Interaction Pattern-Based Fault Localization in Multi-Agent Systems

Correlating Agent Execution Sequences with System Failures

Bogdan-Stelian Duminică[1]

Supervisors: Dr. Burcu Kulahcioglu Ozkan, Dr. Annibale Panichella, Zahra Seyedghorban

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Bogdan-Stelian Duminică
Final project course: CSE3000 Research Project
Thesis committee: Dr. Burcu Kulahcioglu Ozkan, Dr. Annibale Panichella, Zahra Seyedghorban, Dr. Matthijs T.J. Spaan

Abstract

Debugging Large Language Model-based Multi-Agent Systems (LLM-MAS) is challenging because failures emerge from semantic, non-deterministic conversational breakdowns rather than syntactic errors, turning verbose execution logs into a major debugging bottleneck. Traditional Spectrum-Based Fault Localization (SBFL) cannot isolate these flaws since it tracks code-level execution rather than agent actions and interactions. This research utilizes interaction pattern-based SBFL, enhanced by Markov Chain Surprise and statistical validation, to narrow the developer search space by correlating short n-grams with execution failures. Evaluated on the MAST dataset across MetaGPT, ChatDev, HyperAgent, and AG2, the pipeline abstracts raw logs into uniform sequences by using a multi-framework tokenizer. Across the four evaluated frameworks, at n=4, SBFL and Markov surprise selected the same top-three candidate patterns, although their internal rank order sometimes differed. System effectiveness is evaluated through single- and cross-task evaluations, qualitative mapping to MAST failure modes, and semantic verification via an LLM-as-a-judge baseline. Additionally, this validation shows that top-ranked windows capture initiating failure interactions rather than downstream effects: on MetaGPT traces, rank-1 windows receive a *caused* verdict in 68.4% of triggered failing runs at n=4 (and 70.4% at n=3), vs. $\sim$15.3% for random windows, as well as 63.4% vs. 20.9% on AG2 (n=4).

1 Introduction

Large Language Model-based Multi-Agent Systems (LLM-MAS) are transforming software engineering by autonomously handling complex tasks, from planning to coding and reviewing. By leveraging natural language to coordinate specialized agents, these systems can operate without direct human intervention [9]. However, as LLM-MAS are deployed in increasingly complex scenarios, debugging them has become an important bottleneck: verbose multi-turn execution logs force developers to inspect long conversational histories before locating where the execution first breaks down. Unlike traditional software crashes that point to specific lines of code [19], LLM-MAS failures are subtle, semantic, and non-deterministic [4]. Rather than stemming from isolated programming defects, these failures typically manifest as consequences of inter-agent misalignments, semantic ambiguities during task hand-offs, or inadequate verification loops. When an incorrect output occurs, identifying the specific patterns most strongly associated with the failure remains a significant challenge.

To address these identification limitations, multi-agent trace analysis may be reframed by treating logs as sequential execution paths [18] and isolating recurring interaction subsequences that present a significant statistical association with failures. Traditional Spectrum-Based Fault Localization (SBFL) provides a well-established frequency-based foundation for this approach, ranking suspicious entities by their statistical skew towards failing runs compared to successful baselines [1; 19]. In software testing, these entities correspond to static program statements or basic blocks observed under deterministic execution paths. In an agentic environment, this localization mechanism must be translated to target dynamic, short interaction patterns captured over a unified action vocabulary, prioritizing windows that contain initiating failure interactions rather than eventual downstream effects.

Since LLM-MAS failures are difficult to localize from verbose traces alone, this work asks the following primary research question: *To what extent can interaction pattern-based SBFL and Markov Analysis identify and prioritize failure-associated interaction patterns in LLM-MAS by correlating them with execution failures?* To tackle this, four research questions are addressed:

- **RQ1:** How stable and failure-prone are interaction windows ranked by SBFL and Markov surprise across multiple tasks within the same benchmark suite?
- **RQ2:** Which window length balances interpretability, stability, and localization usefulness?
- **RQ3:** Can SBFL identify failures in repeated executions of the same task?
- **RQ4:** Do top-ranked windows point to the initiating failure of the run, not just subsequent symptoms?

This study offers three primary contributions to bridge the gap between traditional software testing and LLM agent actions and communication. First, it includes a multi-framework tokenizer that deterministically maps raw, verbose logs into a uniform vocabulary of abstract agent actions. Second, it introduces a hybrid fault localization engine combining coverage-focused SBFL formulas with Markov Chain Surprise analysis to isolate both frequent failure-linked windows and anomalous conversational transitions, as well as using statistical validation methods. Third, it includes an LLM-as-a-judge evaluation that tests how well do top-ranked windows contain initiating faults, providing semantic validation beyond statistical association.

The remainder of this paper is organized as follows. Section 2 reviews LLM-MAS, the MAST failure taxonomy, cross-task and same-task trace aggregation, and prior work on fault localization and multi-agent verification. Section 3 formalizes interaction-pattern failure identification and describes the tokenization pipeline together with the hybrid diagnostic mechanism (SBFL and Markov transition surprise). Section 4 establishes the experimental baseline, describing the evaluation data from the MAST suite and the multi-framework evaluation setup. Section 5 reports findings for RQ1–RQ4. Section 6 interprets ranked patterns as coordination mechanisms, discusses the practical value, states the limitations, and illustrates one concrete localization example. Section 7 presents the conclusions and outlines directions for future research, while Section 8 reflects on data ethics and reproducibility.

2 Background and Related Work

2.1 Frameworks and the MAST Taxonomy

LLM-MAS coordinate autonomous, role-driven agents through natural language dialogue and structured tool use [9]. However, such platforms implement these execution environments differently, changing their underlying interaction patterns:

- **MetaGPT** [10]: Enforces a rigid, document-driven collaboration workflow controlled by static Standard Operating Procedures (SOPs). Interactions are strictly bound to role-based state changes and professional artifact generation loops.
- **ChatDev** [14]: Utilizes a sequential, game-like development pipeline where pairs of agents communicate asynchronously within isolated, phase-gated chat rooms to iteratively design and refine software components.
- **HyperAgent** [13]: Coordinates operations via dynamic task-graph decompositions, allowing flexible, asynchronous planner-to-editor tool routing and alternating multi-tool execution chains.
- **AG2** [20]: Makes use of an event-driven architecture focused on conversation, where customizable agents interact as peers rather than following fixed, linear phases.

To evaluate these multi-agent vulnerabilities systematically, the Multi-Agent System Failure Taxonomy (MAST) [4] establishes a standardized suite consisting of 14 distinct semantic failure classes organized into three foundational families: **Specification issues** (e.g., disobeying task constraints or entering step repetition loops), **Inter-agent misalignment** (e.g., reasoning-action mismatches or communication boundary breakdowns), and **Task verification** vulnerabilities (e.g., weak or incomplete validation loops by critic components). A complete reference mapping of all 14 failure modes, including formal operational definitions, is provided in Appendix A.

2.2 Cross-Task and Same-Task Aggregation

Classical SBFL operates on execution traces generated by running a static program across distinct test inputs to isolate explicit source-code defects. To adapt it to multi-agent systems, the analysis must expand beyond static code bugs to capture behavioral failures. Consequently, the analysis is split into two configurations depending on whether the goal is to find framework-wide interaction flaws or instance-level bugs.

The cross-task configuration shifts the focus by holding the multi-agent framework architecture constant while varying task requirements across a shared benchmark suite. This setup abstracts away task-specific anomalies to isolate systemic, framework-wide vulnerabilities that persist across different applications. Rather than pointing to a localized code bug, this configuration identifies high-level interaction patterns that are inherently prone to failure, offering a diagnostic capability for architectural flaws.

Conversely, the same-task configuration provides a controlled baseline that closely mirrors traditional software testing by analyzing repeated executions of identical task instances. By holding both task requirements and environmental conditions constant, this configuration tests classical SBFL core assumptions within a non-deterministic environment. It isolates the system to evaluate whether faults can be localized similarly to traditional code defects, establishing a way to evaluate the baseline performance and limits of classical SBFL when applied to LLM-MAS.

2.3 Fault localization and Verification of Multi-Agents

Traditional fault localization for sequential architectures primarily focuses on microservice causal discovery graphs [12] and agent mutation testing [11; 15]. While effective for infrastructure issues or explicit logic mutations, these techniques assume a deterministic control flow. Consequently, they cannot attribute faults that emerge from the non-deterministic, inter-agent behaviors inherent in LLM-MAS. [17; 14].

On the other hand, modeling execution logs as sequential text streams is a recognized strategy in software quality assurance. This methodology builds upon the core concept of the Bugram framework [18], which demonstrated that sequential token execution paths can be modeled effectively through n-gram sliding windows to catch structural execution bugs. However, while Bugram targets low-level, deterministic source code syntax to find implementation bugs, this research abstracts macro-level conversational logs to isolate non-deterministic, semantic interaction flaws between autonomous agents.

While recent multi-agent literature has attempted to monitor active conversational boundaries using online assertions and runtime checks [16; 6], these monitoring techniques typically rely on handcrafted rules or isolated semantic checks. For this reason, translating the n-gram abstraction of Bugram from low-level token paths directly to multi-turn conversation traces allows this framework to isolate architectural vulnerabilities across an entire multi-agent system without relying on predefined rules.

Existing literature applies SBFL to multi-agent failure attribution. FAMAS [8] replays repeated executions, abstracts trajectories into activation patterns over agents and their actions, and ranks candidate failure sources on the Who-and-When dataset. That benchmark uses different handcrafted web-agent tasks (e.g., multi-step retrieval and open-ended research queries), rather than the software-engineering tasks represented in MAST: ProgramDev program synthesis, SWE-Bench-Lite issue repair, and math-oriented runs through MetaGPT, ChatDev, HyperAgent, and AG2. MAST further annotates failures along a taxonomy in which defects may appear at the level of a single agent action (e.g., an incorrect edit or verification call) or as coordination faults that develop across multiple actions and turns (e.g., stalled handoffs and repeated critique loops). However, although FAMAS also covers AG2 (and Magentic-One, not analyzed here), a direct comparison is not possible: it labels the specific roles and actions behind a failure, while MAST annotates only the failure type, not where the root cause emerged, so the initiating interaction this work localizes. The two use different labels, not a shared metric.

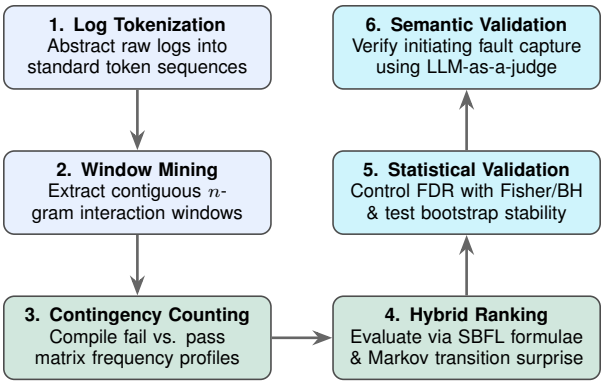

Figure 1: Interaction pattern-based fault localization pipeline, from log abstraction to semantic validation.

Therefore, the proposed approach extends frequency-based fault localization and sequential trace mining to multi-agent logs in a software-engineering setting. Instead of evaluating static code blocks, relying on fixed runtime assertions, or attributing isolated agent actions on general web tasks, the method ranks abstracted interaction patterns that recur across failing runs within each framework.

3 Methodology

Figure 1 outlines the six stages of the fault localization methodology. The pipeline begins by parsing raw, framework-specific conversational logs and mapping them to a standardized, eight-token action vocabulary. Next, this token stream is organized into overlapping, contiguous n-gram windows to capture agent interactions. The framework then uses frequency profiles by tracking how often each unique window appears across successful vs. failed execution traces, prioritizing suspicious behavior by combining frequency-based SBFL metrics with conditional Markov Chain transition surprise analysis. Discovered candidate patterns are then filtered using Fisher's Exact Test and Benjamini-Hochberg adjustments to control false discoveries, along with bootstrap resampling to verify structural ranking stability. Finally, an LLM-as-a-judge model evaluates the top-ranked windows to semantically confirm whether they pinpoint the initiating fault rather than downstream system symptoms.

3.1 Problem Formulation and Formal Notation

Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ be an evaluation dataset consisting of N independent multi-agent execution traces. Each raw trace x_i includes a framework-specific log containing asynchronous agent statements, multi-turn tool invocations, and internal orchestration metadata. The variable $y_i \in \{0, 1\}$ denotes the assigned binary execution label, where successful runs form the negative baseline ($y_i = 0$) and failed runs represent the target positive class ($y_i = 1$).

Each individual trace x_i is projected to a sequence $s_i = (t_{i,1}, \ldots, t_{i,L_i})$ of length L_i through a deterministic mapping function executed over a fixed action alphabet $\mathcal{V}$:

$$f : x_i \mapsto s_i = (t_{i,1}, t_{i,2}, \ldots, t_{i,L_i}), \quad \text{where } t_{i,j} \in \mathcal{V} \quad (1)$$

Given a target sequence window length of order n, the set of all unique contiguous sub-sequences mined from s_i is defined as $G_n(s_i)$. For any candidate n-gram pattern $g \in \mathcal{V}^n$, a binary presence indicator for trace i is defined as:

$$z_{i,g} = \mathbf{1}[g \in G_n(s_i)] \quad (2)$$

The fault localization unit is defined as an interaction pattern g. The system processes these sequence vectors to produce three rank outputs: the primary spectrum-based ranking R_{Ochiai}, the sequential transition ranking R_{Markov} and the combined formula ranking R_{RRF}.

3.2 Log Abstraction and Tokenization

Multi-agent execution logs are unstructured, framework-specific text streams containing role-tagged turns, tool outputs, and orchestration metadata. To enable pattern discovery across multiple frameworks, each raw trace x_i is converted into a standardized sequence over a shared action alphabet, by using a deterministic three-stage pipeline:

1. **Segmentation (Layer 1):** A framework-specific parser splits the raw log into an ordered list of events, isolating speaker identities, raw text, and structural metadata.
2. **Event Typing (Layer 2):** Adapters assign each event a framework-neutral `event_kind` label (e.g., `turn_code`, `terminal`) using regex and milestone tags. This step bridges the structural gap between diverse framework logs and the shared vocabulary.
3. **Semantic Projection (Layer 3):** A shared labeling function maps each event to exactly one of the eight action tokens (Table 1), producing the sequence $s_i = (t_{i,1}, \ldots, t_{i,L_i})$.

In scenarios featuring multi-label ambiguity (an event simultaneously exhibits characteristics of multiple action semantics), Layer 3 resolves conflicts by evaluating a priority hierarchy. Infrastructure noise is isolated and filtered first, followed by explicit runtime errors, validation outcomes, and finally dialogue rules. For instance, an agent message containing an unhandled exception traceback is prioritized as an `ERROR` rather than normal dialogue, ensuring that critical structural failures are not masked by background conversation.

Parser outputs were tested independently via an LLM quality judge on 260 MetaGPT/HyperAgent traces (1–5 rubric, Appendix D.3). Ultimately, 94.6% of judged traces were accurate overall (score $\geq$ 4 out of 5), supporting that mined tokens reflect the actual coordination structure.

3.3 SBFL Mechanism

To determine coordination failures across traces, frequency-based coverage metrics are computed over the abstracted interaction patterns. For each unique candidate n-gram element g, execution frequencies are compiled by cross-referencing pattern indicators ($z_{i,g}$) against binary execution labels (y_i). This updates a standard 2×2 contingency matrix consisting of four absolute metrics:

$$\begin{pmatrix} e_f(g) & e_p(g) \\ n_f(g) & n_p(g) \end{pmatrix} = \sum_{i=1}^{N} \begin{pmatrix} z_{i,g} \cdot y_i & z_{i,g} \cdot (1 - y_i) \\ (1 - z_{i,g}) \cdot y_i & (1 - z_{i,g}) \cdot (1 - y_i) \end{pmatrix}$$

where e and n denote presence and absence, while f and p denote failing and passing runs.

Table 1: Action vocabulary

Token	Operational meaning
REQUEST	Initial task specifications, direct instructions, or subsequent explicit clarification prompts.
INFORM	Default communication: context delivery, reasoning, status updates, or environment outputs.
ACT	Executable work: tool invocation, code generation/execution, or post-verification repairs.
PLAN	Architectural decomposition, role routing, phase handoffs, or coordination transitions.
VERIFY_PASS	Successful validation: passing test suites, environment assertions, or clean milestone reports.
VERIFY_FAIL	Failed validation: test failures, reviewer critiques, validator challenges, or error states.
TERMINATE	Explicit task completion or final answer emission.
ERROR	Hard runtime failures: Python tracebacks, tool crashes, or non-recoverable execution faults.

The primary suspiciousness profile relies on the Ochiai coefficient [2], which has been empirically shown to outperform other similarity metrics in fault localization accuracy, flagging patterns common in failures while filtering out those frequent in passing executions.

$$\text{Ochiai}(g) = \frac{e_f(g)}{\sqrt{(e_f(g) + n_f(g))(e_f(g) + e_p(g))}} \quad (3)$$

Sorting all candidate patterns in descending order of their Ochiai coefficient leads to the primary spectrum ranking, R_{Ochiai}.

To ensure that the prioritized patterns are not artifacts of a specific coverage formula, four additional SBFL metrics are concurrently computed: Tarantula, D^*, Jaccard, and OP2. Reciprocal Rank Fusion (RRF) is applied to consolidate these five independent equations into a single robust ranking [5]:

$$\text{RRF}_{\text{formula}}(g) = \sum_{m \in \{\text{Ochiai, Tarantula, } D^*\text{,Jaccard, OP2}\}} \frac{1}{k + r_m(g)} \quad (4)$$

where $r_m(g)$ represents the absolute ordinal rank assigned to pattern g by formula m. While Ochiai remains the primary standalone metric for the main analysis, RRF serves as a secondary method that treats all five formulas equally. This unweighted approach is used as a consistency check to analyze whether the aggregated multi-formula results align with the primary Ochiai rankings, producing the R_{RRF} ranking. Full formula definitions are present in Appendix D.1.

Before statistical testing, mined patterns are selected using three operational filters to eliminate noise. First, a pattern must appear in a minimum number of failing and total traces to ensure significance. Second, it must not appear so widely that it simply reflects generic framework overhead. Finally, it must demonstrate a divergence between its failure and success rates before entering the hypothesis testing phase. These are fixed, pre-established hyperparameters: chosen before analysis, held constant across all frameworks and n-gram orders, and not tuned on reported outcomes. Thus, this filtering process ensures that this mechanism focuses on wide, repeating flaws in how the agents coordinate rather than isolated, incidental execution errors. All threshold parameters are detailed in the experimental setup section, 4.2.

3.4 Sequential Markov Transition Surprise

Since frequency-based metrics do not take into account the sequential probability of each step, the localization framework is complemented by a conditional sequence model. Multi-agent conversation traces are modeled as an n-gram Markov chain over the action tokens in $\mathcal{V}$. Each step is scored by a conditional probability $P_{\text{pass}}(t_k \mid t_{k-n+1}, \ldots, t_{k-1}) = \frac{\text{count}(t_{k-n+1}, \ldots, t_{k-1}, t_k) + k}{\text{count}(t_{k-n+1}, \ldots, t_{k-1}) + k\,|\mathcal{V}'|}$, estimated from successful executions only ($y_i = 0$). For a trace $s_i = (t_1, \ldots, t_L)$, the pass-only likelihood factorizes as $P_{\text{pass}}(s_i) = \prod_{k=n}^{|s_i|} P_{\text{pass}}(t_k \mid t_{k-n+1}, \ldots, t_{k-1})$, capturing the expected communication routing of a stable task run. While SBFL focuses on windows that are common across failures, Markov surprise ranks interaction patterns whose containing traces present anomalous step transitions under this pass-only baseline.

For each trace s_i, sequence surprise is the average per-step negative log-likelihood of the full token sequence under P_{pass}:

$$S(s_i) = -\frac{1}{|s_i|} \log P_{\text{pass}}(s_i). \quad (5)$$

For an n-gram pattern $g = (t_1, t_2, \ldots, t_n)$, $-\log P_{\text{pass}}(g)$ is not scored in isolation. Instead, the mean trace surprise is compared between failing and passing runs in which g is present. The sequence surprise contrast is:

$$\Delta_{\text{surprise}}(g) = \mathbb{E}_{\text{fail}}[S(s_i) \mid g \in s_i] - \mathbb{E}_{\text{pass}}[S(s_i) \mid g \in s_i] \quad (6)$$

where the expectations are sample averages over failing traces $\mathcal{D}_{\text{fail}}(g)$ and passing traces $\mathcal{D}_{\text{pass}}(g)$ that contain g:

$$\mathbb{E}_c[S \mid g] = \frac{1}{|\mathcal{D}_c(g)|} \sum_{s_i \in \mathcal{D}_c(g)} S(s_i). \quad (7)$$

Ordering patterns by descending Δ_{surprise} yields the Markov ranking R_{Markov}. Aggregating successful runs across the evaluation suite provides a robust model of standard, framework-wide coordination. Thus, deviations from standard coordination appear as increased sequence surprise under the pass-only model.

3.5 Statistical Validation of Failure-Prone Patterns

While SBFL and Markov rankings isolate candidate patterns, ranking alone does not account for the multiple testing problem inherent in datasets. To bridge this gap, candidate sequences are evaluated through a hypothesis-testing pipeline that simultaneously verifies failure association strength, controls for false discoveries, and tracks ranking stability under resampling.

For each candidate pattern g, failure association is assessed with a one-sided Fisher's exact test [7] on the 2×2 contingency table formed from (e_f, e_p, n_f, n_p). This test evaluates whether the pattern appears in failing traces frequently

enough to establish a true statistical correlation, rather than showing up by random chance. The test outputs a nominal p-value, computed as:

$$p = \frac{(e_f + e_p)!\,(n_f + n_p)!\,(e_f + n_f)!\,(e_p + n_p)!}{e_f!\,e_p!\,n_f!\,n_p!\,N!} \tag{8}$$

where N is the total number of independent execution traces ($N = e_f + e_p + n_f + n_p$). Specifically, it compares the failure rate $e_f/(e_f + n_f)$ with the pass rate $e_p/(e_p + n_p)$ under the null hypothesis that g is not associated with failure (g occurs at an equal or lower rate in failing traces than in passing ones). To address n-gram sparsity, testing is restricted to a pre-specified candidate family $\mathcal{G}_{\text{test}}$ defined as:

$$\mathcal{G}_{\text{test}} = \left\{ g \in G_n \;\middle|\; \Delta_{\text{prop}}(g) \geq \Delta_{\min},\ \text{Ochiai}(g) > 0 \right\} \tag{9}$$

where $\Delta_{\text{prop}}(g)$ represents the fail-pass proportion difference:

$$\Delta_{\text{prop}}(g) = \frac{e_f(g)}{e_f(g) + n_f(g)} - \frac{e_p(g)}{e_p(g) + n_p(g)} \tag{10}$$

and $\Delta_{\min}$ is the minimum required effect-size threshold. Candidates within $\mathcal{G}_{\text{test}}$ are ordered by Ochiai score and capped at a maximum testing size of $K_{\text{cap}} = 20$.

Within each testing family of size $m = |\mathcal{G}_{\text{test}}|$, nominal p-values are sorted as $p_{(1)} \leq \cdots \leq p_{(m)}$. Under the Benjamini–Hochberg (BH) method [3], the adjusted q-value for the pattern with rank i is calculated as:

$$q_{(i)} = \min\left(1,\ \min_{j \geq i} \left\{ \frac{m}{j}\, p_{(j)} \right\}\right). \tag{11}$$

Discovered patterns are categorized into three distinct reporting tiers based on this evaluation: `bh_significant` patterns fall within $\mathcal{G}_{\text{test}}$ and satisfy $q \leq \alpha_{\text{FDR}}$, indicating statistically supported failure association; `descriptive` patterns capture high-ranking sequences with substantial Δ or Ochiai scores that fail to meet the adjusted significance threshold ($q > \alpha_{\text{FDR}}$); and `untested` patterns fall outside $\mathcal{G}_{\text{test}}$, meaning they are omitted from the hypothesis testing pipeline.

Finally, to ensure that point-estimate significance is not just an artifact of sampling variation, ranking stability is evaluated through a nonparametric bootstrap over the trace dataset. For each of B resamples drawn with replacement, SBFL scores are recomputed over the baseline top pattern pool to determine how consistently a pattern appears within the top-$k_{\text{stability}}$ set. The bootstrap retention frequency is defined as:

$$\hat{\pi}(g) = \frac{1}{B} \sum_{b=1}^{B} \mathbf{1}\left[g \in \text{Top-}k^{(b)}_{\text{stability}} \right], \tag{12}$$

where Top-$k^{(b)}_{\text{stability}}$ represents the top ranking under resample b, with k=10. This metric decouples statistical significance from structural stability: a pattern may achieve $q \leq \alpha_{\text{FDR}}$ yet present high sensitivity to data composition, or remain highly stable across resamples without qualifying for $\mathcal{G}_{\text{test}}$.

3.6 Semantic Validation of Localized Windows

Statistical ranking identifies failure-associated interaction windows but does not, by itself, establish that a highlighted window contains the initiating fault. To evaluate localization usefulness separately from this designed framework, an LLM-as-a-judge pipeline is applied.

For each failing trace in which a discovered pattern appears, the first occurrence of the pattern window is presented to a judge model as a fixed-length highlighted slice of the tokenized trace. The judge returns a binary verdict: `caused` if the window contains the initiating fault (the earliest concrete mistake that puts the run on the path to failure) and `not_caused` if the window captures only downstream effects or if the origin lies outside the slice.

Aggregated metrics report *caused / triggered* (share of failing runs that contain the respective window, judged `caused`). Random window controls drawn from the same failing traces, without ranking guidance, approximate undirected log inspection and provide a semantic baseline for comparison.

4 Experimental Setup

4.1 Evaluation Data

The cross-task quantitative experiments use execution traces from the MAST Dataset (MAD) [4]. Each record is one complete multi-agent system run: a hierarchical execution log, a binary task outcome (pass vs. fail), and a binary list of failure types (1 if applicable, 0 otherwise). The same-task HyperAgent runs were generated using *deepseek-v4-pro*. Table 2 lists the four used frameworks, broken down by pass/fail counts. All code artifacts, datasets, and experiment scripts are publicly available and can be found on GitHub.

Table 2: MAST evaluation dataset trace aggregates by framework.

Framework	Pass	Fail	Total
AG2	100	497	597
ChatDev	37	93	130
HyperAgent	16	14	30
MetaGPT	58	172	230
Total	**211**	**776**	**987**

4.2 Experimental Process and Parameters

The design varies framework and the n-gram length $n \in \{2, 3, 4, 5\}$. Primary analysis uses n=3 and n=4. n=2 provides a coarse sensitivity check, while n=5 is reported only in the appendix since longer windows capture more trace-specific variations.

Table 3 presents the support, prevalence, testing, and resampling thresholds. A minimum amount of 3 failing traces and 5 total traces ensures that patterns recur across independent failures rather than in isolated runs. On HyperAgent, the smallest subset, these bounds require windows in at least 21% of failures and 17% of all traces. Also, A 75% coverage cap filters out common framework routines that appear in more than three-quarters of all execution traces. Since these highly frequent patterns occur regularly in both passing and failing runs, they represent standard operational overhead rather than windows strongly associated with failures. A required effect-size gap of $\Delta \geq 0.10$ establishes a clear

Table 3: Experimental thresholds.

Setting	Value
Minimum failing-trace support	3
Minimum total-trace support	5
Maximum cross-trace coverage	0.75
Fisher testing	Top 20 by Ochiai, fail-pass gap $\Delta_{\text{prop}} \geq 0.10$
FDR	$q \leq 0.05$
Bootstrap / Bootstrap seed	B=100, seed 42
RRF (formula fusion)	k=60

divergence between failure and success rates. This coverage difference separates fault-linked windows from high-Ochiai boilerplate sequences that show minimal statistical differentiation. Additionally, capping the Fisher family and applying FDR control limit false positives under n-gram sparsity, while bootstrap resampling checks whether top-10 ranks are stable under trace resampling.

Furthermore, the final LLM-as-a-judge evaluation uses `deepseek-v4-pro` at temperature 0. For each failing trace where the rank-1 SBFL pattern appears, the judge evaluates the first occurrence of that pattern as a fixed 4-token window highlighted within the tokenized trace and raw event segment. The verdict is `caused` when the window contains the initiating fault and `not_caused` otherwise. A paired random-window control extracts one fixed-length slice per triggered failing trace from the same run. This approximates undirected trace inspection (opening a random log segment rather than the prioritized window) and serves as an operational lower bound, not a human debugging study. This approach covers MetaGPT cross-task rank-1 windows at $n \in \{2, 3, 4\}$, AG2 cross-task rank-1 windows at $n = 4$ and HyperAgent cross-task and same-task (RQ3) rank-1 windows at $n \in \{2, 3, 4\}$. Reported metrics are *caused / triggered* (share of triggered failing runs where the pattern is judged as `caused`). The full judge prompt is in Appendix C.2.

5 Results

5.1 RQ1: Cross-Task Pattern Discovery

For all four frameworks, R_{Ochiai} and R_{Markov} select the same top-3 pattern set (3/3 exact overlap), though rank order can differ (HyperAgent and AG2). This agreement confirms that these prioritized interaction windows are robust and meaningful, rather than artifacts of a single ranking metric. Instead of indicating architectural redundancy, this alignment reveals that primary multi-agent failure sequences are simultaneously frequent across localized system issues (SBFL) and highly anomalous relative to standard successful transitions (Markov surprise). R_{RRF} overlaps with R_{Ochiai} on the same top-3 sets (3/3) in every framework, proving that overall combining these methods simply reshuffles the same top results instead of highlighting new ones. Bootstrap resampling (B=100) yields near-100% SBFL rank-1 retention across frameworks, while hybrid rank-1 agreement (SBFL vs. Markov) remains high except for HyperAgent at n=4 (52%).

Table 4 lists the top-3 SBFL and Markov patterns at n=4. Under BH control ($q \leq 0.05$), MetaGPT shows statistically supported patterns at all three top ranks. For ChatDev, rank-1 (`ACT:INF:ACT:TERM`) is descriptive, being common in both outcomes with low fail-pass separation, whereas rank-3 (`VP:PLAN:ACT:INF`) meets the significance threshold. For HyperAgent and AG2, all top-3 patterns remain descriptive under the Fisher candidate policy (top 20 patterns with fail-pass $\Delta_{\text{prop}} \geq 0.10$). Ultimately, these statistical distributions imply that multi-agent systems present stable, framework-specific interaction anomalies across varying tasks. While these quantitative metrics establish pattern stability, the specific mechanisms and semantic interpretations of these windows as behavior and coordination are discussed in Section 6.

Table 4: Top-3 patterns by framework n=4.

Fw.	Rk	Pattern	Och.	Δ_{surp}	Tier
MetaGPT	1	`REQ:PLAN:ACT:VF`	0.74	0.11	`bh_sig.`
	2	`VF:INF:VF:INF`	0.73	0.10	`bh_sig.`
	3	`PLAN:ACT:VF:ACT`	0.70	0.06	`bh_sig.`
ChatDev	1	`ACT:INF:ACT:TERM`	0.70	0.10	`descript.`
	2	`ACT:VP:PLAN:ACT`	0.64	0.05	`descript.`
	3	`VP:PLAN:ACT:INF`	0.62	0.04	`bh_sig.`
HyperAgent	1	`PLAN:ACT:TERM:ACT`	0.62	0.05	`descript.`
	2	`TERM:PLAN:ACT:PLAN`	0.58	0.17	`descript.`
	3	`ACT:PLAN:ACT:PLAN`	0.54	0.14	`descript.`
AG2	1	`REQ:PLAN:ACT:INF`	0.65	0.11	`descript.`
	2	`PLAN:ACT:INF:TERM`	0.56	0.15	`descript.`
	3	`ACT:INF:TERM:INF`	0.50	0.17	`descript.`

Abbrev.: REQ = `REQUEST`; INF = `INFORM`; TERM = `TERMINATE`; VP = `VERIFY_PASS`; VF = `VERIFY_FAIL`. Ranks by Ochiai (SBFL).

5.2 RQ2: N-Gram Granularity, Specificity, and Stability

Window lengths $n \in \{2, 3, 4, 5\}$ were evaluated to balance interpretability and stability. Figure 2 measures structural stability as the Jaccard overlap of top-10 bigram-transition sets between adjacent lengths (set Jaccard defined in Appendix D.2). The n=3↔4 transition achieves the highest mean overlap (≈ 0.78), whereas n=2 yields coarse bigrams and n=5 reshuffles top transition sets for MetaGPT and HyperAgent (Appendix C.1). These results justify using n=4 and n=3 for primary analysis.

5.3 RQ3: Same-Task Localization

Compared to cross-task generality which was addressed previously, RQ3 isolates a single task to hold requirements constant and test SBFL's core assumptions under repeated execution. Analysis is conducted on 186 generated HyperAgent executions with `deepseek-v4-pro` model on SWE-bench Verified instance `psf__requests-1142` (115 pass, 71 fail), where the required fix is localized to `prepare_content_length` in `requests/models.py`. The tokenizer maps these logs into the shared vocabulary (Subsection 3.2). Sequences are mined at $n \in \{3, 4\}$ and ranked via Ochiai.

Table 5 reports the top patterns at $n = 4$. Rank-1 `REQ:PLAN:ACT:VERIFY_FAIL` achieves Ochiai $= 0.79$ with

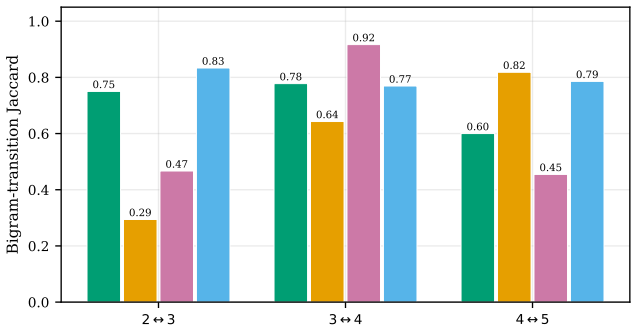

Figure 2: Structural stability across adjacent window lengths. For each framework and pair $(n, n{+}1)$, SBFL top-10 patterns are combined at both lengths and Jaccard overlap of their bigram-transition sets is computed.

MetaGPT ChatDev HyperAgent AG2

Table 5: Same-task SBFL top patterns (n=4, HyperAgent).

Rk	Pattern	Och.	Δ_{prop}	Tier
1	`REQ:PLAN:ACT:VF`	0.79	0.63	`bh_sig.`
2	`ACT:VF:ACT:VF`	0.75	0.57	`bh_sig.`
3	`VF:ACT:VF:ACT`	0.73	0.55	`bh_sig.`

Abbrev.: REQ = `REQUEST`; VF = `VERIFY_FAIL`.

$\Delta_{\text{prop}} = 0.63$, while ranks 2–3 are verify-failure loops (all $q \leq 0.001$). Bootstrap rank-1 retention is 84% at $n = 4$.

SBFL localizes which runs fail the SWE-bench oracle after patch submission rather than the ground-truth line in `prepare_content_length`: both outcomes reach the correct function, but failing runs submit test-rejecting patches.

5.4 RQ4: Semantic Validation via LLM-as-a-Judge

To evaluate the practical debugging utility, RQ4 measures how consistently do top-ranked windows pinpoint initiating faults within multi-agent failure traces. Figure 3 includes MetaGPT cross-task results across $n \in \{2, 3, 4\}$: at n=4, 67 of 98 triggered failing runs (68.4%) receive a `caused` verdict, compared with 15.3% (15/98) for paired random windows. At n=3, *caused / triggered* rises to 70.4% (69/98), vs. 16.3% (16/98) for random controls. Moreover, for AG2, at n=4, RQ4 identifies initiating faults in 63.4% (149/235) of rank-1 windows, compared to just 20.9% (49/235) for random controls. Even though this rank-1 window is not labeled `bh_sig` under the Fisher/BH policy, the performance difference demonstrates that evaluating top-ranked windows is still significantly more effective than random inspection.

As a result, Rank-1 windows are judged to contain initiating faults far more often than random controls (and thus more often than undirected inspection of the same trace would), supporting their use as debugging targets rather than only isolated statistical correlations. However, with only 9 triggered runs out of 14 for HyperAgent cross-task, the rank-1 caused rate (2/9, 95% Wilson CI [6%, 55%]) overlaps the random baseline's interval (0/9, [0%, 30%]). Thus, the HyperAgent cross-task result lacks the statistical power to be conclusive

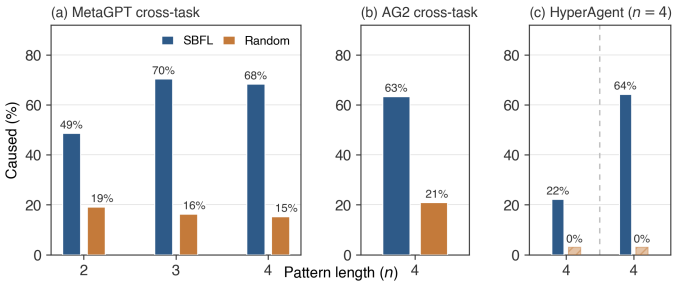

Figure 3: LLM judge *caused* rate for rank-1 windows vs. paired random controls. (a) MetaGPT cross-task at $n \in \{2, 3, 4\}$; (b) AG2 cross-task at n=4; (c) HyperAgent cross-task at $n \in \{2, 3, 4\}$ and same-task RQ3 at n=4.

and is reported for completeness, not as definitive support. On the other hand, on same-task Hyperagent from Subsection 5.3 (`psf__requests-1142`), the results are much better, with rank-1 windows receiving a `caused` verdict in 27/42 triggered failing runs (64.3%), vs. 0/42 for random windows.

6 Discussion

6.1 From Patterns to Failure Mechanisms

On MetaGPT, the rank-1 pattern `REQ:PLAN:ACT:VF` captures the standard operational workflow up to the initial validation rejection. Within the MAST taxonomy, this specific sequence frequently signals a breakdown in Task Verification (FC3), where a reviewer critique or test failure (`VERIFY_FAIL`) occurs but fails to trigger an effective correction loop. For traces mapped to inadequate verification (such as MAST 3.2), this pattern demonstrates that while verification components are used, the system cannot recover from the reported errors. Downstream execution typically stagnates into repetitive critiques, failing to produce code fixes that resolve the core implementation-test misalignment.

Rank-2 (`VF:INF:VF:INF`) describes this subsequent phase. After the first failed check, agents often continue exchanging status and critiques without an effective repair `ACT`. Repeated `VERIFY_FAIL` tokens in this loop are consistent with verification that remains active but never closes the critical gap, rather than with absent verification. The same stagnation can also align with **Reasoning-action mismatch (2.6)** or **Fail to ask for clarification (2.2)** when agents discuss the problem without resolving it. Rank-3 (`PLAN:ACT:VF:ACT`) contrasts with rank-2: after `VERIFY_FAIL`, agents replan and patch instead of only inform, sometimes fixing the run, sometimes failing again.

On ChatDev, rank-3 `VP:PLAN:ACT:INF` is the most failure-specific pattern: a validator emits `VERIFY_PASS`, agents replan based on that, and the trace ends in open-ended `INFORM`, consistent with **Incomplete or incorrect verification (3.2-3.3)**, since correct verifications still result in a failing run eventually.

HyperAgent cross-task rank-1 (`PLAN:ACT:TERM:ACT`) and AG2 rank-1 (`REQ:PLAN:ACT:INF`) describe coordination stagnation with weak verification structure. RQ4 judge rates on cross-task HyperAgent are low (2/9), so these patterns are

better read as framework-level workflow signals than as reliable initiating-fault windows. This interpretation is further limited by HyperAgent's small cross-task fail pool (30 total runs, 14 failing runs, 9 triggered runs by rank-1).

6.2 Practical Value and Illustrative Localization

While statistical ranking identifies coordination windows that co-occur with failure, RQ4 complements this with a semantic check: whether a highlighted window contains the initiating fault (the earliest concrete mistake that commits the run to failure). Furthermore, the overall output consists of a ranked list of suspicious interaction windows prioritized for raw log inspection, which additionally pinpoint potential recurring, failure-associated patterns.

For instance, for MetaGPT (cross-task), consider ProgramDev trace `tid47`, where the team must build a CLI that rejoins split file parts (e.g., `test.part001`, `test.part002`, ...) into the original file. Rank-1 interaction window `REQ:PLAN:ACT:VF` fires at the first validation arc. Here, `REQUEST` is the user requirement, `PLAN` is role coordination, `ACT` is the engineer's first implementation (`join_file_parts`), and `VERIFY_FAIL` is the tester's/reviewer's first rejection (the reviewer notes that the code cannot handle missing middle parts or a missing first part). The judge returns `caused`: this four-turn handoff is the initiating interaction, because the team moves from planning to delivery to verification before the defect is caught, and the first `ACT-VERIFY_FAIL` exchange already exposes flawed join logic. Later `VERIFY_FAIL:INFORM` turns only extend the same critique loop. Thus, inspection can begin at this first implement-and-verify interaction rather than at a later stage of the trace.

6.3 Threats to Validity and Limitations

Some factors may limit how well the results generalize:

- **No root-cause ground truth**: MAST annotates which failure type occurred, but not where it originated, so there are no labels marking the initiating action or interaction that represents the root cause. Since this work localizes that initiating window, all *caused/triggered* validation relies on an LLM-as-a-judge proxy rather than a verified expert-labeled source.
- **Sample size and imbalance:** The evaluation includes 987 total traces, but the distribution across frameworks is unequal. HyperAgent is particularly small (30 traces), and some AG2 splits are extremely skewed toward failing runs. MetaGPT and ChatDev provide larger pools and more stable rankings.
- **LLM judge reliability:** Semantic validation uses a single model (`deepseek-v4-pro`) with a fixed robust prompt and no human in the loop. Thus, verdicts depend on both the LLM and the prompt, and should be treated as scalable semantic checks, not ground truth.
- **Parser accuracy:** The tokenizer relies on deterministic, rule-based matching. While this is very fast and ensures reproducibility, an independent LLM quality judge rated 94.6% of sampled traces as accurate overall (Appendix D.3), though HyperAgent failing traces scored lower (71.4%) than MetaGPT failing ones (98.3%). It cannot capture every nuance of natural-language agent dialogue. Therefore, the focus is on patterns that highlight plausible coordination windows to inspect, not a perfect transcript of raw interaction logs.
- **Rankings purpose:** This approach is designed to flag suspicious coordination patterns for review, not to provide fully automated root-cause diagnosis. Cross-task pooling targets broad framework-level tendencies rather than single-task bugs, and the eight-token vocabulary intentionally trades granular semantic detail for universal analysis. Thus, the results serve as a prioritized debugging starting point.

7 Conclusions and Future Work

Interaction-pattern SBFL with Markov transition analysis successfully prioritizes failure-associated patterns in LLM-MAS logs, within the defined operational boundaries. Under cross-task pooling, SBFL and Markov identify the same top-three pattern set in every framework, with bootstrap-stable rank-1 retention on MetaGPT, a descriptive rank-1 but sharper rank-3 signal on ChatDev, and predominantly `descriptive` top ranks on HyperAgent and AG2. RRF reorders candidates within this shared set rather than introducing new patterns, and window lengths $n{=}3/4$ balance interpretability and rank stability. RQ4 judge validation indicates that rank-1 windows localize initiating faults more often than paired random controls on MetaGPT cross-task traces (68.4% vs. $\sim$15.3% caused / triggered at $n{=}4$, 70.4% at $n{=}3$) and on AG2 at $n{=}4$ (63.4% vs. 20.9%). HyperAgent cross-task judge rates use only nine triggered rank-1 windows from a 30-trace pool, so top windows should be read as indicative rather than conclusive. On AG2, while rank-1 does not meet BH-supported criteria, the semantic validation still judges its first occurrences to contain initiating faults far more often than paired random windows at $n{=}4$, so this still represents a useful debugging start point. Moreover, the code-based tokenization pipeline prioritizes determinism, reproducibility and speed over complete semantic capture.

Future work should test this pipeline on a wider variety of LLM-MAS architectures by collecting more, as well as better balanced, execution traces. Transitioning from offline analysis to live operations would allow the system to match patterns against tokenized traces during the run, catching suspicious coordination windows before a full system failure occurs. These early warnings could trigger automated self-repair, such as an immediate critic rerun, a replanning step, or a rollback to a safe state. Finally, with these capabilities in place, user studies should quantify how much debugging time this pattern-guided approach saves compared to scanning raw logs, and how effectively it helps people isolate root causes.

8 Responsible Research

To ensure full reproducibility, the entire evaluation pipeline (including the multi-framework tokenizer, SBFL execution formulas, Markov chain surprise computations, and statistical validation workflows) is constructed to be entirely deterministic given a fixed input dataset. To enable full external

replicability and verification, all source code, environment configurations, and experimental scripts, along with the extracted execution datasets, have been made publicly available on GitHub. This allows full replicability, so people can directly recreate every table, figure, and ranking metric presented in this study.

For the LLM-as-a-judge semantic validation, model temperature was set to 0, each highlighted window received a single verdict and a seed was fixed for the selection of baseline comparison windows. Additionally, the prompt template was kept constant from the beginning and not iteratively tuned to produce more favorable results. However, to isolate more comprehensively how LLM non-determinism might influence pattern categorization and judge stability, evaluating alternative configurations with judge temperatures greater than zero ($T > 0$) remains an important direction for future work.

Research integrity was maintained by pre-establishing all baseline operational thresholds (such as minimum failing-trace support counts, false discovery rate boundaries - $q \leq 0.05$ - and the eight-token action vocabulary) prior to executing the pattern discovery and evaluation pipeline. This deliberate separation prevented hyperparameter tuning designed to favor specific multi-agent frameworks. Weaker or more descriptive framework outcomes, such as HyperAgent's lower hybrid agreement profile, are reported with the same transparency as MetaGPT's stronger statistical results.

Furthermore, generative AI tools (Gemini 3.5 Flash) were used during the study to improve grammatical structure and clarity, assist in formatting layouts within the Overleaf environment, and support the development of data plotting and final experimental scripts. Finally, regarding data ethics, this study relies exclusively on open-source, publicly released benchmark execution logs and involves no human subjects.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION)*, 2007.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2006.

[3] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society: Series B*, 57(1), 1995.

[4] M. Cemri et al. Why do multi-agent LLM systems fail? In *Proceedings of the 39th Conference on Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025.

[5] Gordon V. Cormack, Charles L. A. Clarke, and Stefan Buettcher. Reciprocal rank fusion outperforms Condorcet and individual rank learning methods. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2009.

[6] Debora C. Engelmann, Angelo Ferrando, Alison R. Panisson, Davide Ancona, Rafael H. Bordini, and Viviana Mascardi. Rv4jaca—towards runtime verification of multi-agent systems and robotic applications. *Robotics*, 12(2), 2023.

[7] Ronald A. Fisher. On the interpretation of χ^2 from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, 85(1), 1922.

[8] Y. Ge et al. Who is introducing the failure? automatically attributing failures of multi-agent systems via spectrum analysis. *arXiv preprint arXiv:2509.13782*, 2025.

[9] Junda He, Christoph Treude, and David Lo. LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 34(5), 2025.

[10] Sirui Hong et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[11] Z. Huang, R. Alexander, and J. Clark. Mutation testing for jason agents. In *Proceedings of the International Workshop on Engineering Multi-Agent Systems (EMAS)*, 2014.

[12] A. Ikram et al. Root cause analysis of failures in microservices through causal discovery. In *Proceedings of the 36th Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[13] Huy Nhat Phan, Tien N. Nguyen, Phong X. Nguyen, and Nghi D. Q. Bui. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*, 2024.

[14] Chen Qian et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

[15] S. Savarimuthu and M. Winikoff. Mutation operators for the GOAL agent language. In *Proceedings of the International Workshop on Engineering Multi-Agent Systems (EMAS)*, 2013.

[16] Thomas J. Sheffler. An approach to checking correctness for agentic systems. *arXiv preprint arXiv:2509.20364*, 2025.

[17] Lei Wang et al. A survey on large language model-based autonomous agents. *Frontiers of Computer Science*, 18(6), 2024.

[18] S. Wang et al. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[19] W. E. Wong et al. A survey on software fault localization. *IEEE Transactions on Software Engineering (TSE)*, 42(8), 2016.

[20] Qingyun Wu et al. Autogen: Enabling next-gen LLM applications via multi-agent conversation. In *Proceedings of the First Conference on Language Modeling (COLM)*, 2024.

A Complete MAST Taxonomy Reference

This appendix lists the 14 failure modes from the Multi-Agent System Failure Taxonomy (MAST) [4]. Table 6 gives each code, official name, and operational definition.

Table 6: Complete mapping of the 14 MAST failure modes (FM-1.1–FM-3.3).

Failure family	Code — Failure mode	Operational definition
FC1: Specification issues (system design)	1.1 — Disobey task specification	Failure to adhere to the specified constraints or requirements of a given task, leading to suboptimal or incorrect outcomes.
	1.2 — Disobey role specification	Failure to adhere to the defined responsibilities and constraints of an assigned role, potentially leading to an agent behaving like another.
	1.3 — Step repetition	Unnecessary reiteration of previously completed steps in a process, potentially causing delays or errors in task completion.
	1.4 — Loss of conversation history	Unexpected context truncation, disregarding recent interaction history and reverting to an antecedent conversational state.
	1.5 — Unaware of termination conditions	Lack of recognition or understanding of the criteria that should trigger termination of the agents' interaction, potentially leading to unnecessary continuation.
FC2: Inter-agent misalignment	2.1 — Conversation reset	Unexpected or unwarranted restarting of a dialogue, potentially losing context and progress made in the interaction.
	2.2 — Fail to ask for clarification	Inability to request additional information from another agent when faced with unclear or incomplete data, potentially resulting in incorrect actions.
	2.3 — Task derailment	Deviation from the intended objective or focus of a given task, potentially resulting in irrelevant or unproductive actions.
	2.4 — Information withholding	Failure to share important data that an agent has already obtained, potentially leading to failures or inefficiencies for other agents.
	2.5 — Ignored other agents' input	Disregarding or failing to adequately consider input or recommendations provided by other agents, potentially leading to suboptimal decisions or missed collaboration.
	2.6 — Reasoning–action mismatch	Discrepancy between the logical reasoning process and the actual actions taken by an agent, potentially resulting in unexpected or undesired behaviors.
FC3: Task verification	3.1 — Premature termination	Ending a dialogue, interaction, or task before all necessary information has been exchanged or objectives met (e.g., missing verification of outputs or failure to communicate key data before shutdown).
	3.2 — No or incomplete verification	Failure to adequately validate or cross-check crucial information or decisions during iterations when verification is expected (verifier present but weak, poorly designed, or incomplete coverage of robustness and edge cases).

Table 6: Complete mapping of the 14 MAST failure modes (FM-1.1–FM-3.3).

Failure family	Code — Failure mode	Operational definition
	3.3 — Incorrect verification	Omission of proper checking or confirmation of task outcomes or system outputs, potentially allowing errors or inconsistencies to propagate undetected.

B Framework Token Mapping Rules

Representative Layer-2 to action-token mapping rules for each framework parser.

Table 7: Framework-specific log signals mapped to action tokens (high-level representative rules).

Framework	Layer-2 signal	Action mapping and distinguishing rules
MetaGPT	`requirement`	`REQUEST` on the first `UserRequirement` / `add_requirement` action or content block.
	`action_invoke`, `turn_code`	`ACT` when actions contain `runcode`, `run_terminal`, `pytest`, `tool_call`, or when Coder/Engineer agents emit code blocks or Python keywords (`def` , `class` , `import`).
	`reviewer_turn`	`VERIFY_FAIL` on critique headers or challenge keywords from Reviewer/Critic/Tester agents on fail-path traces. Tester *reposts* of pytest code after critique remain `ACT`.
	`milestone_ret_{fail,success}`	`VERIFY_FAIL` / `VERIFY_PASS` from pytest output patterns (`short test summary`, `N failed`).
	`terminal`	`TERMINATE` on each terminal event. A conditional log-closure `TERMINATE` may be appended when the raw log contains `Communication Log Ended`, unless a stuck verify loop (≥ 2 `VERIFY_FAIL`) suppresses clean closure.
	`route` (From: header)	`PLAN` for role routing, except Coder $\rightarrow$ Tester handoffs carrying code, which map to `ACT`/`INFORM`.
ChatDev	Warehouse milestones	Milestone keys influence typing: `TestErrorSummary` $\rightarrow$ `VERIFY_FAIL`. `TestReports`/`TestInfo` $\rightarrow$ pass/fail from pytest text. `CmdExecute` $\rightarrow$ `ACT`. `SpeakerNext`/handoff $\rightarrow$ `PLAN`.
	`terminal` (seminar)	`TERMINATE` on final `SeminarConclusion` milestones.
	Testing-phase prose	During `Testing`/`CodeReview` phases, free-text pytest outcomes map to `VERIFY_PASS`/`VERIFY_FAIL` even when embedded in traces otherwise classified as infrastructure.
	Post-completion noise	After `TERMINATE`, trailing `software info`, `cost:`, and OpenAI usage summaries are discarded from the mining window.
AG2	`task_spec`	`REQUEST` from YAML problem blocks or explicitly tagged task-spec segments.
	`turn_code`	`ACT` on code execution turns.
	`terminal`, final answer	`TERMINATE` on `final_answer` segments or `\boxed{...}` / `Final Answer` markers.
	Verifier / Critic turns	`VERIFY_FAIL` when challenge keywords or return-failure patterns appear in verifier-class agents.
	Env agent output	Short numeric or status outputs from `MathProxy`/`CodeExecutor` agents map to `INFORM` (via environment-success typing).
	Phase / speaker change	`PLAN` on markdown section headers or detected speaker transitions.
HyperAgent	`task_spec` (YAML)	`REQUEST` once on the initial SWE-Bench-style instance specification.
	Inner-agent structure	Thought-action headers influence typing: `^Thought:` $\rightarrow$ `PLAN` `^Action:` $\rightarrow$ `ACT`.`^Observation:` $\rightarrow$ pass/fail from observation text.
	Tool / code execution	`ACT` on `._run()`, `pip install`, `subprocess.`, shell/pytest invocations.
	Observation failure	`VERIFY_FAIL` when observations match SWE failure patterns (`tests failed`, `linter error`, `AssertionError`, `exit code: [1-9]`), excluding GitHub-issue description text.
	`terminal`	`TERMINATE` on `Final Answer:` or `\boxed{...}` in the log tail.

Table 7: Framework-specific log signals mapped to action tokens (high-level representative rules).

Framework	Layer-2 signal	Action mapping and distinguishing rules
	Dual layout	Traces may follow an inner ReAct timeline, a ChatDev-style warehouse layout, or a mixed format. The parser selects the appropriate adapter while preserving the same action vocabulary output schema.

C Additional Result Tables and Figures

C.1 RQ2: Granularity and Stability

This part supplements Subsection 5.2 with rank-1 patterns across window lengths and bigram-transition overlap between n=3 and n=4.

Table 8: Rank-1 SBFL patterns by window length n

Framework	n=2	n=3	n=4	n=5
MetaGPT	`ACT:VF` (0.84)	`PLAN:ACT:VF` (0.74)	`REQ:PLAN:ACT:VF` (0.74)	`PLAN:REQ:PLAN:ACT:VF` (0.74)
ChatDev	`ACT:TERM` (0.72)	`INF:ACT:TERM` (0.72)	`ACT:INF:ACT:TERM` (0.70)	`INF:PLAN:ACT:INF:PLAN` (0.80)
HyperAgent	`TERM:ACT` (0.65)	`ACT:TERM:ACT` (0.62)	`PLAN:ACT:TERM:ACT` (0.62)	`PLAN:ACT:PLAN:ACT:TERM` (0.64)
AG2	`INF:PLAN` (0.78)	`REQ:PLAN:ACT` (0.73)	`REQ:PLAN:ACT:INF` (0.65)	`REQ:PLAN:ACT:INF:TERM` (0.62)

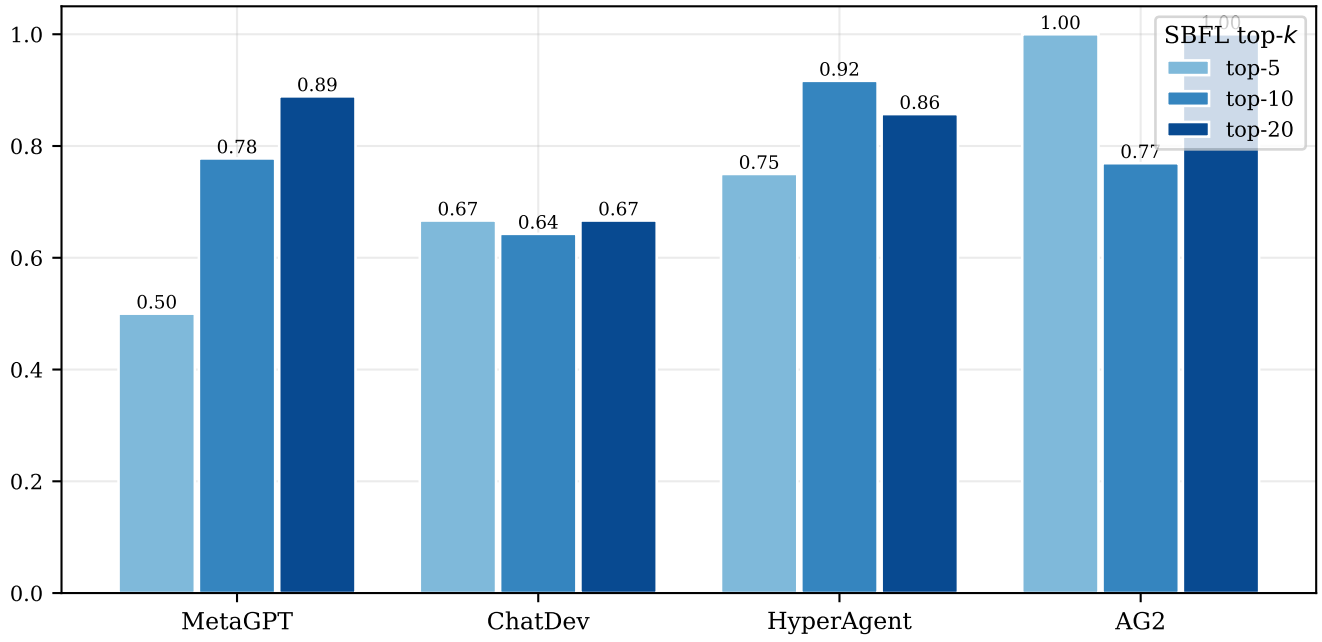

Figure 4: Bigram-transition Jaccard between SBFL top-k unions at n=3 vs n=4.

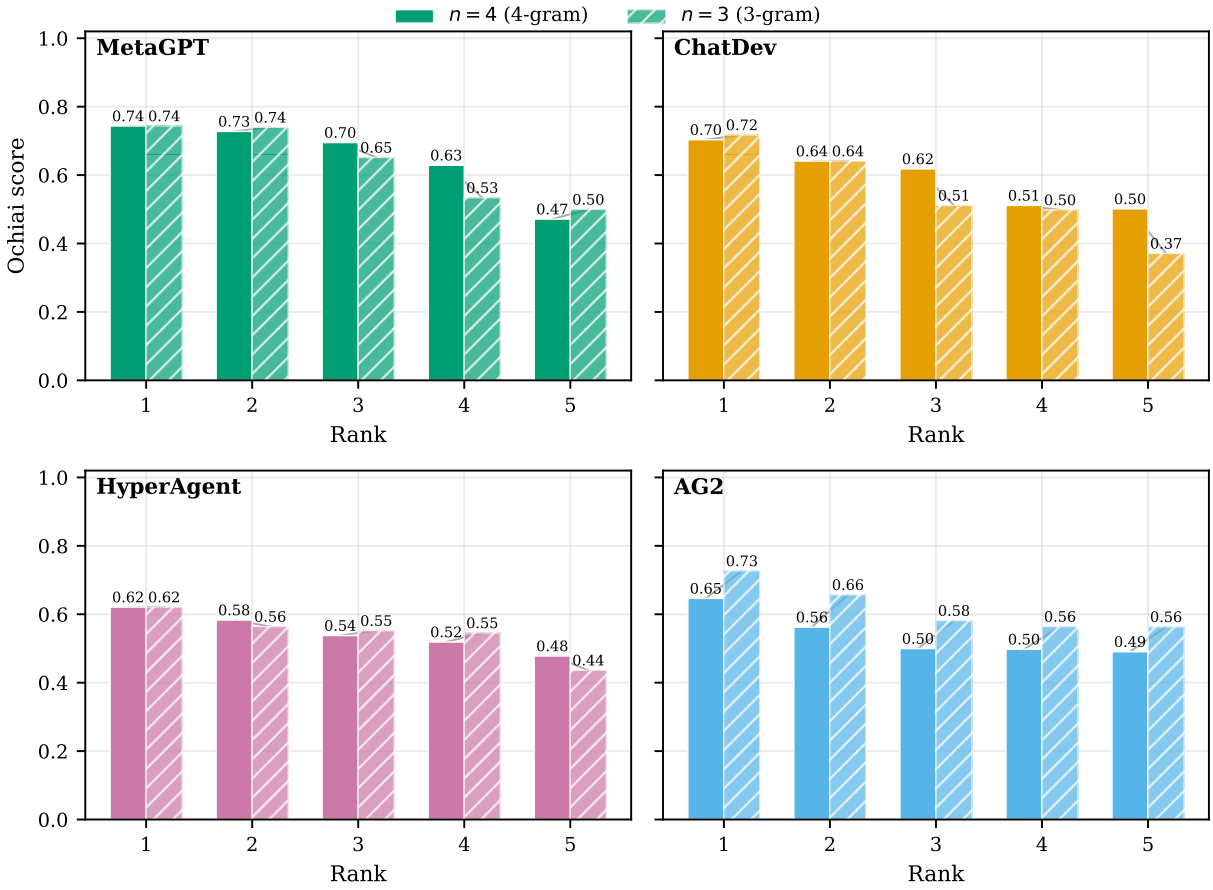

Figure 5: Score stability between n=3 and n=4. SBFL Ochiai for top-5 patterns at each rank.

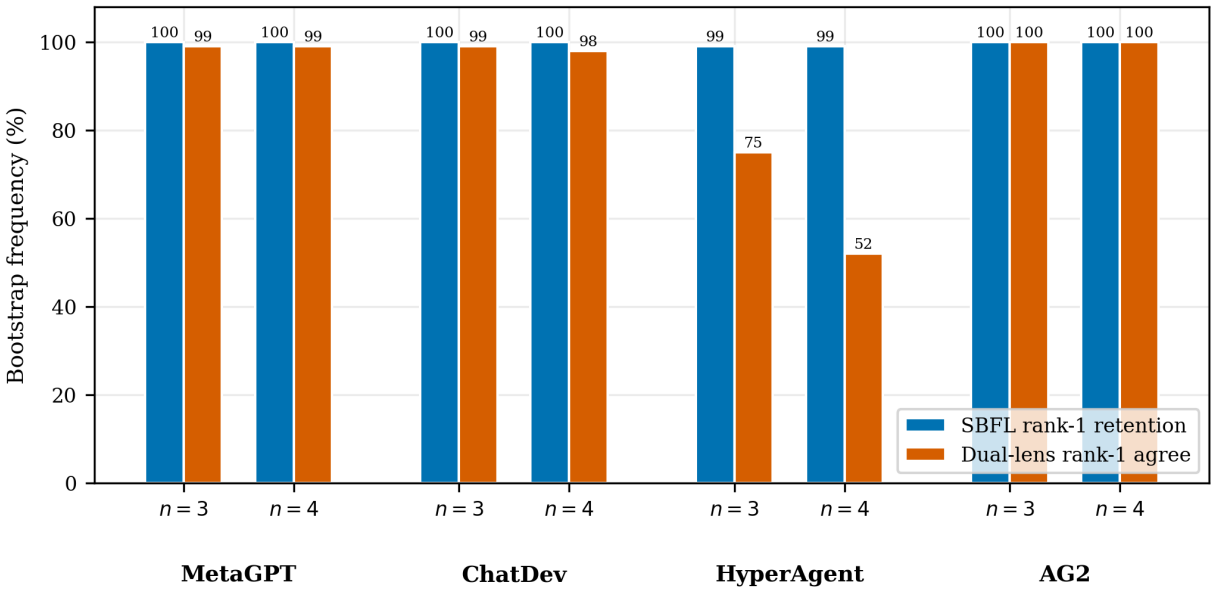

Figure 6: Bootstrap rank-1 retention (SBFL) and hybrid rank-1 agreement (SBFL vs. Markov) at $n \in \{3, 4\}$ (B=100).

C.2 LLM-as-a-Judge Prompt: Failure-Window Localization (RQ4)

The full judge prompt below supports reproducibility of RQ4 semantic validation (Subsection 5.4). Rank-1 and random conditions use the same template, only the window-context line differs.

```
You are an expert analyst of multi-agent LLM systems. You are given a
trace from a run that FAILED.

Token legend (action vocabulary):
- REQUEST: initial task specification
- PLAN: architectural decomposition, role routing, phase handoff, or
  phase transition signaling a new coordination stage
- ACT: code generation, tests, or tool execution
- VERIFY_FAIL: reviewer critique, failed test, or validation rejection
- VERIFY_PASS: successful validation or test pass
- INFORM: status updates or discussion without repair
- ERROR: explicit error signal or exception
- TERMINATE: explicit completion signal

Localization under evaluation:
- Interaction pattern: <pattern tokens or RANDOM>
- Window: a fixed-length contiguous slice of the trace, highlighted
  with <<< >>> below.
- <context line; see note below>

Task:
Decide whether this highlighted interaction caused the failure.

Initiating fault:
The initiating fault is the earliest point in the run where a concrete
mistake or harmful decision puts the execution on the path to failure
(but-for: without it, the run would not have failed in the way it did).
Use the full trace to locate it, then decide whether that fault falls
inside the highlighted window.

Verdict rubric (your answer is final):
- **caused**: the highlighted window contains the initiating fault that
  started this failure.
- **not_caused**: the initiating fault is not in this window (it lies
  earlier or outside the highlighted slice).

Rules:
- Use the full trace for context, but judge only the highlighted
  <<< >>> window.
- If the same token sequence appears elsewhere in the trace, ignore
  those occurrences.
- The window is a fixed-length slice, not a minimal span.

Justification style:
- Describe the highlighted window as a short action-token sequence using
  types from the legend (e.g. PLAN:ACT:VERIFY_FAIL).
- Write at most 2-3 sentences: what interaction the window shows, and
  whether it contains the initiating fault and why.

--- Trajectory excerpt ---
[omitted]

--- Chronological labeled events ---
(Highlighted window marked with <<< >>>.)
[omitted]

Respond with JSON ONLY, keys exactly:
{
  "failure_summary": "one sentence: how this run failed overall",
  "verdict": "<caused|not_caused>",
  "justification": "2-3 sentences in action-type terms: what
```

```
    interaction the window shows, and whether it contains the
    initiating fault and why"
}
```

Context line variants. Rank-1 SBFL: *"This window is the first occurrence of the top-ranked failure-associated pattern in this trace."* Random control: *"This window is a randomly selected fixed-length slice from the same failing trace."*

D Mathematical Definitions

D.1 SBFL Suspiciousness Formulas

For pattern g, let (e_f, e_p, n_f, n_p) denote the contingency counts from Subsection 3.3. Primary ranking uses Ochiai (defined in the main text). Four secondary SBFL heuristics are computed in parallel and combined through reciprocal rank fusion (RRF, k=60) as an internal robustness check:

$$\text{Tarantula}(g) = \frac{e_f/(e_f+n_f)}{e_f/(e_f+n_f) + e_p/(e_p+n_p)} \tag{13}$$

$$D^*(g) = \frac{e_f(g)^2}{e_p(g)+n_f(g)} \tag{14}$$

$$\text{Jaccard}_{\text{SBFL}}(g) = \frac{e_f(g)}{e_f(g)+n_f(g)+e_p(g)} \tag{15}$$

$$\text{OP2}(g) = e_f(g) - \frac{e_p(g)}{e_p(g)+n_p(g)+1} \tag{16}$$

$$\text{RRF}_{\text{formula}}(g) = \sum_{m\in\mathcal{M}} \frac{1}{k + r_m(g)}, \quad \mathcal{M} = \{\text{Ochiai, Tarantula, } D^*, \text{Jaccard, OP2}\} \tag{17}$$

where $r_m(g)$ is the ordinal rank of g under formula m. Reported ranks use R_{Ochiai}. R_{RRF} validates that top patterns are not artifacts of a single formula.

D.2 Bigram-Transition Set Jaccard (RQ2)

Structural stability in Subsection 5.2 uses set overlap on bigram-transition unions, distinct from $\text{Jaccard}_{\text{SBFL}}$ above. For framework f and window lengths n and n', let T_n be the set of bigram transitions induced by the union of SBFL top-10 n-grams at length n:

$$J(T_n, T_{n'}) = \frac{|T_n \cap T_{n'}|}{|T_n \cup T_{n'}|}. \tag{18}$$

Figure 2 reports J for adjacent pairs $(n, n{+}1)$. Table **??** reports J between n=3 and n=4 top-k unions.

D.3 LLM-as-a-Judge Prompt: Tokenization Quality

The deterministic parser was judged independently of fault localization. `deepseek-v4-pro` (temperature 0) scores whether each exported token sequence faithfully represents the raw log on a 1–5 rubric (≥ 4 = acceptable).

```
You are evaluating tokenization quality for the <MAS_NAME> multi-agent framework.
Given the raw trace excerpt and the chronological labeled events table, rate how well
the token sequence captures the trace's interaction flow.

<OUTCOME NOTE: PASSING or FAILING in the dataset>
Outcome label does not affect the score; judge mapping quality only.

Token legend (v1_vocab; shared across frameworks):
- REQUEST: initial task specification
- PLAN: role routing / phase handoff
- ACT: code generation, tests, or tool execution
- VERIFY_FAIL: reviewer critique, failed test, or validation rejection
- VERIFY_PASS: successful validation or test pass
- INFORM: status updates or discussion without repair
- ERROR: explicit error signal or exception
- TERMINATE: explicit completion signal

Token sequence (in order):
  <REQUEST :: PLAN :: ...>
```

```
--- Raw trajectory excerpt ---
<trajectory text>

--- Chronological labeled events (index, token, source text preview) ---
<events table>

Scoring rubric (overall_score, integer 1-5):
- 5: Faithful mapping; token order and semantics match the trace.
- 4: Accurate overall; minor borderline labels (e.g. INFORM vs VERIFY_FAIL).
- 3: Rough structure preserved but several clear mismatches.
- 2: Systematic mislabeling or major gaps.
- 1: Token sequence does not reflect the trace.
Most acceptable tokenizations should score 4 or 5. Reserve 3 and below for clear problems.
Judge semantic fit, not segmentation granularity. Do not penalize correctly dropped
infra/log noise.

Respond with JSON ONLY matching this shape (keys exactly):
{
  "overall_score": "integer 1-5",
  "acceptable": "true if overall_score >= 4 else false",
  "summary": "one sentence: how well the tokenization maps the trace",
  "strengths": ["brief bullet points"],
  "issues": ["brief bullet points; empty list if none"],
  "worst_mismatch_index": "null or integer event index with the clearest mismatch"
}
```

Summary: Model: `deepseek-v4-pro`, temperature 0. 260 traces judged (MetaGPT n=230, HyperAgent n=30). MetaGPT: mean score 4.11, 98.7% acceptable (≥ 4); HyperAgent: mean 3.87, 86.7% acceptable; overall 94.6% acceptable (246/260).