produce feeds engineering dashboards that give platform teams the visibility they need to prioritize optimization efforts based on measured impact rather than intuition. Keeping evaluation output schemas registered in a central schema registry ensures that schema changes to evaluation result formats surface as governed events that downstream consumers can detect and respond to, rather than silent breaking changes that corrupt quality monitoring pipelines without warning. Prompt configurations, model provider specifications, and assertion suite definitions are versioned and managed under the same change governance processes applied to primary data pipeline assets, making evaluation-as-code a concrete operational standard rather than an aspirational one [7].

6. DSPy Programmatic Alignment: From Manual Prompts to Optimized Modules

Manual prompt engineering confronts three structural limitations when applied to multi-agent systems that prevent it from achieving the quality levels available through systematic optimization [4][12]. Hand-tuned prompts reflect the prompt author's perception of quality rather than measured quality as defined by the evaluation stack, creating a persistent gap between subjective prompt design intuitions and objective performance metrics. Prompt engineers working on individual agents lack visibility into downstream cascade effects: a prompt modification that improves classification precision by a marginal amount may introduce output distribution shifts that degrade synthesis quality several stages later.

AgentForge-Eval resolves these limitations by implementing each agent stage as a declarative module with a typed signature that defines the input-output contract for that stage. AgentForge-Eval implements each agent stage as a declarative module with a typed signature that defines the input-output contract for that stage. The classification agent signature maps user descriptions to artifact type, category, and complexity assessments. The planning agent signature maps user descriptions, artifact type classifications, category assignments, and contextual metadata to entity specifications, module hierarchies, and interaction patterns. These typed signatures define the interfaces between pipeline stages, preserving compatibility constraints that allow DSPy to optimize prompt instructions without violating the structural contracts that enable agents to consume each other's outputs [12].

The composite metric function used as the DSPy optimization objective spans all four evaluation dimensions captured by the AgentForge-Eval stack. The structural assertion pass rate and browser execution pass rate each contribute thirty-five percent of the composite score, reflecting the equal importance of structural validity and functional correctness as quality dimensions. The semantic judge score contributes twenty percent, capturing qualitative quality dimensions that the other

metrics do not directly measure. A latency term contributes the remaining ten percent, representing normalized execution time to prevent optimization strategies that achieve quality gains at the cost of unacceptable inference latency.

Two complementary optimization strategies address different aspects of the prompt optimization problem. The Multi-prompt Instruction Proposal Optimizer navigates the instruction space through Bayesian search, generating candidate prompt formulations and scoring each one against the composite metric to find high-performing instructions without having to enumerate every possibility. The bootstrapped few-shot search strategy takes a different angle, drawing demonstration examples directly from pipeline execution traces that scored well, letting empirical performance determine which examples enter the few-shot set rather than leaving that selection to manual curation that tends to reflect the prompt author's assumptions about what good output looks like [4].

Cross-stage optimization represents the most significant advantage of programmatic alignment over manual prompt engineering in multi-agent contexts. Conventional prompt engineering treats each agent as an independent optimization target, ignoring the cascade relationships between agent stages. Programmatic optimization using the full evaluation stack as the objective function enables joint optimization of classification and planning prompts simultaneously, because the metric captures the downstream consequences of classification errors rather than evaluating classification quality in isolation. This end-to-end perspective reveals optimization opportunities that per-agent evaluation cannot identify, producing prompt configurations that achieve better composite quality than any combination of individually optimized per-agent prompts.

Accumulated evaluation data from production pipeline runs is used to train the optimization process, producing candidate prompt sets that reflect the current distribution of production inputs. Candidate prompts undergo A/B evaluation against current production configurations on held-out test sets before any promotion decision is made. A candidate configuration earns promotion only when it clears a three percentage point improvement on the composite metric while holding steady across every individual evaluation layer, a dual requirement that filters out configurations that inflate the overall score by trading off quality on dimensions the composite weight happens to discount. Configurations that clear both criteria are deployed with full version tracking, so if post-deployment monitoring picks up quality degradation that the held-out evaluation did not surface, the pipeline can be rolled back to the prior configuration without delay.

Metric Component	Contribution
Structural Assertion Pass Rate	Thirty-five percent weight reflecting structural validity as a primary quality dimension
Browser Execution Pass Rate	Thirty-five percent weight reflecting functional correctness as equal quality priority
Semantic Judge Average Score	Twenty percent weight capturing qualitative dimensions not covered by deterministic metrics
Normalized Latency Term	Ten percent weight preventing quality gains achieved at the cost of inference performance

Table 3: DSPy Composite Optimization Metric Components [4,12]

7. Multi-Agent Adversarial Robustness and Production Results

7.1 Adversarial Testing Framework

Multi-agent pipelines present attack surfaces that have no equivalent in single-model systems, because vulnerabilities can be introduced at any stage boundary rather than at a single inference point [1][2]. AgentForge-Eval addresses this through a structured adversarial testing module that targets three categories of pipeline vulnerability. Input adversarial probing constructs test inputs designed to stress the classification and planning stages specifically. Descriptions that sit ambiguously across multiple domain categories test whether classification agents produce stable outputs under genuine semantic uncertainty. Contradictory requirements test whether planning agents make principled trade-offs or embed internal inconsistencies into specifications. Injection attempts embedded within legitimate-appearing inputs test whether classification preprocessing neutralizes manipulation before it reaches planning prompts. Boundary-condition specifications, both stripped to minimum content and padded to maximum density, test whether structural constraint policies hold at their edges. Inter-agent adversarial scenarios inject controlled errors at specific pipeline stages to measure how failures propagate. Deliberate misclassification tests whether downstream agents compensate for upstream errors or carry them forward. Edge-case entity counts at policy boundaries test constraint enforcement under adversarial pressure.

Mid-pipeline provider switching tests whether output distribution differences between providers produce detectable quality drops at transition points. Security assertions validate the defensive properties of the pipeline itself, confirming that injection attempts are neutralized before reaching agent prompts, that generated artifacts do not carry executable injection payloads, and that iterative fix loops respect their configured bounds even when inputs are crafted to maximize iteration consumption.

7.2 Production Deployment Results

Production deployment across a high-volume pipeline showed measurable improvement across every primary quality dimension [8][9]. Before integration, the pipeline passed first-pass quality review thirty-four percent of the time, required an average of 4.2 human feedback cycles per artifact, and left twenty-three percent of failures undetected until end-user exposure. After deployment, first-pass acceptance reached eighty-seven percent. Average cycles to approval fell to 1.4. Undetected failures dropped below two percent. DSPy optimization across three cycles produced a twelve percent cumulative composite metric gain, with the planning stage contributing the largest share at eighteen percentage points on structural assertion pass rates. Browser-grounded testing proved its distinct value by catching thirty-one percent of failures that semantic judge evaluation did not flag, confirming that execution-based evaluation captures a failure class that text-based assessment structurally cannot reach.

Quality Dimension	Outcome After Deployment
First-Pass Acceptance Rate	Increased from thirty-four percent to eighty-seven percent across production pipeline runs
Average Iterations to Approval	Reduced from 4.2 human feedback cycles to 1.4 cycles per artifact
Undetected Failure Rate	Dropped from twenty-three percent to below two percent before end-user exposure
Playwright Detection Advantage	Thirty-one percent of failures caught exclusively through browser-grounded execution testing

Table 4: Production Deployment Quality Improvements [8, 9]

7.3 Scalable Evaluation Infrastructure

Sustaining evaluation throughput at production request volumes requires infrastructure that scales with pipeline load rather than becoming a bottleneck [5][6]. The production deployment runs on a cluster that scales worker capacity up and down in direct response to evaluation queue depth, spreading load across availability zones so that neither a regional disruption nor a sudden request spike degrades throughput. Playwright test scenarios are distributed across parallel workers rather than queued behind a single executor, with scheduling decisions weighted by estimated scenario execution time to prevent long-running tests from stalling shorter ones and inflating overall evaluation latency. Shared infrastructure introduces the risk that a high-volume deployment consumes capacity at the expense of others running on the same cluster, and multi-tenant isolation addresses this directly by namespacing evaluation workloads per deployment and enforcing resource quotas that prevent any single deployment from drawing more than its allocated share. Infrastructure configurations are version-controlled alongside pipeline configurations, enabling coordinated rollback of both layers when a quality regression requires reverting to a prior state [6].

7.4 Multi-Agent Failure Taxonomy

Production analysis across diverse domain categories produced a taxonomy of five failure modes, each with a defined detection mechanism and mitigation strategy [1][2]. State corruption, arising from concurrent agent mutations to shared state, is detected through version conflict signals at patch application and mitigated by serializing state commits through optimistic locking. Cascading misclassification occurs when an error introduced at an early stage passes unchallenged through subsequent transitions, growing in impact with each step. Late-stage evaluation discrepancy exposes this pattern, and structural policy validation applied at every stage boundary prevents the error from advancing further once detected.

Semantic drift presents a different problem, where repeated refinement cycles pull the evolving artifact away from what the original specification asked for. Coherence score decay across iterations signals when this divergence crosses an acceptable threshold, and keeping an immutable copy of the original specification in scope for every refinement cycle gives the process a fixed reference point to correct against. Model routing artifacts, producing quality drops at provider transition boundaries, are detected through per-provider score comparison and addressed through provider-aware prompt adaptation at transition points. Fix loop oscillation, where repair cycles alternate between failure states without converging, is detected through the monotonic improvement constraint and resolved by escalating to coarse-grained module regeneration.

8. Discussion and Future Directions

The thirty-one percent gap between failures detected by browser-grounded execution testing and those caught by semantic judge evaluation is the most consequential empirical finding this framework produces. Any evaluation methodology for multi-agent systems that produce executable outputs and rely solely on text-based quality assessment will systematically miss this failure class, regardless of how sophisticated the rubric design becomes. The implication for the field is direct: execution-based grounding is not an optional enhancement to semantic evaluation but a necessary complement to it.

The twelve percent composite metric improvement achieved through programmatic prompt optimization across three cycles represents quality headroom that existed in a pipeline already running in production with manually tuned prompts [4][12]. That gap between manually optimized and systematically optimized quality is not unusual; it reflects the combinatorial complexity of cross-stage prompt interactions that manual engineering cannot navigate without visibility into downstream cascade effects.

Formal safety verification would use adversarial findings to generate structural constraints for the agent interaction graph, proving that specific failure modes are architecturally impossible rather than merely unlikely. Automated adversarial generation would apply the same programmatic optimization infrastructure to produce test cases that maximize failure probability, creating a self-challenging evaluation capability. Standardized cross-pipeline benchmarking would make the framework available as an open evaluation standard for multi-agent systems [11]. Real-time metric streaming would connect evaluation telemetry to observability infrastructure for automated rollback on quality degradation. Federated evaluation sharing would allow organizations to exchange assertion pass rate distributions and failure taxonomy statistics without exposing proprietary prompts or generated outputs [5][7].

Conclusion

Production multi-agent systems fail in ways that no single evaluation strategy can fully anticipate, and the gap between per-agent quality scores and pipeline-level artifact quality has long gone without a systematic solution. AgentForge-Eval closes that gap through a layered architecture that moves from deterministic structural checking through semantic rubric assessment to browser-grounded execution testing, with each layer catching a distinct class of failure that the preceding layers cannot reach. The production results establish that execution-based grounding is not an enhancement to text-based evaluation but a necessary component of it. A material share of the failures that matter most to end users are invisible to semantic assessment regardless of rubric sophistication and only become detectable when generated artifacts are executed in realistic deployment conditions. Programmatic prompt alignment compounds these gains by optimizing the full pipeline against measured quality rather than against the intuitions of individual prompt authors, unlocking quality improvements that manual engineering cannot access without visibility into cross-stage cascade effects. The failure taxonomy developed through production deployment provides a reusable foundation for evaluation framework design across multi-agent architectures beyond the specific system described here.

References

[1] Xueqiang Zhang et al., "A Survey of Multi-AI Agent Collaboration: Theories, Technologies and Applications," in Proc. 2nd Guangdong-Hong Kong-Macao Greater Bay Area Int. Conf. Digital Economy and Artificial Intelligence (DEAI '25), pp. 1875–1881, Jul. 2025. https://dl.acm.org/doi/full/10.1145/3745238.3745531

[2] Nesreen Otoum and Nuha Elkhalili, "Methods and Techniques of Agentic Software Engineering: A Systematic Literature Review," IEEE Access, vol. 14, Jan. 2026. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=11343819

[3] Nicholas Christakis and Dimitris Drikakis, "Evaluating Large Language Models in Code Generation: INFINITE Methodology for Defining the Inference Index," Applied Sciences, vol. 15, no. 7, p. 3784, Mar. 2025. https://www.mdpi.com/2076-3417/15/7/3784

[4] Wenli Yang, Lilian Some, Michael Bain, and Byeong Kang, "A Comprehensive Survey on Integrating Large Language Models with Knowledge-Based Methods," Knowledge-Based Systems, vol. 318, p. 113503, Jun. 2025. https://www.sciencedirect.com/science/article/pii/S0950705125005490

[5] Radoslav Furnadzhiev, Mitko Shopov, and Nikolay Kakanakov, "Efficient Orchestration of Distributed Workloads in Multi-Region Kubernetes Cluster," Computers, vol. 14, no. 4, p. 114, Mar. 2025. https://www.mdpi.com/2073-431X/14/4/114

[6] Vedran Dakić, Goran Đambić, Jurica Slovinac, and Jasmin Redžepagić, "Optimizing Kubernetes Scheduling for Web Applications Using Machine Learning," Electronics, vol. 14, no. 5, p. 863, Feb. 2025.

https://www.mdpi.com/2079-9292/14/5/863

[7] Antonio M. Burgueño-Romero, Cristóbal Barba-González, and José F. Aldana-Montes, "Big Data-Driven MLOps Workflow for Annual High-Resolution Land Cover Classification Models," Future Generation Computer Systems, vol. 163, p. 107499, Feb. 2025. https://www.sciencedirect.com/science/article/pii/S0167739X24004631

[8] Sichong Hao, Xianjun Shi, and Hongwei Liu, "Exploring the Potential of Pre-Trained Language Models of Code for Automated Program Repair," Electronics, vol. 13, no. 7, p. 1200, Mar. 2024. https://www.mdpi.com/2079-9292/13/7/1200

[9] Shaheer Rehan, Baidaa Al-Bander, and Amro Al-Said Ahmad, "Harnessing Large Language Models for Automated Software Testing: A Leap Towards Scalable Test Case Generation," Electronics, vol. 14, no. 7, p. 1463, Apr. 2025. https://www.mdpi.com/2079-9292/14/7/1463

[10] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang, "Automated Program Repair in the Era of Large Pre-Trained Language Models," in Proc. 2023, IEEE/ACM 45th Int. Conf. Software Engineering (ICSE), Jul. 2023. https://ieeexplore.ieee.org/document/10172803

[11] Carlos E. Jimenez et al., "SWE-Bench: Can Language Models Resolve Real-World GitHub Issues?" in Proc. Int. Conf. Learning Representations (ICLR), 2024.

https://openreview.net/forum?id=VTF8yNQM66

[12] Antonio Sabbatella et al., "Prompt Optimization in Large Language Models," Mathematics, vol. 12, no. 6, p. 929, Mar. 2024. https://www.mdpi.com/2227-7390/12/6/929

[13] Eike Permin, Carsten Wohlgemuth, and Tom Keller, "Use-Case-Driven Architectures for Data Platforms in Manufacturing," Platforms, vol. 3, no. 3, p. 15, Aug. 2025. https://www.mdpi.com/2813-4176/3/3/15