Specification-Based Fault Localization for LLM-Based Multi-Agent Systems

Mete Aksoy[1]

Supervisor(s): Burcu Kulahcioglu Ozkan[1]**, Annibale Panichella**[1]**, Zahra Seyedghorban**[1]

[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 2026

Name of the student: Mete Aksoy
Final project course: CSE3000 Research Project
Thesis committee: Burcu Kulahcioglu Ozkan, Annibale Panichella, Zahra Seyedghorban, Matthijs Spaan

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

We investigate whether specification-based fault localization can identify failure modes and families in failing LLM-based multi-agent systems (LLM-MAS), evaluated on the MAST Multi-Agent Debate (MAD) dataset. We implement a six-stage pipeline that extracts global and dynamic behavioral constraints from execution traces, evaluates them step-by-step, and uses the resulting violation log to drive an LLM judge toward a structured failure diagnosis. On the 18-trace human-annotated MAD-Human dataset, the pipeline achieves 33.3% strict mode and 50.0% strict family accuracy, compared to 5.6% and 22.2% for a no-specification baseline; comparable gains are observed on a 14-trace HyperAgent SWE-Bench-Lite subset. Analysis of constraint violation logs suggests that the *taxonomy targets* carried by constraints, not their syntactic type, may be a primary driver of diagnostic accuracy, and that three constraints per step achieves equivalent accuracy to five at substantially lower cost.

1 Introduction

Large Language Model-based Multi-Agent Systems (LLM-MAS) are rapidly becoming a mainstream paradigm for tackling complex software engineering tasks. In these systems, multiple LLM-powered agents, each assigned a distinct role such as Planner, Coder, Reviewer, or Tester, collaborate through prompts, messages, and shared tool invocations to produce a joint outcome [7]. Frameworks such as ChatDev [8] and Generative Agents [16] demonstrate that this architecture can successfully automate complex collaborative tasks, and an active research community is exploring further applications in code generation, automated testing, and system design [17].

The growing deployment of LLM-MAS also exposes a significant engineering challenge: debugging. When a multi-agent system produces an incorrect or incomplete result, isolating which agent is responsible is far harder than in conventional software. Cemri et al. [9] empirically identify several properties that make LLM-MAS failures qualitatively different from traditional software bugs: failures are often semantic rather than structural; execution is non-deterministic; errors propagate across agents disguising their origin; and an end-to-end pass/fail signal conceals each individual agent's contribution. Together, these properties make manual debugging prohibitively expensive.

Specification-based fault localization (spec-based FL) [13] addresses the agent attribution problem by deriving formal or semi-formal behavioral specifications for each agent from the trace itself, checking whether each step satisfies those specifications, and using the resulting violation signals to attribute the failure to a specific agent and step. Recent systems including AgentRx [10], AgentPex [11], and ErrorProbe [12] have explored this approach, but all were evaluated on narrow benchmarks and none have been tested on the MAST Multi-Agent Debate (MAD) dataset [9], which provides richer human-annotated ground truth across six heterogeneous frameworks and fourteen failure modes spanning three families: specification violations, inter-agent misalignment, and task verification failures.

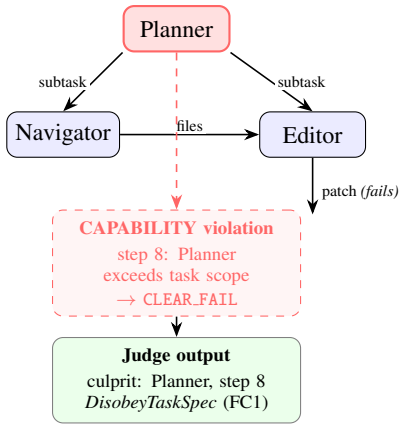

Figure 1: Motivating example: three HyperAgent agents collaborate on a bug-fixing task. The spec-based FL pipeline detects a CAPABILITY violation at step 8 (Planner) and attributes the failure to *DisobeyTaskSpec* (FC1).

Motivating example. Consider a software development team using HyperAgent [2], an LLM-MAS in which a Planner agent decomposes a bug-fixing task and delegates subtasks to a Navigator and an Editor. Their automated test suite reports a failure after the system submits a patch, but the end-to-end output gives no clue as to which agent is at fault: the Planner produced a plan, the Navigator located the relevant files, and the Editor submitted a diff, all apparently without error. Figure 1 illustrates the agent interactions and the constraint violation identified by the pipeline. Running the spec-based FL pipeline on the execution trace proceeds as follows. Stage 1 normalizes the HyperAgent log into a step-indexed representation. Stage 2b generates a CAPABILITY constraint at step 8: *the Planner must not assign a subtask whose scope exceeds the originally reported issue*. Stage 3 evaluates this constraint and finds a CLEAR_FAIL: the Planner extended the task scope to refactor unrelated code. Stage 4 logs this as the earliest-occurring violation, and Stage 5 passes the resulting validation log to the judge, which outputs: culprit agent = Planner, decisive step = 8, predicted failure = *DisobeyTaskSpec* (FC1). Armed with this signal the developer inspects step 8, confirms the unwanted scope expansion, and adds an explicit scope constraint to the Planner's system prompt. The patch succeeds on the next run. This scenario illustrates the end-to-end value of spec-based FL: it converts an opaque multi-agent failure into a specific, actionable diagnosis without requiring the developer to read through the full trace manually.

This paper investigates *whether specification-based fault localization can correctly identify the failure mode and family in failing LLM-based multi-agent systems, as evaluated on the MAST MAD dataset*, and characterises the conditions under which it succeeds or falls short.

We decompose this into three sub-questions. **RQ1**: To what extent can spec-based FL identify failure modes and families in LLM-MAS traces? **RQ2**: How does trace metadata availability (system prompts, tool schemas, role definitions) affect

Table 1: The MAST taxonomy: three families and fourteen failure modes.

Family / Mode	Description
FC1: Specification Issues	
DisobeyTaskSpec	Agent ignores its assigned task
DisobeyRoleSpec	Agent acts outside its assigned role
FC2: Inter-Agent Misalignment	
IgnoredOtherAgentInput	Agent ignores another agent's message
ReasoningActionMismatch	Action contradicts stated reasoning
PrematureTermination	Agent stops before task is complete
UnawareOfStoppingCond	Agent continues past a stop criterion
IncompleteVerification	Only part of the output is verified
IncorrectVerification	Verification reaches wrong conclusion
FC3: Task Verification Failures	
NoVerification	Output is not verified at all
Hallucination	Agent fabricates information
FaultyReasoning	Reasoning is logically incorrect
ContextLoss	Agent loses track of prior context
FailTaskSpec	Output fails the task specification
InsufficientInfo	Agent acts on insufficient information

specification extraction quality and downstream localization accuracy? **RQ3**: Which characteristics of dynamically generated specifications (type, targeted failure family, or evaluation method) determine their diagnostic effectiveness?

The main contributions of this work are: (1) a spec-based FL pipeline adapted to the heterogeneous trace formats of MAST MAD, handling six frameworks including AppWorld, AG2, HyperAgent, ChatDev, MetaGPT, and GAIA; (2) an empirical evaluation on MAD-Human showing that the full pipeline substantially outperforms a no-specification baseline on both failure mode and family identification; (3) a corroborating evaluation on HyperAgent-SWE confirming consistent accuracy gains under the same configuration; and (4) an analysis of dynamic constraint logs identifying taxonomy target coverage as the primary determinant of diagnostic effectiveness, with constraint type playing no discriminating role.

The remainder of this paper is structured as follows. Section 2 provides background on LLM-MAS failure modes and prior spec-based FL work. Section 3 presents our pipeline. Section 4 describes the experimental setup and reports results for all three research questions. Section 5 discusses responsible research considerations. Section 6 places the findings in broader context. Section 7 concludes and outlines directions for future work.

2 Background

2.1 LLM-MAS Failure Modes

Cemri et al. [9] introduce the MAST taxonomy, which organizes LLM-MAS failures into three families and fourteen modes summarized in Table 1. **FC1: Specification Issues** captures agents that violate their assigned role or task constraints. **FC2: Inter-Agent Misalignment** captures breakdowns in coordination between agents, including ignoring each other's outputs, acting contrary to stated reasoning, stopping too early or too late, and verifying incorrectly. **FC3: Task Verification Failures** captures errors in verifying or producing the final output, such as hallucination, faulty reasoning, or missing verification altogether.

In our evaluation (Section 4), the pipeline predicts which of these 14 modes explains each failing trace; predictions are compared against human- or LLM-assigned ground-truth labels at both the mode level (14-class) and family level (3-class). This taxonomy is more fine-grained than those used in prior spec-based FL work: AgentRx defines nine failure categories and ErrorProbe uses a binary pass/fail signal, making MAST the first opportunity to test whether spec-based FL can discriminate at family granularity.

2.2 The MAST MAD Dataset

The Multi-Agent Debate (MAD) dataset [9] consists of 19 execution traces from six frameworks: AppWorld [6], AG2 [3], HyperAgent [2], ChatDev [8], MetaGPT [4], and GAIA/MagenticOne. Each trace was annotated by three human annotators who independently voted yes/no on all fourteen failure modes; we use majority-vote labels as ground truth. Eighteen of the nineteen traces have at least one majority-voted label; the nineteenth is excluded from evaluation. Ground truth is available only at the failure mode and family level; no annotation identifies the culprit agent or decisive step.

In addition to the 18-trace human-annotated split, the full MAD dataset contains 30 HyperAgent [2] SWE-Bench-Lite [1; 18] traces with LLM-annotated (OpenAI GPT) ground truth. Of these, 16 are passing traces (all-zero annotations) and 14 are failing traces that form our second evaluation set. The HyperAgent traces are structurally distinct from the human-annotated MAD traces: they consist of plain-text execution logs with no embedded system prompts, no tool schema definitions, and no agent role specifications, making them a natural probe for RQ2. We use the same model family (GPT-4o) for our pipeline's constraint extraction and judge, keeping the annotation and evaluation models consistent.

2.3 Prior Specification-Based Fault Localization

Fault localization in traditional software has been studied extensively [14; 15]; specification-based approaches adapt these ideas to the LLM-MAS setting by deriving behavioral constraints from agent specifications rather than from test coverage.

AgentRx [10] extracts global constraints (role-based, tool-based, protocol-based, behavioral) from system prompts and tool schemas, evaluates each constraint step-by-step, and passes the resulting violation log to an LLM judge for failure attribution. It is evaluated on 115 manually annotated failed trajectories spanning structured API workflows, incident management, and open-ended web/file tasks, demonstrating improved step localization and failure attribution over baselines.

AgentPex [11] extracts behavioral rules from agent prompts and system instructions and evaluates agentic traces for compliance against those rules. It is evaluated on 424 traces from τ^2-bench in telecom, retail, and airline customer service domains, showing that specification-based compliance checking surfaces violations missed by outcome-only benchmarks.

ErrorProbe [12] introduces a self-improving diagnostic framework that operationalizes a MAS failure taxonomy to detect local anomalies, applies symptom-driven backward tracing to prune irrelevant context, and validates error hypotheses via a specialized multi-agent team. It maintains verified

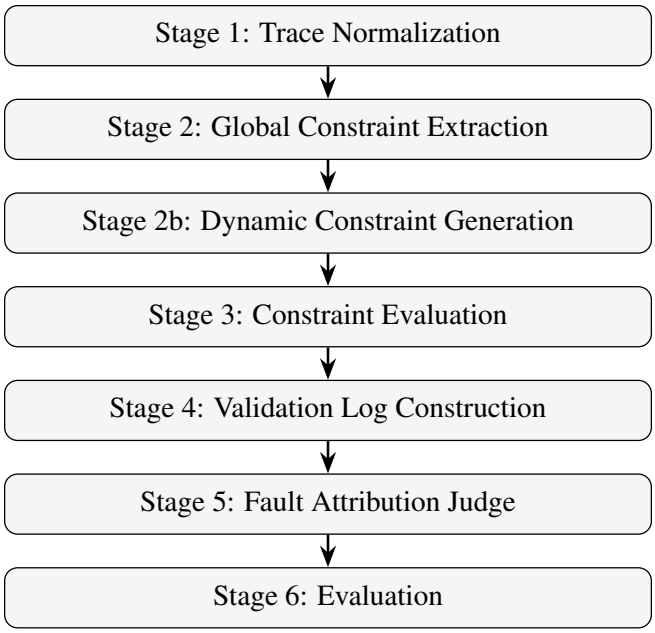

Figure 2: Overview of the six-stage specification-based FL pipeline.

episodic memory to enable cross-domain transfer without retraining, and is evaluated on the TracerTraj and Who&When benchmarks.

Our work differs from all three in what is being measured: we ask whether spec-based FL can identify the correct failure mode and family from a predefined 14-class taxonomy, a classification task none of these works address. AgentRx and AgentPex measure step localization accuracy and specification compliance detection on their respective benchmarks; ErrorProbe measures agent-level and step-level attribution. None of these systems have been evaluated on MAST MAD, and none report accuracy against a multi-family failure taxonomy, leaving the gap this work addresses.

3 Specification-Based FL Pipeline

Our pipeline consists of six stages, following the AgentRx architecture with adaptations for the heterogeneous trace formats of MAST MAD. Figure 2 provides a high-level stage overview; Appendix A illustrates the data flow in detail using the motivating example. The design is model-agnostic; specific LLM choices used in our implementation are described in Section 4.

Stage 1: Trace Normalization
MAST MAD contains six structurally distinct trace formats across frameworks. We implement a format-routing normalizer that converts each format into a common intermediate representation (IR) consisting of a list of steps, where each step records the agent name, message content, and step index. The normalizer handles: plain-text AppWorld traces (boundary-delimited by agent names), AG2 JSON trajectory dictionaries, HyperAgent log-line strings, ChatDev timestamped conversation logs, MetaGPT structured and communication logs, and GAIA/MagenticOne separator-delimited traces. *Example: the HyperAgent log from Section 1 is parsed into a sequence of numbered steps, each attributed to Planner, Navigator, or Editor.*

Stage 2: Global Constraint Extraction
Global constraints $\mathcal{C}^G$ are trace-independent behavioral rules derived from static metadata (system prompts, tool schemas, role definitions) that must hold across the entire trace (hence "global"). When such metadata is present, we extract $\mathcal{C}^G$ using four LLM calls following AgentRx [10], one per category: (i) role constraints (what each agent is and is not permitted to do), (ii) tool constraints (which tools each agent may invoke), (iii) protocol constraints (ordering and communication rules between agents), and (iv) behavioral constraints (task-level properties that must hold end-to-end). When no system prompt or tool schema is present, we fall back to a role-template library for common roles (Planner, Coder, Reviewer, Tester, Orchestrator). Each constraint carries a `constraint_type` that determines how it is evaluated: **CAPABILITY** (agent must not exceed its role scope), **TEMPORAL** (something must hold before or after a given step), **PROTOCOL** (agent must follow a defined interaction ordering), **RELATIONAL_POST** (a relationship must hold between two agents' outputs), or **ANY** (general semantic property). Each constraint also carries `taxonomy_targets` (one or more of the 14 MAST modes it is designed to surface) and a `check_type` (`nl_check` for semantic evaluation via an LLM evaluator, or `python_check` for deterministic code execution). *Example: the HyperAgent trace has no system prompt, so the extractor falls back to role templates and produces, e.g., a CAPABILITY constraint "the Planner must not assign tasks outside its delegated scope" targeting* DisobeyTaskSpec.

Stage 2b: Dynamic Constraint Generation
Unlike global constraints, dynamic constraints $\mathcal{C}_k^D$ are generated fresh at each step k conditioned on the trajectory prefix $T_{\leq k}$, hence "dynamic": they capture context-dependent violations that only become apparent as the trace unfolds and cannot be anticipated from static metadata alone. For each step k, an LLM generates up to five such constraints (or three, as discussed in Section 4.2), which are then evaluated by an LLM evaluator following AgentPex [11]. The generator is provided the complete authoritative list of all 14 valid MAST modes, explicitly grouped by family, with example constraint patterns for each of the three failure families. Crucially, once generated, dynamic constraints are *retained* in the constraint pool: a constraint generated at step k is re-evaluated at every subsequent step $k' > k$. *Example: at step 8 the generator emits a CAPABILITY constraint:* the Planner must not assign a subtask whose scope exceeds the originally reported issue*, which is then re-evaluated at all subsequent steps.*

Stage 3: Constraint Evaluation
Let $\mathcal{C}^D_{\leq k} = \bigcup_{j \leq k} \mathcal{C}_j^D$ denote all dynamic constraints generated up to and including step k. Each constraint $c \in \mathcal{C}^G \cup \mathcal{C}^D_{\leq k}$ is evaluated at step k and assigned one of three verdicts: `CLEAR_PASS`, `CLEAR_FAIL`, or `UNCLEAR`. Deterministic constraints (e.g., ordering rules expressible as code) are checked programmatically; semantic constraints (e.g., whether an agent's action violates a role boundary) are assessed by an

LLM evaluator. Ambiguous cases are treated as a pass to avoid false positives. *Example: the CAPABILITY constraint is evaluated at step 8 and returns* `CLEAR_FAIL`*: the Planner has extended the task scope to refactor unrelated code.*

Stage 4: Validation Log

All constraint verdicts across all steps are aggregated into a step-indexed validation log $\mathcal{V}$, where each entry records the step index, the constraint that fired, its verdict (`CLEAR_FAIL`), and the MAST taxonomy targets it carries. This log serves as structured evidence that the judge reads in Stage 5: rather than processing raw trace text, the judge reasons over a compact record of *what was violated, when, and which failure family it implicates*. Because constraints are re-evaluated at every subsequent step, the same constraint can fire repeatedly; we keep only its first occurrence, as the earliest violation is the most diagnostic signal. *Example: the step-8 CAPABILITY violation is recorded once; later re-evaluations of the same constraint at steps 9–12 are discarded.*

Stage 5: Fault Attribution Judge

The validation log $\mathcal{V}$, the full trajectory, and the MAST taxonomy checklist are passed to an LLM judge [5] that outputs the culprit agent, decisive step, predicted failure mode, and predicted failure family. For traces with at most 50 steps the judge receives the full trajectory at once; for longer traces it processes step-by-step before making a final decision, following AgentRx [10]. *Example: the judge reads the violation log and outputs culprit = Planner, decisive step = 8, failure mode* = DisobeyTaskSpec *(FC1).*

Stage 6: Evaluation

We compare the judge's predictions against majority-vote ground-truth labels and report two accuracy variants, each computed at both failure mode (14-class) and failure family (3-class) granularity. *Strict* accuracy requires the prediction to match the single highest-vote-count label (ties broken by taxonomy order). *Lenient* accuracy credits any majority-voted label, which is the methodologically appropriate primary metric here: the MAD annotation scheme has three annotators independently vote yes/no on all fourteen modes, so multiple modes can share majority support and each is an equally valid ground-truth answer; treating only one as correct would unfairly penalise correct predictions that happen not to be the plurality winner.

4 Experimental Setup and Results

4.1 Setup

We evaluate on two datasets summarized in Table 2. **MAD-Human** consists of 18 human-annotated MAD traces spanning six frameworks; **HyperAgent-SWE** consists of 14 failing HyperAgent [2] SWE-Bench-Lite [1] traces with LLM-annotated ground truth (16 passing traces are excluded).

The extraction and judge model is GPT-4o [19]; the per-constraint evaluation model is GPT-4o-mini. All LLM calls use temperature 0 to minimize non-determinism. We run the full pipeline with 5 dynamic constraints per step (5c) and 3 per step (3c). Each pipeline run is time-intensive, requiring dozens of LLM calls per trace. We therefore run most configurations once, and restrict multi-run evaluation to the primary configuration (Full Pipeline, 3c) on HyperAgent-SWE, which we run 10 times to characterize run-to-run variability (see Section 4.3).

Table 2: Dataset summary. The table lists the number of traces per failure mode (modes with zero occurrences omitted).

	MAD-Human	**HyperAgent-SWE**
Traces	18	14
Annotation	Human	LLM (GPT)
Frameworks	6	1
Ground-truth failure mode distribution		
DisobeyTaskSpec	6	7
UnawareOfStoppingCond	3	5
ReasoningActionMismatch	4	—
NoVerification	3	—
InsufficientInfo	2	1
IncorrectVerification	—	1

4.2 Results on MAD-Human

Table 3 reports results on MAD-Human for all configurations.

Both pipeline configurations substantially outperform the no-specification baseline across all metrics. Comparing the configurations, 5c and 3c: the 3c configuration matches or exceeds 5c on every metric, with observed differences of 5.5 percentage points in strict mode (33.3% vs 27.8%) and 5.6 points in strict family (50.0% vs 44.4%). On a dataset of 18 traces, differences of this magnitude may fall within run-to-run variation due to LLM non-determinism, so we do not treat the gap as conclusive evidence that 3c is inherently more accurate. The practical case for 3c is clear regardless: 5c generates 67% more constraint evaluation calls per step, each trace already requires dozens of LLM invocations, and the added cost yielded no observable benefit. We adopt 3c as the recommended configuration on cost-efficiency grounds.

Inspecting the 3c prediction distribution suggests that the pipeline produces a realistic FC1 and FC2 mix: predictions concentrate on *DisobeyTaskSpec* and *ReasoningActionMismatch*, broadly matching the ground truth distribution, which is dominated by the same modes. All six correctly predicted traces (strict mode) are FC1 or FC2 failures.

4.3 RQ2: Metadata Availability

To isolate the impact of trace metadata, we run the full pipeline on HyperAgent-SWE and inspect global constraint extraction.

HyperAgent traces contain no system prompts, no tool schema definitions, and no agent role specifications. The global spec extractor consequently produces **zero global constraints** for 13 of 14 traces; one exception yields 4 constraints. All 513 violations in the HyperAgent constraint logs come from dynamic generation alone. This suggests that the absence of structural metadata renders global constraint extraction ineffective: without an external source of role knowledge, the extractor has nothing to work from.

Table 4 shows the pipeline results. Despite the metadata limitation, dynamic constraints enable meaningful localization: the 3c configuration achieves a mean strict mode accuracy of 25.7% and a mean strict family accuracy of 45.7%, compared

Table 3: Failure mode and family accuracy on MAD-Human (18 traces).

Configuration	Mode (strict)	Mode (lenient)	Family (strict)	Family (lenient)
Baseline (no specs)	5.6%	22.2%	22.2%	55.6%
Full Pipeline (5c)	27.8%	38.9%	44.4%	61.1%
Full Pipeline (3c)	**33.3%**	**38.9%**	**50.0%**	**61.1%**

Table 4: Failure mode and family accuracy on HyperAgent-SWE (14 traces). †Mean over 10 independent runs; standard deviation in parentheses.

Configuration	Mode (strict)	Mode (lenient)	Family (strict)	Family (lenient)
Baseline (no specs)†	0.0% (±0.0%)	0.7% (±2.3%)	14.3% (±0.0%)	45.0% (±3.5%)
Full Pipeline (5c)	28.6%	42.9%	42.9%	57.1%
Full Pipeline (3c)†	**25.7%** (±6.5%)	**36.4%** (±10.3%)	**45.7%** (±12.0%)	**58.6%** (±12.3%)

to a mean of 0.0% and 14.3% for the baseline (also averaged over 10 runs).

Examining one representative run, the 3c prediction distribution is an FC1/FC2 mix: *DisobeyTaskSpec* ×10, *IgnoredOtherAgentInput* ×2, *ReasoningActionMismatch* ×2, aligning with the ground truth (DisobeyTaskSpec ×7, UnawareOfStoppingCond ×5). In runs where traces are correctly predicted, they are predominantly *DisobeyTaskSpec* traces, suggesting the pipeline's strongest signal is on FC1 failures.

Answer to RQ2: Our evaluation on HyperAgent-SWE shows that metadata availability has a substantial impact on global constraint extraction quality. On this dataset, traces without system prompts or role schemas yield zero global constraints, and the pipeline must rely entirely on dynamic generation. Dynamic generation alone lifts accuracy above baseline on this metadata-limited setting, though the results suggest that providing external role knowledge would likely yield further gains, a claim that warrants evaluation on metadata-rich traces.

4.4 RQ3: Constraint Characteristics

Constraint type distribution

We analyze the 513 violations in the HyperAgent-SWE constraint logs from one representative run of the Full Pipeline (3c). Table 5 shows the distribution by constraint type and whether the trace was correctly predicted under strict matching in that run (4 correct, 10 incorrect).

The type distribution is similar between correctly and incorrectly predicted traces (PROTOCOL dominates in both at ∼40–45%, TEMPORAL at ∼28–32%). Constraint type alone does not appear to be a reliable discriminator of diagnostic effectiveness.

Taxonomy target distribution

Recall that each constraint carries one or more *taxonomy targets*: the MAST failure mode(s) it is designed to detect (e.g., a CAPABILITY constraint may target *DisobeyTaskSpec*). The taxonomy target distribution of the violation log therefore reflects *which failure modes the fired constraints were aimed at*, and is the key signal the judge uses to select a predicted failure mode.

Table 6 reports the top taxonomy targets across all 513 violations in the representative run. FC2 coordination modes dominate: *ReasoningActionMismatch* (157, 30.6%), *IncompleteVerification* (140, 27.3%), and *IgnoredOtherAgentInput* (89, 17.4%). FC1 modes account for 24.0% (*DisobeyTaskSpec* 79 + *DisobeyRoleSpec* 44). FC3 modes contribute fewer than 5% of all targets.

Constraint vocabulary design

The *constraint vocabulary* refers to the set of failure modes and constraint patterns the generator is prompted to produce. A well-designed vocabulary covers all three MAST failure families, so that the resulting violation log contains taxonomy targets spanning FC1, FC2, and FC3, giving the judge evidence to discriminate between families. In our design, the generator prompt lists all 14 MAST modes grouped by family, with dedicated FC2 patterns targeting coordination phenomena (stopping conditions, inter-agent acknowledgement, repeated-failure detection) and dedicated FC1 patterns targeting task-compliance and role-boundary violations. The five constraint types (PROTOCOL, CAPABILITY, TEMPORAL, RELATIONAL_POST, ANY) provide the syntactic form, but the taxonomy targets they carry determine the diagnostic signal.

The resulting taxonomy target distribution closely mirrors the ground truth failure distribution on both datasets. Correctly predicted traces show a higher proportion of *DisobeyTaskSpec* targets from CAPABILITY and TEMPORAL constraints than incorrectly predicted traces, consistent with the hypothesis that constraints naming the correct failure family steer the judge toward the right prediction. This finding is independent of constraint type: the same PROTOCOL or TEMPORAL constraint can carry an FC1 or FC2 target and produce correspondingly different diagnostic guidance.

The UnawareOfStoppingCond blind spot

All five traces with *UnawareOfStoppingCond* as primary ground truth remain misclassified. Their violation logs show abundant FC2 evidence: *ReasoningActionMismatch* and *IncompleteVerification* dominate, but *UnawareOfStoppingCond* targets appear only sparsely (37 total across 14 traces, compared to 157 for *ReasoningActionMismatch*). Detecting this failure mode requires recognising that an agent continued past a clear termination criterion, which entails counting repeated failed attempts against an expected stopping point. TEMPO-

Table 5: Constraint type distribution in violation logs (HyperAgent-SWE, 513 violations). **Total/%**: counts and share across all traces. **Correct/Incorrect**: share of violations of this type among the 4 correctly vs. 10 incorrectly predicted traces (strict mode), averaged per trace within each group.

Type	Total	%	Correct (avg. %)	Incorrect (avg. %)
PROTOCOL	221	43.1%	40.2%	44.8%
TEMPORAL	152	29.6%	32.0%	27.9%
CAPABILITY	108	21.1%	18.9%	22.4%
RELATIONAL_POST	32	6.2%	9.0%	4.8%
Total	513	100%		

Table 6: Top taxonomy targets in violation logs (HyperAgent-SWE, Full Pipeline 3c, all 513 violations).

Taxonomy Target	Count	%
ReasoningActionMismatch	157	30.6%
IncompleteVerification	140	27.3%
IgnoredOtherAgentInput	89	17.4%
DisobeyTaskSpec	79	15.4%
DisobeyRoleSpec	44	8.6%
UnawareOfStoppingCond	37	7.2%
IncorrectVerification	20	3.9%
Other (7 modes)	31	6.0%

RAL constraints track what must hold at a future step but do not count repetitions; PROTOCOL constraints track ordering but do not measure against a threshold. A dedicated constraint type for stopping-condition monitoring would be needed to close this gap.

Answer to RQ3: Constraint type does not discriminate diagnostic effectiveness: PROTOCOL and TEMPORAL constraints appear at similar rates in both correctly and incorrectly diagnosed traces. The critical factor is the *taxonomy targets* that constraints carry; constraints naming the correct failure family steer the judge toward the correct prediction. This result suggests the design choice of covering all three failure families in the constraint generator and motivates targeted constraint categories for each failure family.

4.5 RQ1: Identification Accuracy

Tables 3 and 4 report the full results; here we summarise the accuracy gains across both datasets. On MAD-Human, the 3c pipeline achieves 33.3% strict mode and 50.0% strict family accuracy, compared to 5.6% and 22.2% for the no-spec baseline, a net gain of +27.7 and +27.8 percentage points respectively. On HyperAgent-SWE, the same configuration achieves a mean strict mode accuracy of 25.7% and a mean strict family accuracy of 45.7% across 10 runs, compared to a mean of 0.0% and 14.3% for the baseline, a net gain of +25.7 and +31.4 percentage points respectively.

Under lenient matching, the 3c pipeline achieves 38.9% mode and 61.1% family accuracy on MAD-Human (vs 22.2% and 55.6% for baseline) and a mean of 36.4% mode and 58.6% family on HyperAgent-SWE (vs a mean of 0.7% and 45.0%). The lenient family gain on HyperAgent-SWE (+13.6 points mean) reflects that the pipeline places predictions in the right failure family more consistently than the baseline.

Answer to RQ1: Our results suggest that spec-based FL can improve failure mode and family identification accuracy over a raw LLM judge on both datasets evaluated. The observed gains are consistent across frameworks and ground truth annotation styles, and appear to be driven principally by the taxonomy target coverage of the dynamic constraint generator rather than by architectural choices in the pipeline.

5 Responsible Research

Data. The MAST MAD dataset [9] is publicly available and was used as-is without modification. For the MAD-Human evaluation we use the 18-trace human-labelled split. For RQ2 and RQ3 we use the 14 failing HyperAgent traces with LLM-annotated ground truth. The constraint generator prompt was designed to cover all three MAST failure families using the complete taxonomy list and was not tuned against dataset ground truth labels.

Model use. All LLM calls use the OpenAI API (GPT-4o and GPT-4o-mini) under standard terms of service. No fine-tuning or model modification was performed.

Threats to validity. LLM-based evaluators introduce non-determinism; results may vary across API versions or runs. To quantify this, the Full Pipeline (3c) and the Baseline on HyperAgent-SWE were each run 10 times; the resulting standard deviations ($\pm$6.5 pp strict mode for 3c; $\pm$3.5 pp lenient family for the baseline) reflect the magnitude of run-to-run variability on this dataset size. The 5c configuration was run only once; for a rigorous statistical comparison between all configurations, the 5c variant would also need multi-run coverage.

Use of AI assistance. Claude (Anthropic) was used to assist with writing and editing this paper. All research decisions, experimental design, analysis, and conclusions are the authors' own.

Replication package. The source code, prompts, and pre-computed evaluation results are publicly available at https://doi.org/10.5281/zenodo.20767030. The dataset can be downloaded from HuggingFace as described in the repository README.

6 Discussion

6.1 Limitations

The human-annotated dataset consists of 18 traces, which is small for strong statistical conclusions. The HyperAgent ground truth uses LLM annotation rather than human annotation, which may introduce labeling noise. Results may not

generalise beyond the six frameworks in MAD or the SWE-Bench-Lite task distribution. The ground truth annotates failure modes but not culprit agents or decisive steps, preventing evaluation of the most specific pipeline outputs.

6.2 Constraint Vocabulary Is the Primary Lever

A key observation from this work is that the taxonomy targets carried by dynamic constraints appear to be a primary factor in spec-based FL effectiveness. The taxonomy target distribution in the violation logs closely mirrors the ground truth failure distribution on both datasets, and correctly predicted traces show a higher concentration of targets matching the true failure family. This finding suggests a practical implication: within a fixed model and pipeline architecture, a key lever for improving spec-based FL may be the constraint vocabulary: prompts that deliberately cover all failure families with appropriate example patterns, ensuring the vocabulary is representative of the full target taxonomy.

6.3 More Constraints Are Not Better

The 5c vs 3c comparison on MAD-Human (Section 4.2) shows that generating five dynamic constraints per step rather than three does not improve accuracy. On an 18-trace dataset, the observed differences (5.5 pp in strict mode, 5.6 pp in strict family) could plausibly fall within run-to-run variation from LLM non-determinism; larger-scale evaluation would be needed to confirm whether the gap is systematic. On HyperAgent-SWE (Table 4), the 10-run mean for 3c (25.7% strict) is close to the single 5c run (28.6%), consistent with the difference being within the observed ±6.5 pp variability range. What is clear is the cost difference: 5c generates 67% more API calls per step, and given that each full pipeline run on 14 traces is time-intensive, running 5c at the same statistical coverage as 3c would require substantial additional resources. We adopt 3c as the recommended budget on cost-efficiency grounds.

6.4 The Metadata Gap

The HyperAgent traces expose a structural limitation of global spec extraction: without system prompts, role definitions, or tool schemas, the extractor produces no useful constraints. A natural remedy is to inject the framework's documented role architecture as external knowledge. HyperAgent defines four agents (Planner, Navigator, Editor, Executor) with explicit responsibilities and a Planner-driven delegation protocol. Encoding these as static role constraints would provide a global constraint foundation even on metadata-sparse traces, and would likely further improve accuracy beyond what dynamic generation alone achieves.

6.5 The *UnawareOfStoppingCond* Gap

The five USC traces that remain misclassified represent a concrete design target for future work. Detecting *UnawareOfStoppingCond* requires counting repeated failed attempts and comparing against a termination criterion; neither PROTOCOL nor TEMPORAL constraints capture this naturally. A dedicated stopping-condition constraint type that tracks attempt counts and expected exit conditions would close this gap without requiring vocabulary changes to the existing constraint types.

6.6 The No-Agent Ground Truth Limitation

Both datasets provide ground truth only at the failure mode and family level. Agent-level and step-level attribution, the most specific outputs of the pipeline, cannot be evaluated here. This is a dataset limitation; the pipeline still produces agent and step attribution that is qualitatively inspectable. Future work should evaluate on datasets with agent-level ground truth, either via fault injection or manual annotation.

6.7 Strict vs Lenient Evaluation

The two datasets differ in how lenient matching is defined. For MAD-Human, three annotators independently vote yes/no on all fourteen modes; any mode receiving at least two votes is an equally valid ground-truth label. The primary label used in strict evaluation is a post-processing artifact (highest vote count, ties broken by taxonomy order), not a property the dataset asserts; lenient accuracy is the methodologically sound primary metric here. For HyperAgent-SWE, annotation was performed by a single LLM run that assigns multiple failure modes per trace with equal weight; lenient matching credits any of those LLM-assigned modes. In both cases strict accuracy is reported for comparability with prior work, but the interpretation differs: on MAD-Human it reflects human annotator consensus, while on HyperAgent-SWE it measures agreement with a single arbitrarily chosen primary label.

7 Conclusions and Future Work

We presented a specification-based fault localization pipeline for LLM-MAS and evaluated it on two complementary datasets spanning six frameworks and fourteen failure modes. The pipeline outperforms a no-specification baseline on both datasets evaluated, suggesting that structured, constraint-based diagnosis is a promising direction for debugging LLM-based multi-agent systems.

A key observation is that diagnostic effectiveness appears to be determined not by the number of constraints but by their semantic content: constraints that name the correct failure family tend to steer the judge toward the correct prediction, while constraints aimed at the wrong taxonomy target are associated with misdiagnosis regardless of their syntactic type. This suggests that improving spec-based FL may be primarily a prompt engineering and taxonomy coverage problem, rather than a pipeline design problem: the architectural scaffolding appears sufficient; a likely bottleneck is generating constraints that map onto the relevant failure vocabulary.

The remaining accuracy gap is concentrated in *UnawareOfStoppingCond* (5 traces on HyperAgent-SWE), which requires a dedicated constraint pattern for repeated-failure detection that current constraint types do not provide. The metadata analysis further shows that traces without system prompts or role schemas yield zero global constraints, making dynamic generation the sole source of diagnostic signal on metadata-sparse datasets such as HyperAgent-SWE.

Future work includes: (1) adding a dedicated stopping-condition constraint type that monitors repeated failed attempts and triggers when an expected exit criterion is not met; (2) injecting framework-documented role architectures as external knowledge into the global extractor, to recover global

constraints on metadata-sparse traces; and (3) evaluating on datasets with agent-level and step-level ground truth to assess attribution accuracy beyond the failure mode and family level.

References

[1] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" *ICLR*, 2024.

[2] H. N. Phan, T. N. Nguyen, P. X. Nguyen, and N. D. Q. Bui, "HyperAgent: Generalist Software Engineering Agents to Solve Coding Tasks at Scale," *arXiv:2409.16299*, 2024.

[3] Q. Wu et al., "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," *arXiv:2308.08155*, 2023.

[4] S. Hong et al., "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," *arXiv:2308.00352*, 2023.

[5] L. Zheng et al., "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," *NeurIPS Datasets and Benchmarks Track*, 2023.

[6] H. Trivedi et al., "AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents," *ACL*, 2024.

[7] J. He, C. Treude, and D. Lo, "LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead," *ACM TOSEM*, 34(5):124:1–124:30, 2025.

[8] C. Qian et al., "Communicative Agents for Software Development," *arXiv:2307.07924*, 2023.

[9] M. Cemri et al., "Why Do Multi-Agent LLM Systems Fail?" *NeurIPS*, 2025.

[10] S. Barke, A. Goyal, A. Khare, A. Singh, S. Nath, and C. Bansal, "AgentRx: Diagnosing AI Agent Failures from Execution Trajectories," *arXiv:2602.02475*, 2026.

[11] R. K. Sharma, S. Barke, and B. Zorn, "Willful Disobedience: Automatically Detecting Failures in Agentic Traces," *arXiv:2603.23806*, 2026.

[12] J. Li, E. Yilmaz, B. Chen, and D.-T. Le, "Towards Self-Improving Error Diagnosis in Multi-Agent Systems," *arXiv:2604.17658*, 2026.

[13] D. Gopinath, R. N. Zaeem, and S. Khurshid, "Improving the effectiveness of spectra-based fault localization using specifications," *ASE '12: Proc. 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 40–49, 2012.

[14] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," *ASE*, pp. 273–282, 2005.

[15] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[16] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, "Generative Agents: Interactive Simulacra of Human Behavior," *UIST*, 2023.

[17] Z. Xi et al., "The Rise and Potential of Large Language Model Based Agents: A Survey," *arXiv:2309.07864*, 2023.

[18] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," *NeurIPS*, 2024.

[19] OpenAI, "GPT-4 Technical Report," *arXiv:2303.08774*, 2023.

A Pipeline Data-Flow Diagram

Figure 3 illustrates the data flow through the key pipeline stages using the motivating example from Section 1.

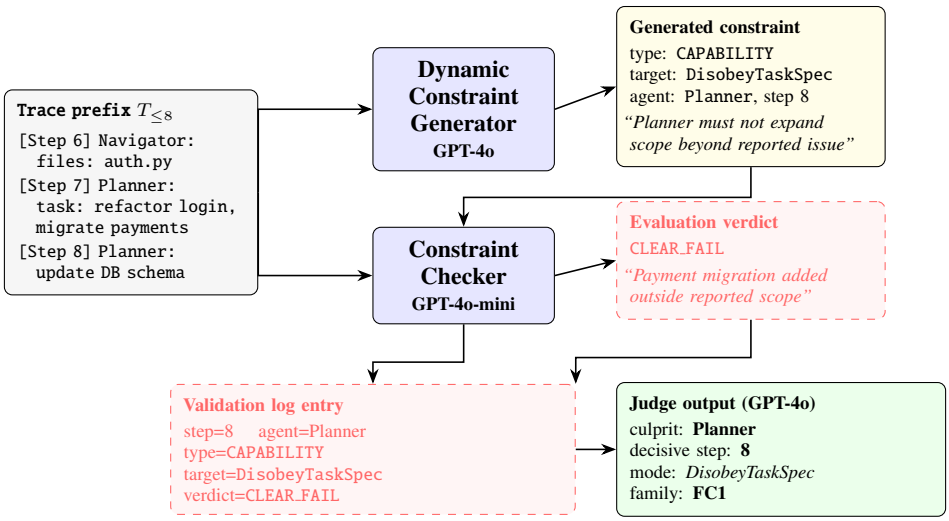

Figure 3: Data flow through the spec-based FL pipeline for the motivating example. The trace prefix $T_{\leq 8}$ feeds the dynamic constraint generator (Stage 2b) and the constraint checker (Stage 3). The generated CAPABILITY constraint targeting *DisobeyTaskSpec* is evaluated at step 8, returns `CLEAR_FAIL`, and is recorded in the validation log (Stage 4). The judge (Stage 5) reads the log and outputs the failure diagnosis.

B Example Constraint and Violation Log

This appendix shows a representative constraint generated by Stage 2b and the corresponding violation log entry produced by Stages 3–4, using the motivating example from Section 1.

Generated Dynamic Constraint (Step 8)

```
{
  "assertion_name": "planner_scope_constraint",
  "constraint_type": "CAPABILITY",
  "taxonomy_targets": ["DisobeyTaskSpec"],
  "trigger": {
    "step_index": 8,
    "agent_id":   "Planner"
  },
  "check_type": "nl_check",
  "check_hint":
    "The Planner must not assign subtasks whose scope exceeds
     the originally reported issue. Any work not directly
     required by the issue report constitutes a scope violation."
}
```

Constraint Evaluation (Stage 3)

The constraint is passed to the LLM evaluator together with the content of step 8. The evaluator returns:

```
{
  "assertion_name": "planner_scope_constraint",
  "step_index":     8,
  "agent_id":       "Planner",
  "verdict":        "CLEAR_FAIL",
  "reasoning":
    "At step 8 the Planner assigns a DB schema migration task that was
     not part of the reported issue. The reported issue concerned only
     the login flow refactor; the payment migration constitutes an
```

```
    out-of-scope expansion."
}
```

Validation Log Entry (Stage 4)

After deduplication (first firing only), the validation log $\mathcal{V}$ records:

```
Validation Log
==============
[VIOLATION] step=8 | agent=Planner
  constraint : planner_scope_constraint
  type       : CAPABILITY
  source     : dynamic
  targets    : DisobeyTaskSpec (FC1)
  verdict    : CLEAR_FAIL
  reasoning  : "Planner assigned payment migration outside reported scope"

[PASS]      step=9  | agent=Editor   | constraint=editor_patch_complete
[PASS]      step=10 | agent=Planner  | constraint=planner_scope_constraint
             (re-evaluation suppressed -- first firing at step 8 retained)
```

The validation log, together with the full trajectory and the MAST taxonomy checklist, is passed to the GPT-4o judge in Stage 5, which produces:

```
{
  "culprit_agent":    "Planner",
  "decisive_step":    8,
  "failure_mode":     "DisobeyTaskSpec",
  "failure_family":   "FC1",
  "reason_for_step":  "Step 8 is where the Planner first expanded scope
                       beyond the reported issue, as flagged by the
                       CAPABILITY constraint violation.",
  "reason_for_category":
                      "The failure is DisobeyTaskSpec because the Planner
                       violated the task specification by adding work not
                       requested in the issue report."
}
```