Can Timing Localize Agent Failures?

Incorperating a temporal dimension into spectrum-based fault localization for LLM multi-agent systems

Hein Schouwenaars

Supervisor(s): Burcu Kulahcioglu Ozkan[1], Annibale Panichella[1], Zahra Seyedghorban1[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 21, 2026

Name of the student: Hein Schouwenaars
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

Large language models are fallible and indeterministic, which makes fault localization harder than in traditional deterministic software. It gets harder still in LLM-based Multi Agent Systems (LLM-MAS), where a fault can be introduced by any agent at any point, rarely reproduces, and tends to propagate throughout the rest of an execution. Spectrum-based fault localization (SBFL) has worked well on classical software, and prior work has applied it to LLM-MAS with promising but substandard accuracy. This research proposes a new spectrum, one based on temporality. Each spectrum element is defined by an agent, an action, and the temporal window in which the action occurs. These windows can take many shapes; we compare four: relative static, absolute static, sliding, and dynamic windowing. The spectra are built from a new dataset of HyperAgent traces collected over three SWE-Bench Verified tasks, with the ground-truth faults and message labels assigned by an LLM-as-a-judge, and each windowing is evaluated by top-k accuracy. Relative static windowing with a high partition count performed best. The other strategies showed little consistent improvement over the baseline. Temporal ordering can therefore be correlated with faults, but the agent-action-window spectrum does not yet capture them reliably. A richer, denser spectrum will be needed to reach a workable level of accuracy.

1 Introduction

Large language models (LLMs) have become ubiquitous in our lives, finding applications in a large number of fields. However, LLMs do not come without drawbacks. They are fallible and indeterministic, making fault localization harder than in traditional deterministic software.

Large language model-based multi-agent system (LLM-MAS) have become an increasingly popular paradigm. An LLM-MAS allows a single LLM *agent* to talk with and delegate tasks to other LLM agents within the system. While this paradigm has grown in popularity due to its potential to reduce faults, localizing faults *when* they occur has become increasingly difficult. Faults can be introduced by any agent at any point in the execution, are not reproducible due to the indeterministic nature of LLMs, and tend to propagate and snowball throughout an execution.

These complexities require an evaluation of which classical debugging strategies transfer to LLM-MAS debugging. One such strategy is spectrum-based fault localization [1] (SBFL), a technique that found great use when applied to deterministic software. SBFL assigns a *suspiciousness score* to program elements based on their contribution to passing and failing test cases. SBFL estimates this by comparing how often an element is executed in passing and failing runs. The new LLM-MAS paradigm requires abstracting LLM execution steps to program elements. Abstraction is achieved by grouping execution steps on common traits. Prior research has examined one such grouping to test the effectiveness of SBFL on LLM-MAS executions [7], yielding promising but substandard accuracies.

Nevertheless, the granularity at which steps are grouped can be re-imagined to capture the intricacies of an LLM-MAS. More concretely, this research will *evaluate the effectiveness of incorporating a temporal dimension into spectrum-based fault localization on large language model-based multi-agent systems*. Since each step of an execution depends on the output of its indeterministic predecessors, assessing the role of *temporal order* alongside that of *agent* and *action* may prove useful.

this temporal dimension is realized by grouping steps into windows. We compare four such strategies: absolute static, relative static, sliding, and dynamic, and ask to what extent they differ in their fault-localization accuracy. Most offer little improvement over the baseline, with one exception: relative static windowing. Its benefit suggests that temporal order can correlate with faults, even if the proposed spectrum does not yet localize them reliably on its own.

The rest of this paper is organized as follows. Section 2 gives background information on LLM-MAS and SBFL and Section 3 reframes SBFL, introduces windowing over the temporal dimension, and states the research questions. Section 4 describes the methodology; how traces are collected, how the spectrum is built, and how each windowing is evaluated. Section 5 reports the technologies and results. Section 6 addresses responsible research, Section 7 discusses threats to validity and the cost of simplifying a complex system, and finally Section 8 concludes .

2 Background

2.1 Agents & Multi Agent Systems

An LLM-agent consists of persistent data and dynamic memory. This persistent data is usually referred to as the agent *role*. A role consists of three parts: an instruction set, an observability sphere, and an action set. The instruction set is the prompt that defines the agent's goals. The observability sphere is the rule set that decides what the agent can observe. And the action set defines the *actions* the agent can take. This include tool calls and agent-to-agent messages. More formally an agent role can be defined as:

$$r_a = (\mathcal{I}, \mathcal{O}, \mathcal{A})$$

Where $\mathcal{I}$ represents the instruction set, $\mathcal{O}$ the observability sphere, and $\mathcal{A}$ the action set. **We represent an agent by it's role.**

Thus an LLM-MAS is a directed graph $G = (\mathcal{R}, E)$, where $\mathcal{R}$ is a set of agents and $(r_i, r_k) \in E$ if agent r_i has an action $a \in \mathcal{A}_i$ targeting r_k. An action targets an agent if it is directed at that agent, either as an LLM invocation or an

inter-agent message. G is weakly connected, meaning each node can reach any other node if the directions are ignored.

One execution of an LLM-MAS can be abstracted to an ordered list of agent *steps*. Where each step s consists of an agent and an action, as originally proposed in [7].

$$s = (r, a)$$

2.2 LLM-MAS Architectures

An LLM-MAS derives its identity from the set of roles it contains. A range of frameworks have been proposed, adapted, and applied in practice. Below we review a few influential systems that have shaped the current baseline for software-development tasks.

ChatDev ChatDev [11] was an early example of a multi-agent system for software development, later adapted to a wider range of tasks. It framed the LLM-MAS as a way to reduce *LLM hallucinations*: ChatDev distributes the work across several agents whose roles reflect those of a real software company, such as a CEO, a CTO, programmer and tester. It also introduced a workflow built around *phases*, in which an agent's goal is determined by the project's current stage. These phases progress from design to coding to testing. The phase-based approach was not widely adopted by later systems, however, and was dropped in subsequent versions of ChatDev itself.

Magentic-One Magentic-One [6] is a generalist multi-agent system from Microsoft that broadens the scope of the LLM-MAS beyond software development to a wide range of tasks. It targets what the authors call *complex* tasks: those that require, or clearly benefit from, repeated cycles of planning, acting, observing, and reflecting. The system addresses these tasks by organizing the work around a single lead agent, the *Orchestrator*, which plans the overall approach, tracks progress, and re-plans when something goes wrong. The Orchestrator does not carry out the work itself; instead it delegates to specialized agents such as a FileSurfer for navigating local files, a Coder for writing code and a ComputerTerminal agent for running code.

HyperAgent HyperAgent [10] is a multi-agent system built specifically for software development, designed around the way a human developer works through a task. It organizes its agents around the natural stages of solving a programming problem. The workflow is split across four agents: the *Planner*, which interprets the task, breaks it into steps, and coordinates the others; the *Navigator*, which searches and reads through the codebase to find the code relevant to the task; the *Code Editor*, which makes the concrete changes to the source; and the *Executor*, which runs the code and tests to confirm whether the change behaves as intended. These roles each have a small team of sub-agents that assist in the completion of the required task.

What stands out across these systems is how little they actually disagree. Each was built by a different group with different ambitions, yet they keep arriving at the same handful of roles: an agent to plan, one to read the code, another to write it, and one to run it. The labels change from one framework to the next, but the underlying division of labor barely does, and it tends to look a lot like how a human team would split the same work.

2.3 Benchmarks

Many benchmark tests have been created to verify the accuracy at which LLM-MAS can complete tasks successfully. These benchmarks enable comparison and troubleshooting of LLM-MAS. They provide a set of tasks with a ground-truth solution, Which span across fields. In software engineering one such benchmark is *SWE-Bench* [8]. SWE-Bench is built from real-world GitHub issues, each paired with the tests that accompanied the original fix. The LLM-MAS is prompted to propose a solution to each issue. The tests act as a ground-truth: a proposed solution is judged by whether it passes them. A higher passrate implicates a 'better' LLM-MAS. Passing the relevant tests does not guarantee passing the entire suite however, some tests may be faulty or unrelated to the issue. *SWE-Bench Verified* [4] was introduced to address this. A team of human coders established a golden solution for 500 tasks; matching the test results of that golden solution counts as a success for the LLM-MAS. By using benchmarks one can reliably label LLM-MAS executions as faulty.

2.4 Spectrum Based Fault Localization

Spectrum-based fault localization (SBFL) is a popular fault localization technique within classical software. The *spectrum* is compromised of a collection of program elements. In classical software these program elements are single lines of code or nodes from the abstract syntax tree. These program elements get assigned a *suspiciousness score*, which estimates how likely it is to contain the fault.

Suspicousness scores are computed by *suspicousness formulae*. These formulae are based on four counts:

	Failing tests	**Passing tests**
Executed	e_f	e_p
Not executed	n_f	n_p

Table 1: The four SBFL counts per program element.

Where e_f is the number of failing tests that executed the element, e_p the number of passing tests that executed it, n_f the number of failing tests that did *not* execute it, and n_p the number of passing tests that did not. Suspiciousness formulae are functions of these four counts that compute the elements suspicousness score

Understanding these formulae in depth is not necessary to grasp the effectiveness of SBFL. Each formula conceptually achieves the same goal: Capturing the element that is most responsible for failing tests. Which fault or passing types are prioritised differentiates the formulae. The effectiveness of that prioritisation depends on the underlying dataset. We shall apply each of these five suspiciousness scores and find the formula best fitting the spectrum.

Formula	Suspiciousness
Kulczynski2	$\frac{1}{2}\left(\frac{e_f}{e_f+n_f}+\frac{e_f}{e_f+e_p}\right)$
Ochiai	$\frac{e_f}{\sqrt{(e_f+n_f)(e_f+e_p)}}$
Jaccard	$\frac{e_f}{e_f+n_f+e_p}$
DStar (2)	$\frac{e_f^2}{e_p+n_f}$
Op2	$e_f-\frac{e_p}{e_p+n_p+1}$

Table 2: Five common SBFL suspiciousness formulae, expressed in terms of the counts from Table 1.

2.5 Trace Datasets

Several datasets of LLM-MAS traces on benchmark tasks have been compiled in prior work. Among the most notable are Who&When [12], MAST (MAD) [2], and TRAIL [5]. Not every dataset fits SBFL, however. The traces must be *consistent*, all coming from the same LLM-MAS and task. They must be *verifiable*, each with a ground truth for the fault location. And they must be *complete*, containing both passing and failing traces in statistically significant quantities, since a spectrum needs pass/fail contrast to be useful.

None of the existing datasets meet all three. Who&When and TRAIL collect only failing traces, so there are no passing runs to contrast against. MAD, though large in volume, spreads its traces too thin across different agentic systems and tasks.

3 Defining the Research

Applying classical fault-localization techniques to LLM's in Multi Agent Systems requires new definitions. For our adaptation for temporality we introduce new definitions in this section. Later in the section we use those definitions to pose the main research question.

3.1 SBFL in a new light

SBFL relies on two major inputs: The spectrum elements and a ground-truth test-suite. Running LLMs on benchmarks like SWE-Bench Verified conveniently supplies the latter. Because mistakes snowball within an LLM-MAS, counting failures across a multi-test suite is biased: the same mistake appears more faulty the earlier it occurs, even though in practice either case simply calls for a rerun. A binary pass/fail label is therefore more fitting; if the proposed code passes the entire test-suite, the run is considered passing.
The spectrum elements are more open to reinterpretation. Each step in an execution trace is a natural baseline.

FAMAS [12] proposed the first spectrum for this setting. Its element is an agent–action–state triple. State is whatever the action returned, in natural language. Thus they build te spectrum by grouping by agent, then merges those whose action–state an LLM judges equivalent. Using their suspicousness formula they reached roughly 58% agent-level attribution, which is not enough to apply in practice despite showing promise.

This research proposes a new spectrum, one based on temporality. It keeps the agent and the action, but in place of the state it asks *when* the step occurred. A spectrum element is defined by the agent and action, together with the temporal window in which the action occurs. Windows group execution steps: rather than asking how often a step occurs in failing and passing runs, this spectrum asks a sharper question—how often does a step occur in failing and passing runs within the i'th group of steps. These windows can take many shapes and sizes. Four proposals are listed below.

3.2 Windowing

An LLM-MAS execution as an ordered list of steps allows the grouping of steps into *windows*. Given an execution $S = \langle s_1, s_2, \ldots, s_n\rangle$, a *windowing* is a sequence of windows $W = \langle W_1, W_2, \ldots, W_k\rangle$ where each window W_i is a contiguous subsequence of S.

All window types satisfy the *ordering constraint*: W_i must start and end earlier in the execution than W_j starts and ends, if and only if $i < j$

Relative Static Windows. Steps are distributed into k non-overlapping, exhaustive windows as evenly as possible, each containing $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$ steps.

Absolute Static Windows. Each window has a fixed size w, so the number of windows depends on the trace length. The final window may be smaller if $n \bmod w \neq 0$.

Sliding windows. Like absolute static windows, but consecutive windows may overlap by a fixed number of steps O, where $0 \leq O < w$. if $O > w$ then O is clamped to $w - 1$ Where w is equal to $\lfloor n/k \rfloor$

Dynamic windows. Window boundaries are placed not at fixed intervals but wherever control passes between agents: each step is attributed to its owning main agent, and a boundary falls between consecutive steps owned by different agents. A parameter $spacing$ pads each transition with a buffer of $spacing$ steps on either side. $spacing = 0$ cuts exactly at the transition. If two buffer windows overlap, they are merged.

In figure 1 you can find a visual representation of each technique.

3.3 Research Questions

The goal of the research is to compare these three methods and explore the merit of grouping the steps in a temporal manner. This brings us to the following research question:

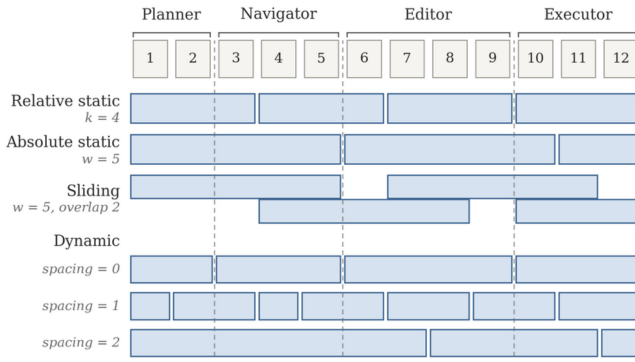


Figure 1: Visual representation of the four windowing techniques

RQ: To what extend do temporal static window, sliding window, and dynamic window based spectra differ in their spectrum-based fault localization accuracy when applied to LLM-based multi-agent systems?

This divides nicely into three subquestions:

RQ1: To what extent can faults be accurately detected using spectrum-based fault localization with temporal static windows in LLM-based multi-agent systems?

RQ2: To what extent can faults be accurately detected using spectrum-based fault localization with temporal sliding windows in LLM-based multi-agent systems?

RQ3: To what extent can faults be accurately detected using spectrum-based fault localization with temporal dynamic windows in LLM-based multi-agent systems?

4 Methodology

4.1 Trace Data

Existing LLM-MAS trace datasets do not adhere to the requirements needed to apply SBFL with statistical significance. Therefore we created a new dataset. HyperAgents role division appendix ref is examplary of many multi agent systems and its prominence in research makes comparison easy. Thus we built the dataset using the HyperAgent system.

We ran Hyperagent on three different SWE-Bench verified tasks and classified the failing execution traces by failure modes. Failed runs with LLM unrelated failures, like API-failures or Tool timeouts are discarded. Two failure modes remain: empty patches and failing tests. An empty patch confirms the system failed to output a valid final result, while failing tests imply the final solution ran, yet turned out incorrect.

We attributed fault location through LLM-as-a-judge on failed runs and we fed the failed runs into an LLM which is prompted to locate the fault based on the failing mode. We used the LLM-as-a-judge paradigmto create a ground truth to compare the resulting suspiciousness scores against.

Task	# Logs	Pass %
pallets__flask-5014	165	33.3
psf__requests-1142	81	70.4
psf__requests-1766	96	49.0

Table 3: Tasks, number of run logs, and pass rate.

4.2 The Spectrum

In order to apply SBFL and test correlation with temporality, we must define a spectrum. Each spectrum element consists of one agent step, and the window in which the step appears.

$$\langle Role, Action, Window \rangle$$

Role is explicit within an execution trace. Action refers to an agent-to-agent message or a tool invocation. Messages come in many shapes and sizes and need to be classified to extract a steps core purpose. Classification is done using a LLM classification process. Each message is classified into one of six categories:

- Instruction – directs another agent or sets the agenda: delegates a subtask, specifies a subgoal, issues a command or request for action, or hands work off.
- reasoning – the sender's own thinking, not yet a result or an order: forming or revising a hypothesis about the bug, interpreting earlier evidence, weighing options, or planning the next step.
- exploration – gathering or requesting information about the code or problem: locating files, symbols, or definitions, reading or describing existing code, or summarising what a search turned up.
- code – concrete code is the substance of the message: a proposed edit, a patch or diff, a function body, or a code block meant to be applied or reviewed.
- report – communicates an outcome: results or observations from an action, interpretation of tool/test/build output, a status update, or a final conclusion that the task is done or has failed.

The final step is to group the role – action pairs into windows by the different windowing techniques, which completes the building of the spectrum. We then count how often each agent–action–window triple occurs using a participation matrix. In section B you can find a compressed example of a participation matrix. Showing exactly how often an action-agent-window triple occurred in passed and failed runs (e_p and e_f). The n_p (not in passed run) and n_f (not in failed run) counts have been left out for readability.

4.3 Evaluation

We evaluate the effectiveness of each windowing by top-k analysis. The k most suspicious elements are compared to the ground-truth faults, as attributed by an LLM. We count only elements distinct by agent and action. The windowing is merely used to find the ranking, not to evaluate it. This is because the windows spread the faults too sparsely in most

instances. We group the ground-truth failures by agent and action, and rank them on the number of occurrences. The final score of a run is defined by:

$$\text{Score} = \frac{|\hat{F}_k \cap G|}{|G|}$$

where $\hat{F}_k$ is the set of the k most suspicious elements (each a distinct agent–action pair) and G is the set of ground-truth faulty elements; the score is therefore the fraction of the ground-truth faults that the windowing successfully ranks within its top k elements, ranging from 0 when none are recovered to 1 when all are.

5 Technologies & Results

5.1 Technologies

To collect the execution traces we used HyperAgent with deepseek-chat. Using deepseek(-chat) was a necessity for budgeting reasons. This required some small changes to the hyperagent infrastructure for compatibility reasons. The same deepseek-chat configuration was used for message classification.

To set the ground-truth fault allocation, we used a more intensive deepseek configuration: Deepseek-pro on max reasoning. This was necessary to have the ground-truth be as accurate as possible.

5.2 Results

Of the four windowing techniques relative static windowing performed best. In table 4 the final scores can be seen computed by the Kulczynski2 suspiciousness formula, which had the best performance on average. The scores are based on the parameter values that scored best on average over the k values. In the case of sliding windows the highest scoring overlap was chosen by averaging over all k values and all window sizes; after this the window size was then chosen like the rest of the parameters. The full results can be seen in section C of the appendix

Score (%)	k=1	k=3	k=5	k=7	k=10
Windowing strategy (Kulczynski2)					
Static (Absolute, w=3)	0.0	25.0	25.0	33.3	49.9
Static (relative, k=39)	**27.8**	**35.2**	**36.9**	**39.6**	**52.8**
Sliding (w=1, O=38)	10.0	25.0	29.2	32.6	35.3
Dynamic (*spacing*=2)	10.0	10.0	25.0	29.2	44.7
Random (weighted)	2.9	8.7	14.5	20.2	28.7
Baseline (one window)	1.4	5.6	29.2	31.9	34.5
Ceiling (best possible)	31.9	63.4	75.1	82.8	91.4

Table 4: Top-k fault-localization score (%), averaged across the three tasks. *Random*: Random suspicousness ranking, weighted by occurance count. *Baseline*: relative windowing with k = 1, using the best formula averaged over k (Kulczynski2). *Ceiling*: best attainable. Best per column in **bold**.

5.3 Understanding the Results

Interpreting the Results Table 4 suggests that windowing has a positive impact on the effectiveness of SBFL when applied to LLM-MAS, with relative static windows with high partitions numbers performing best. The other windowing techniques do not reliably outperform the baseline over different k values. The full dataset paints a similar picture. Those windowing techniques do not consistently outperform the baseline. Relative static windows appear to have a beneficial influence on SBFL in the context of LLM-MAS, but more research is required to back this claim with statistical significance. The other proposed windowing techniques show little correlation to the accuracy of SBFL.

Answering the research questions

RQ1 (static). Temporal static windows aid SBFL, but only in relative form. Relative static windowing with a high partition count recovers 27.8% of faults at k=1, rising to 52.8% at k=10, and beats the one-window baseline at every k. By 26.4 points at k=1 and still 18.3 at k=10. Absolute static windows are far weaker: 0.0% at k=1, reaching 49.9% only at k=10, it beats the baseline mostly on high k values, which is not enough to conclude reliable improvement.

RQ2 (sliding). Sliding windows give little reliable gain. They lead early. 10.0% vs. 1.4% at k=1, but converge on the baseline as k grows, peaking at just 35.3% against the baseline's 34.5% at k=10. High-overlap performed best on average, but only by a small margin. In the best-case the window size is 1, meaning the overlap does not contribute at all. This suggest sliding windows are barely— if at all — better than absolute static windows.

RQ3 (dynamic). Dynamic windows get outperformed by the baseline. They reach 44.7% at k=10, yet sit ahead of the baseline at k=1 and k=3 and behind it at k=5 and k=7. This suggest the proposed form of dynamic windows does not improve SBFL accuracy; it may even contribute to worse accuracy.

Main RQ The strategies differ most at tight k. Relative static wins every column. At k=1 it recovers 27.8% of faults, while sliding and dynamic manage only 10.0 and absolute static finds none. The ranking then reshuffles as k grows. By k=10 absolute static has increased from worst to second, dynamic performed close to average, and sliding performed worst. So the sharpest divide is within the static family: relative and absolute static perform drastically different at low k. The other strategies perform close to the baseline. None of the four comes close to the ceiling score at $k \neq 1$, and their accuracy differences narrow as we test them on higher k values. Relative static windows shows the best potential to improve SBFL accuracy; it does not yet achieve workable accuracies with the proposed spectrum.

Why do relative static windows perform best? The answer starts with the faults themselves. Two agent–action pairs are attributed the most faults: the Planner issuing a *instruction*, and the Inner-Editor-Assistant invoking the Editor tool. The first pair fails reliably at two moments;

the very first step, or the very last. Either the agent never produces a good initial plan, or it declares the task done too early. Both moments land in the same windows under relative static windowing, and a small window isolates them. This is why a high partition count performs best.

The second pair is harder. Its *faults* cluster early: 8 of 14 fall within the first 30% of an execution. Its *occurrences* do not cluster in the same way. The pair recurs throughout the run, so no single window isolates it, and relative static windowing cannot separate its faulty occurrences from the rest. So a temporal spectrum catches only the first fault type reliably. It aids fault localization, but it does not reach a complete solution.

6 Responsible Research

Data & Ethics The dataset is built from SWE-bench verified, which is derived from open-source github projects and the tests that accompanied their fixes. This research did not make use of any human, personal or otherwise private data. The deepseek models used require payment, but these models can be run locally, and the result of all queries have been made openly available.

Reproducability Three steps within the first part of the pipeline are LLM-based. Because of the indeterminism of LLM's reproducing these outcomes is impossible. These steps include the original trace collection, message classification and the failure attribution judged by an LLM. To enable reproducability of the second part of the pipeline the results of the first part of the pipeline are published publicly. The second part of the pipeline, consisting of creating the windows, applying SBFL, evaluating the suspicousness ranking and plotting the results are all deterministic processes and can be reproduced.

Use of generative AI generative AI was used to assist in the writing of this report. In addition AI was used for a few coding tasks, within the published repository AI generated code is marked. All contributions from generative AI were manually reviewed.

7 Discussion

7.1 Threats to validity

LLM-based ground-truth The evaluation is based on faults attributed by an LLM. Given the size of the dataset and the resources available, this was the only feasible choice. The same holds for the message classification; it too is done by an LLM. So both the spectrum and the faults it is scored against lean on a model known to be imperfect.

The accuracy of these attributed faults cannot be determined easily. Verifying them would require a known fault location to compare against, which is precisely what this research aims to uncover. Without an external reference the labels can only be taken at face value. A solution may be a human-annotated dataset, although the validity of human annotation could also be brought into question.

Overfitting The best parameter values are selected over only three tasks. With so few tasks the chosen settings may fit this particular set rather than a property that holds in general, so the discovered optima are at risk of overfitting. Whether they transfer would need a larger and more varied set of tasks to confirm.

Occurrence bias The earliest suspiciousness formula is Tarantula [9]. It has fallen out of use because it does not penalize low occurrence: an element seen once, in a single failing run, is always maximally suspicious. On a sparse spectrum these barely-tested elements rise to the top and infect the ranking, and windowing — especially at a high partition count — produces exactly this kind of spectrum. Formulae that aim to prevent it can only do so by adding bias of their own; bias toward high-occurrence elements. The elements that relative static windowing deems suspicious often have that characteristic. The Planner–request pair, in particular, starts and ends a large number of runs. So the result carries a caveat: these formulas reward high fault occurrence, rather than just a high fault rate.

7.2 Simplifying a complex system

SBFL relies on breaking down a system into program elements. In classical software breaking down these elements does not lose information, this changes when applying the methodology to SBFL. LLM-MAS systems are complicated, and grouping complex messages into agent-action pairs loses a lot of crucial information. A recent study [3] on LLM-as-a-judge found that increasing the context has a significant benefit to the accuracy of fault attribution . This serves as a reminder that localizing fault becomes increasingly hard if we continually abstract and lose important information. Extending this to SBFL suggest a complex multi-parameter spectrum may be the only way to achieve a workable level of accuracy. This research showed that including a temporal dimension into that complex spectrum may be beneficial. Unfortunately SBFL in this world of LLM-MAS requires abstraction. The question remains if a spectrum can be found which captures the complexity of a multi agent system while maintaining a workable spectrum density.

8 Conclusions and Future Work

8.1 Conclusions

This research found that for most of the proposed techniques, temporal windows do little to change the accuracy of SBFL on LLM multi-agent systems. The relative static windowing technique was the exception; it did appear to beneficially influence SBFL accuracy. Thus temporal ordering can be correlated to faults. This encourages further research into how that ordering might be applied, both to SBFL and to other fault localization techniques. The proposed spectrum of agent, action, and window elements does not seem to capture faults within LLM-based multi agent systems reliably. With the best windowing technique localizing A richer spectrum—possibly one containing relative static windows—will need to be built to reach a workable level of accuracy with SBFL.

8.2 Future Work

The future of SBFL in LLM-MAS Section 7.2 covered how abstraction makes adapting SBFL to LLM-based multi-agent systems a difficult paradigm shift. LLM-MAS are indeterministic, so re-runs are impossible, and steps must be grouped into spectrum elements instead. The harder question is how to group them well. Future work should look for grouping techniques that combine into a complex, dense spectrum. One natural next step is to combine both the system state from [12] and relative static windowing into a single spectrum. Another spectrum could come from grouping by agent-to-agent interactions. A final proposition is to expirement with non-temporal windowing, one could imagine other orderings. An example could be to order steps by their closeness to task completion, possibly judged by an LLM.

LLM-as-a-judge Abstracting the complexity of LLM-MAS is often achieved by LLM-as-a-judge. All LLM-MAS focussed fault localization techniques that have been tested before use it in some way or another. As of now, it still achieves the best results when used as a standalone fault localization technique. Until a cheaper or faster reliable fault-localization can be found. This paper attempted to use LLM-as-a-judge to aid the accuracy of classical software fault localization techniques. Another paradigm is the opposite: Using classical techniques to aid the LLM in it's judging. Classical techniques have shown a number of metrics that have worked with fault localization in the past. Finding metrics that transfer well, like specifications in specification-based fault localization opens a new palette of LLM judges.

LLM trace data The lack of fitting datasets is a blocking factor in fault localization research. Generating new traces takes loads of time and resources, and it rules out comparison. So the need for a complete, ground-truth labelled dataset is large. Such a dataset must cover multiple failure modes. It must have human-labelled ground truth. And it must contain enough traces, across multiple LLM-MAS and tasks, to be statistically significant.

A HyperAgent

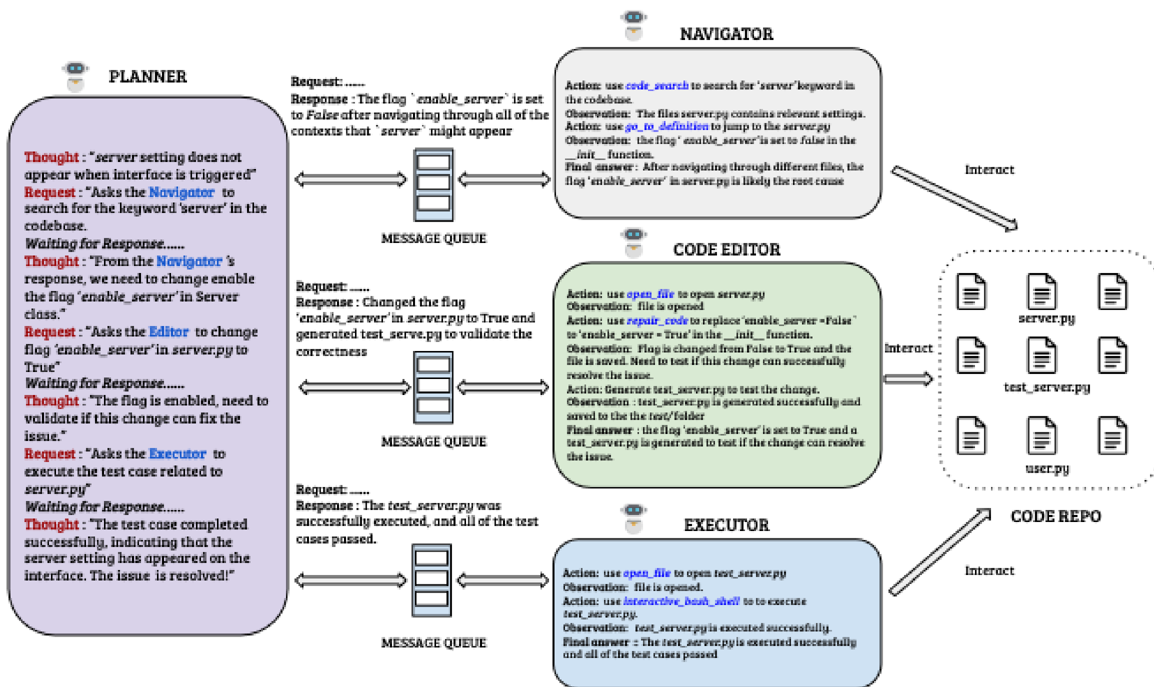


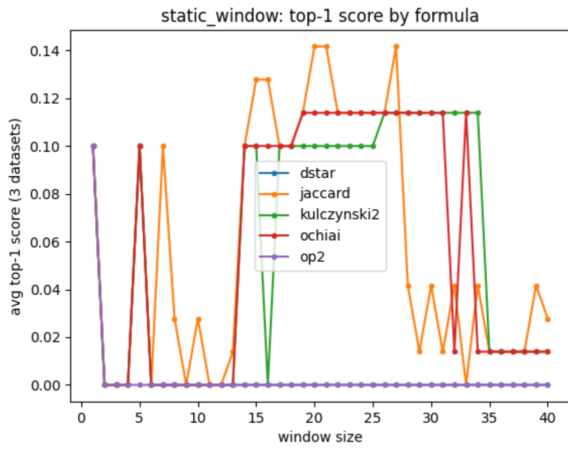
Figure 2: HyperAgent structure

B Example of Participation Matrix

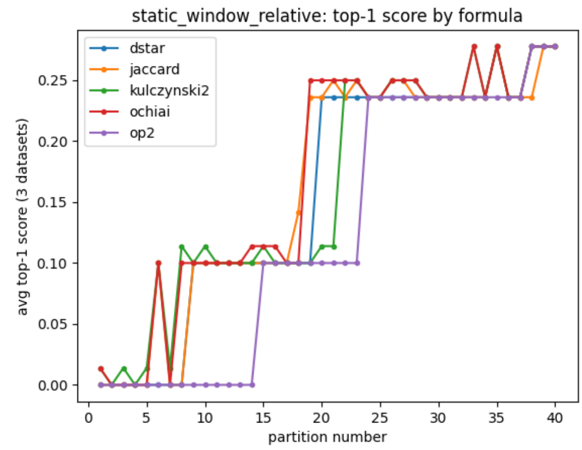
	Failed (n=49)			Passed (n=47)		
Agent / action	w_1	w_2	w_3	w_1	w_2	w_3
Executor Interpreter / report	0	58	199	6	122	382
Editor Interpreter / report	61	167	58	94	242	40
Navigator Interpreter / report	213	72	53	233	56	17
Inner-Navigator-Assistant / exploration	195	69	39	210	49	12
Inner-Editor-Assistant / open_file_gen	61	125	43	82	186	25
Planner / instruction	78	65	94	99	71	104
Inner-Editor-Assistant / exploration	52	128	30	82	185	21
Inner-Navigator-Assistant / open_file	123	59	47	160	30	14
Inner-Executor-Assistant / exploration	0	26	95	3	70	142
Inner-Executor-Assistant / instruction	0	23	75	5	41	139
Inner-Executor-Assistant / report	0	11	61	0	7	124
Inner-Editor-Assistant / editor	26	45	16	34	57	22
Inner-Editor-Assistant / instruction	33	37	17	41	56	12
Navigator / instruction	55	19	31	56	11	17
Executor / instruction	0	17	51	2	26	79
Executor / report	0	7	55	0	12	97
Editor / instruction	26	36	14	39	36	14
Editor / code	3	35	30	7	46	28
Editor Interpreter / instruction	22	31	9	21	50	14
Inner-Navigator-Assistant / find_file	59	8	4	42	6	1
Inner-Editor-Assistant / code	9	32	15	13	32	19
Navigator / report	24	16	22	38	7	10
Inner-Navigator-Assistant / report	23	10	19	40	4	9
Inner-Navigator-Assistant / get_folder_structure	34	13	4	36	12	2
Inner-Navigator-Assistant / instruction	25	13	9	37	7	6
Inner-Navigator-Assistant / code_search	29	8	2	38	5	2
Inner-Executor-Assistant / reasoning	0	12	21	0	16	35
Planner / report	0	1	33	0	0	48
Inner-Navigator-Assistant / reasoning	24	6	10	28	5	8
Navigator Interpreter / exploration	20	12	1	39	0	3
Inner-Editor-Assistant / open_file	6	17	3	9	27	1
Inner-Editor-Assistant / reasoning	7	13	5	8	22	7
Inner-Editor-Assistant / get_folder_structure	4	16	5	5	23	5
Executor Interpreter / code	0	6	20	0	5	25
Inner-Navigator-Assistant / get_all_symbols	17	5	0	17	2	0
Navigator / exploration	5	3	13	10	2	6
Inner-Executor-Assistant / code	0	1	4	0	4	21
Navigator / code	4	4	4	7	3	6
Editor Interpreter / exploration	5	3	0	6	6	1
Inner-Editor-Assistant / report	3	4	5	3	1	4
Inner-Navigator-Assistant / code	4	5	5	2	1	1
Planner / reasoning	5	4	1	4	2	1
Editor / report	3	2	2	2	3	3
Executor Interpreter / exploration	0	1	2	1	1	6
Planner / exploration	2	2	2	2	0	1
Navigator Interpreter / instruction	2	2	1	3	0	0
Navigator / reasoning	1	2	1	1	1	1
Executor / code	0	0	2	0	1	2
Executor / exploration	0	0	2	0	2	0
Inner-Navigator-Assistant / editor	1	0	0	3	0	0
Editor / reasoning	1	2	1	0	0	0
Planner / code	0	0	1	0	1	1
Inner-Editor-Assistant / find_all_refs	0	1	0	0	1	0
Navigator Interpreter / code	1	0	0	1	0	0
Inner-Navigator-Assistant / find_all_refs	0	0	1	0	0	0
Inner-Editor-Assistant / code_search	0	0	0	0	1	0
Inner-Editor-Assistant / open_or_create	0	0	0	0	1	0
Editor / exploration	0	0	1	0	0	0
Executor / reasoning	0	0	1	0	0	0

Table 5: Complete windowing participation matrix for the `requests-1766` taskset under relative static windowing (k=3). All 59 observed (agent, action) pairs are listed; cells are total occurrence counts across the 49 failed and 47 passed runs, bucketed into three proportional windows. The Failed columns sum to each windowed atom's e_f and the Passed columns to its e_p.

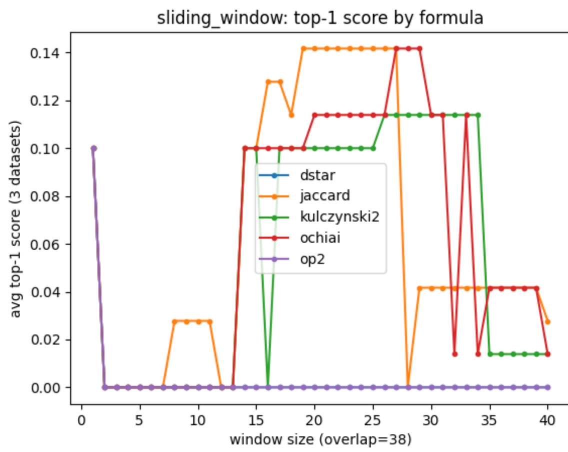
C Result Graphs



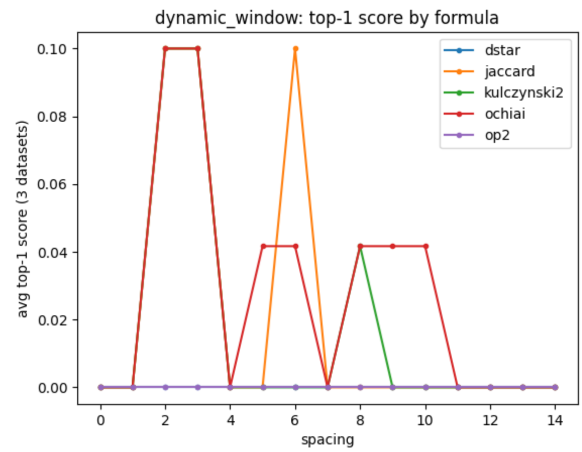
(a) Static window (absolute).



(b) Static window (relative).

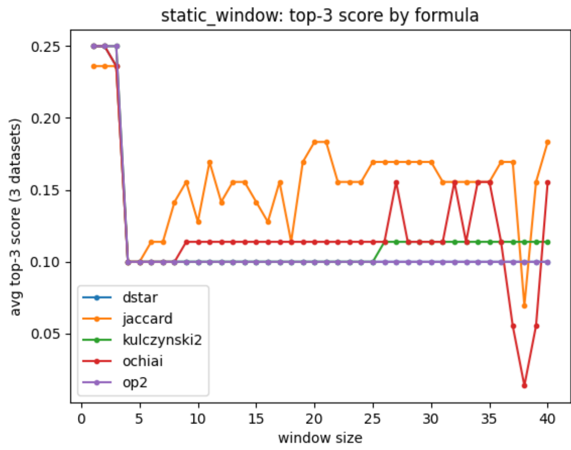


(c) Sliding window.

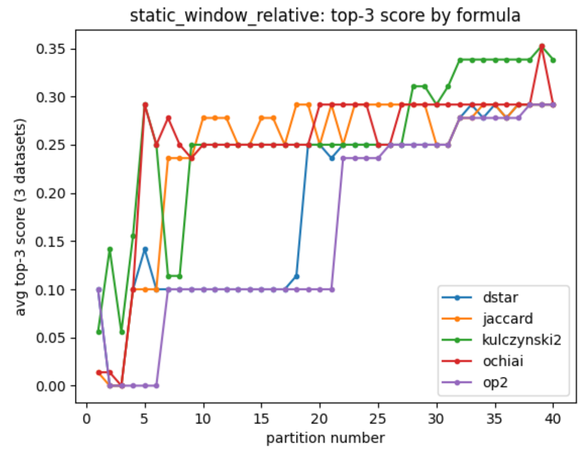


(d) Dynamic window.

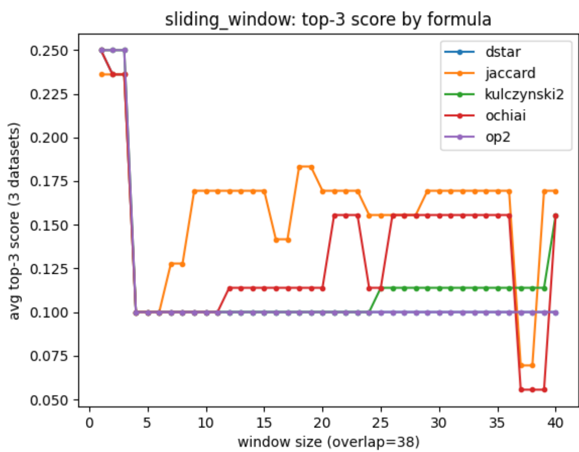
Figure 3: Top-1 accuracy by formula across the four window schemes.



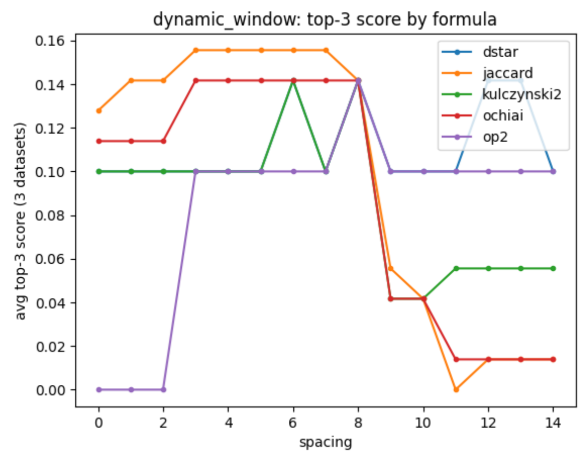
(a) Static window (absolute).



(b) Static window (relative).

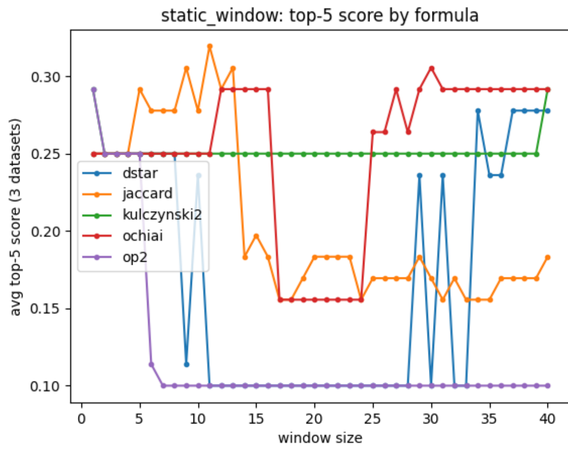


(c) Sliding window.

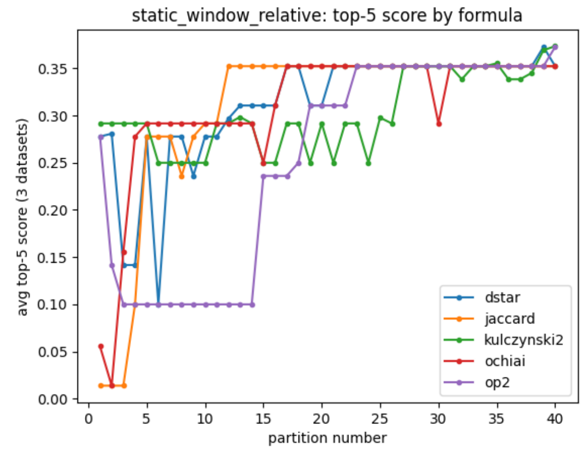


(d) Dynamic window.

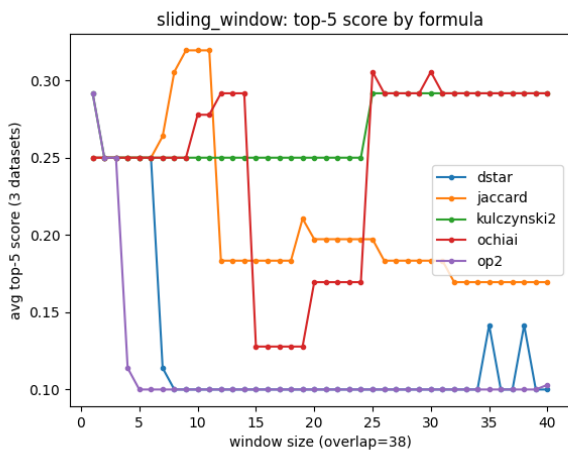
Figure 4: Top-3 accuracy by formula across the four window schemes.



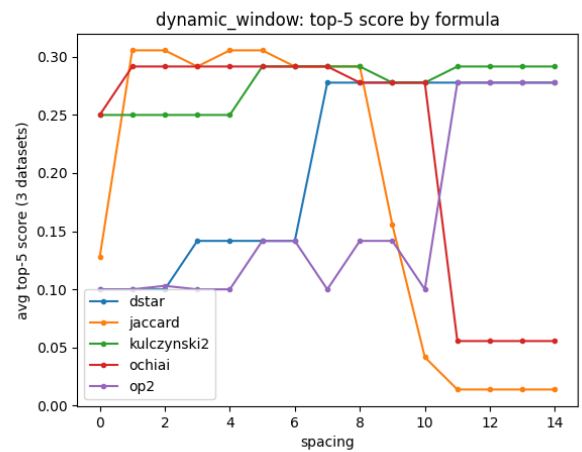
(a) Static window (absolute).



(b) Static window (relative).

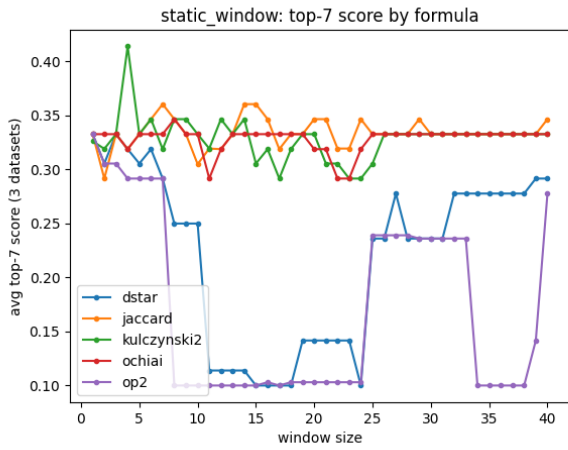


(c) Sliding window.

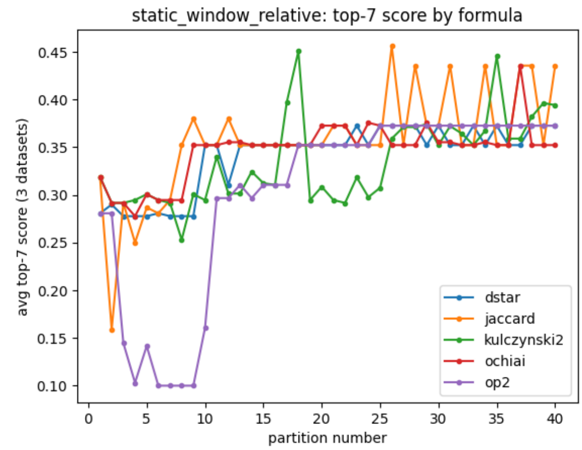


(d) Dynamic window.

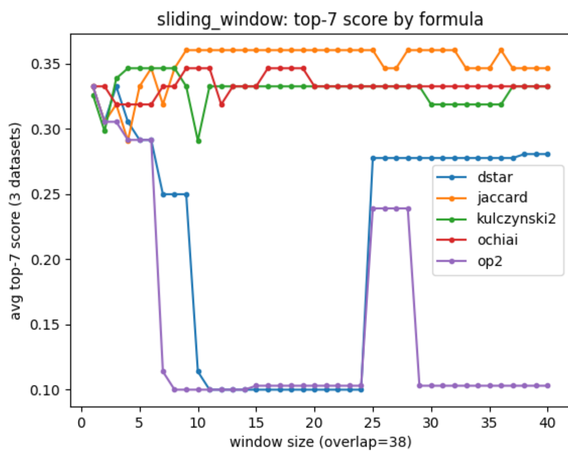
Figure 5: Top-5 accuracy by formula across the four window schemes.



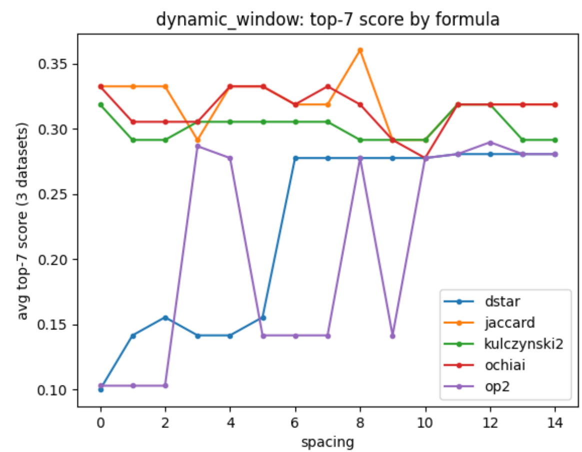
(a) Static window (absolute).



(b) Static window (relative).

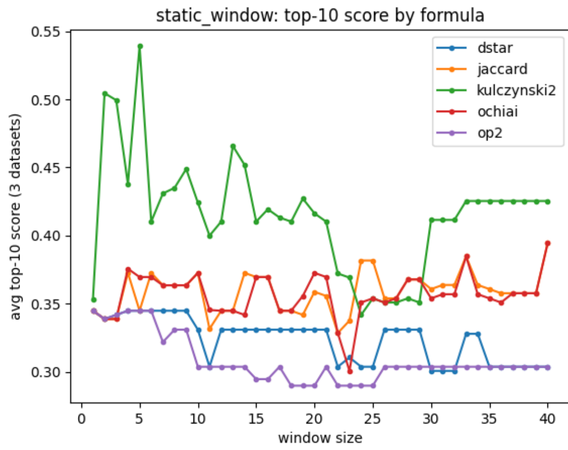


(c) Sliding window.

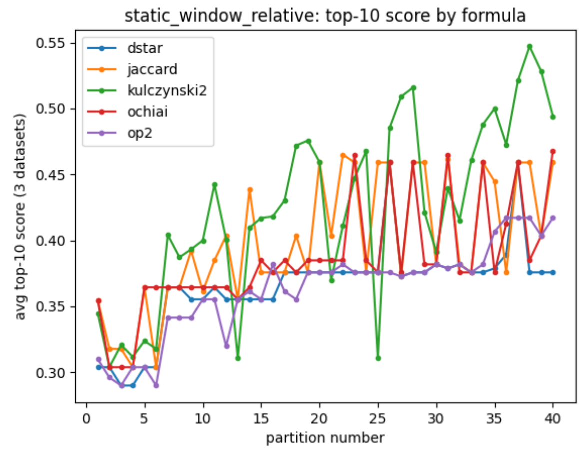


(d) Dynamic window.

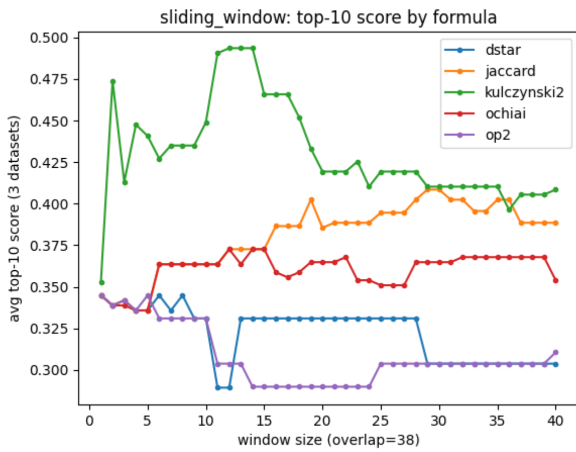
Figure 6: Top-7 accuracy by formula across the four window schemes.



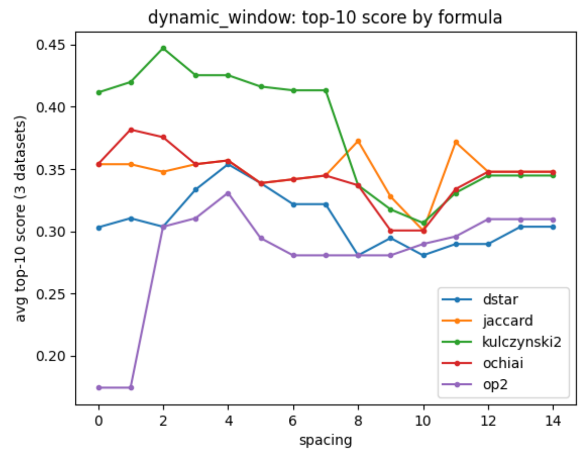
(a) Static window (absolute).



(b) Static window (relative).



(c) Sliding window.



(d) Dynamic window.

Figure 7: Top-10 accuracy by formula across the four window schemes.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, 2007.

[2] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent llm systems fail? 10 2025.

[3] Mengzhuo Chen, Junjie Wang, Fangwen Mu, Yawen Wang, Zhe Liu, Huanxiang Feng, and Qing Wang. Seeing the whole elephant: A benchmark for failure attribution in llm-based multi-agent systems. 4 2026.

[4] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified, 2024.

[5] Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. Trail: Trace reasoning and agentic issue localization, 6 2025.

[6] Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang, Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-one: A generalist multi-agent system for solving complex tasks. 11 2024.

[7] Yu Ge, Linna Xie, Zhong Li, Yu Pei, and Tian Zhang. Who is introducing the failure? automatically attributing failures of multi-agent systems via spectrum analysis, 9 2025.

[8] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[9] James Jones and Mary Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. pages 273–282, 11 2005.

[10] 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. 9 2025.

[11] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development. 6 2024.

[12] Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of llm multi-agent systems, 6 2025.