Tool-Calling LLM Agents Under Realistic MCP Faults: Scale Does Not Purchase Fault Tolerance

Antonio Ken Iannillo[*]
SnT, University of Luxembourg
antonioken.iannillo@uni.lu

Joshua Segun Owotogbe
Tilburg University

Roberto Natella
Gran Sasso Science Institute

Francesco Avallone
University of Naples Federico II

Kristina Kudryavtseva
SnT, University of Luxembourg

Indika Kumara
Tilburg University

Abstract

LLM agents increasingly rely on external tools exposed via the Model Context Protocol (MCP), yet current benchmarks assume tools respond correctly and on time, leaving agent resilience to realistic tool failures untested. We introduce MCP-Injector, a drop-in, protocol-aware fault-injection middleware that interposes on the MCP tool-call path with controlled persistence (permanent, transient, intermittent) and seedable schedules. Its fault model is calibrated to the observed incident distribution (failure mode $\times$ persistence) of the public MCP ecosystem, mined from 57,240 issue and pull-request threads across 1,774 open-source MCP repositories. In a 1,350-cell experiment crossing two backbone sizes (20B, 120B), three agent stacks (custom multi-round plan-and-execute, OpenAI Agents SDK, LangChain), and three fault injection conditions, we find that faults drawn from this permanent-heavy stress profile cause severe quality degradation (task-completion score falls by 60%, from 8.08 to 3.23 out of 10), and that scale does not purchase fault tolerance for the two gpt-oss backbones we test: the 120B retains no larger share of its fault-free score under faults (resilience-ratio model, $p = 0.96$), and its completion rate under faults is, if anything, lower (91.1% vs. 95.6%; difference 4.4 points, 95% CI $[-0.2, 9.3]$). Agent stacks differ in how well they absorb faults, which suggests that framework-level error handling matters more than model scale. These results demonstrate that capability benchmarks alone can mask operational brittleness, motivating protocol-aware fault injection as a complementary evaluation dimension for tool-calling agents.

1 Introduction

LLM agents increasingly act through external tools [8, 18] exposed via the Model Context Protocol (MCP), a vendor-neutral, structured client-server interface for tool invocation [6, 26, 7]. This makes agent quality a *system* question: end-to-end behavior depends on the MCP channel, server implementations, schemas, and the operational health of the tool ecosystem. In production, MCP-facing agents inherit failure modes familiar from distributed services [1, 3], yet current benchmarks evaluate agents assuming tools respond correctly and on time [31, 27, 28, 19]. Without protocol-aware fault injection, the community lacks a controlled method to answer: *Which agent designs degrade gracefully? Does scaling the backbone reduce or increase operational brittleness?*

Because MCP standardizes *structured* tool calls, failures can be semantic: a well-formed but wrong value, a schema violation, a silent hang, or a retry cascade. These behaviors align with classic content and timing failure classes [1] but must be expressed at the MCP interface to exercise agent logic that

[*]Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

interprets outputs, retries, and composes answers. Infrastructure-level chaos engineering [2] operates below this semantic layer, and benchmarks do not control fault persistence or provide calibrated profiles, leaving a gap for *benchmark-compatible, MCP-semantic* fault injection. We introduce MCP-Injector, a drop-in middleware that interposes on the MCP client-server path and injects faults with controlled persistence (permanent, transient, intermittent), with a fault model calibrated to the observed incident distribution of the public MCP ecosystem (57,240 issue and pull-request threads across 1,774 open-source MCP repositories) and packaged as a seedable injection schedule applicable to any MCP-based agent without modifying the agent or the tools.

Contributions. (1) An MCP fault model calibrated to the observed incident distribution (mode $\times$ persistence) of the public MCP ecosystem, mined from 57,240 threads across 1,774 open-source repositories, so that the mix of injected faults follows reported incidents rather than hand-crafted scenarios. (2) A 1,350-cell full-factorial study (three stacks, two backbone scales, three fault conditions, 25 tasks, three seeds) showing that scale does not purchase fault tolerance for the two gpt-oss backbones tested (the larger backbone retains no larger share of its fault-free score under calibrated faults and completes descriptively fewer tasks) and suggestive evidence that framework-level error handling matters more than model scale. (3) MCP-Injector itself, a reproducible, stack-agnostic, drop-in middleware released with full artifacts.

2 Background and Definitions

Tool-using LLM agents are implemented as an *MCP client* embedded in an agent runtime, connected to one or more *MCP servers* that expose executable *tools*. We treat the **tool-call exchange** (the structured invocation request and its result or error) as the key service interface and call it the **MCP tool-call boundary**. Our experiments and injector operate strictly at this boundary; other MCP capabilities (prompts, resources, streaming) are out of scope, since tool calls are the dominant actuator by which agents affect the external world. We adopt a **non-adversarial threat model**: injected faults represent commonplace production problems (misconfiguration, transient unavailability, schema drift, malformed outputs), not an intelligent attacker; security attacks, model-internal faults, and unrelated application-layer issues are excluded.

Dependability terminology. We use standard dependability terminology [1] instantiated at the MCP tool-call boundary. A **service failure** is a deviation of observed tool-call behavior from what the client expects (e.g., an error instead of a result, excessive latency, a schema-violating result); a **fault** is its underlying cause (a buggy implementation, misconfiguration, or dependency outage). Because systems compose, a failure of the MCP server is an *external fault* from the client's perspective. Our injector injects such external faults at the *interface* between the agent and the MCP server, rather than within server internals, for more controlled and efficient experiments as typical of fault injection studies [20].

Failure taxonomy and persistence. At the MCP boundary, interface failures divide into two classes [1]: **timing failures** (results not delivered within the expected time budget, excessively delayed, or never delivered) and **content failures** (results that arrive but are wrong or unusable: incorrect values, malformed payloads, schema/type violations). Beyond *what* fails, dependability evaluation must model *how* failures evolve. We treat **persistence** as the central temporal dimension: **permanent** failures recur on every invocation until external corrective action; **transient** failures are short-lived and resolve without intervention; **intermittent** failures alternate between faulty and correct behavior in temporally correlated bursts. Our injector operationalizes these persistence modes as injection schedules over the tool-call boundary.

3 MCP-Injector

MCP-Injector[2] is a protocol-aware fault-injection middleware for tool-calling LLM agents that communicate over MCP. It is *drop-in* (no modifications to the agent runtime, MCP client, or servers), *reproducible* (seeded injection schedules), *low-interference* (minimal perturbation when faults are disabled), and *stack-agnostic*.

[2]Artifact (code, data, analysis pipeline): `https://gitlab.com/uniluxembourg/snt/sedan/papers/mcp-injector`, archived at `https://doi.org/10.5281/zenodo.23573765`.

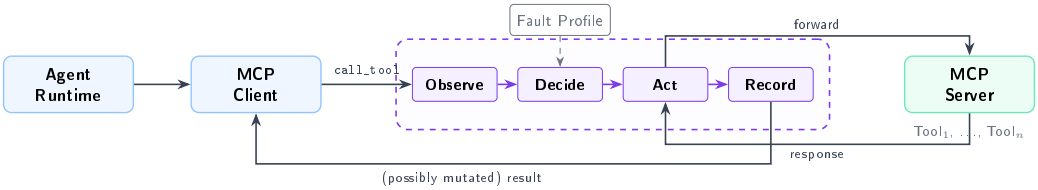


Figure 1: MCP-Injector architecture. The middleware transparently interposes on the MCP tool-call path. Each invocation passes through four stages: **Observe** (read tool name, arguments, and context), **Decide** (select an operator based on the fault profile, persistence state, and seeded randomness), **Act** (forward the call to the real server and optionally mutate the response), and **Record** (emit a structured log entry for post-hoc analysis).

Table 1: Fault operators implemented by MCP-Injector. Content operators act on the server's response; timing operators act before or instead of forwarding.

ID	Operator	Class	Effect
OP0	Null injector	n/a	Passthrough (no mutation)
OP1	Minor mistake	Content	Random character replacement
OP2	Incorrect output	Content	Word-level substitution
OP3	Invalid output	Content	Type coercion (e.g., str→int)
OP4	Explicit error	Content	Returns MCP error message
OP5	Delayed response	Timing	Artificial sleep before return
OP6	Silent drop	Timing	Indefinite hang (no response)

3.1 Architecture

MCP-Injector is implemented as proxy-style middleware on the tool-invocation path (Figure 1). It intercepts the `call_tool` hook exposed by MCP client libraries, acting as an MCP server toward the upstream agent and as an MCP client toward the downstream real server(s), and can be toggled on or off via a single configuration flag. Request-side interception handles timing faults (delays, silent drops) and explicit errors that can be synthesized without contacting the server; response-side interception handles content faults by mutating the real server's output before it reaches the agent.

3.2 Fault API

MCP-Injector exposes faults through a configuration-driven API that separates three orthogonal concerns: *what* failure to emulate (operator), *how long* it persists (schedule), and *where* it applies (scope).

Operators. Table 1 lists the six fault operators plus one passthrough, each corresponding to an empirically observed MCP failure mode (Section 4). Operators either short-circuit the request (explicit error, silent drop), delay forwarding, or mutate the response content.

Persistence schedules. MCP-Injector realizes the three persistence regimes of Section 2 as seeded schedules: **Permanent** fires on every subsequent matching call once activated, **Transient** fires on a single call with a configured activation probability and then resets, and **Intermittent** holds the drawn entry for a random number of calls (released with a per-call deactivation probability) and fires on each held call with an activation probability.

A fault profile may combine multiple (operator, persistence) entries with configured selection weights, which MCP-Injector samples to produce realistic failure mixtures under seeded fault schedules: injection decisions are drawn from a seeded pseudo-random stream, so they are deterministic given the seed and the agent's tool-call history, and every fired fault is logged. End-to-end runs are not bit-reproducible, because local serving and the judge are stochastic (Appendix A).

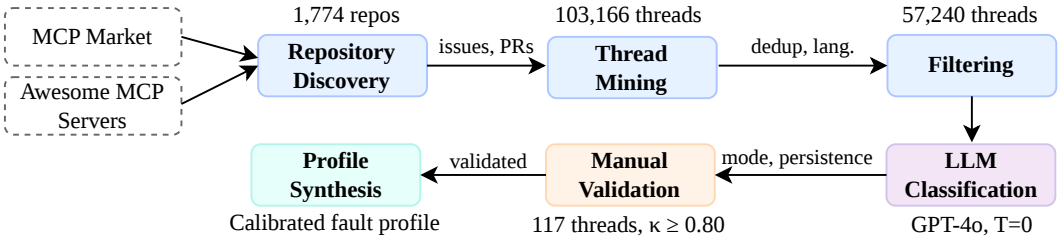


Figure 2: Fault-profile calibration pipeline: from repository discovery to a seedable injection profile. Numbers indicate corpus size after each stage.

4 Fault Profile Methodology

We derive a fault profile, calibrated to the observed incident distribution of the public MCP ecosystem, that specifies *what* MCP-visible failure manifestations to inject and *how* they evolve over time. Rather than relying on ad hoc assumptions, we mine public MCP tool-server repositories and classify observed incidents using standard dependability terminology [1], specialized to the MCP tool-call boundary (Section 2). Figure 2 summarizes the six-stage pipeline.

Fault taxonomy. Within the content and timing classes of Section 2, we distinguish six mutually exclusive **failure modes**. Content failures: *minor mistake* (small deviation, structurally valid), *incorrect output* (semantically wrong, structurally valid), *invalid output* (schema/type violation), and *explicit error* (structured error in place of a result). Timing failures: *delayed response* and *silent drop*. Together with the three persistence regimes of Section 2, these modes define the (mode, persistence) target space for calibration.

Repository discovery and thread mining. We assembled **1,774** active public MCP tool-server repositories (collected September 2025) by combining two public registries, MCP Market[3] and Awesome MCP Servers[4], deduplicating, and removing inaccessible ones. For each repository we mined up to 200 recent issues and pull requests via the GitHub REST API (103k issues; 64k PRs; 158k comments; 258k commits; 71k reviews; 65k review comments) and constructed per-thread narratives by concatenating all artifacts in chronological order, truncating threads above 50k characters to prioritize early failure descriptions. This stage yielded **103,166** consolidated threads.

Filtering. A keyword-based heuristic partitioned threads into failures, non-failures, and uncertain cases following prior issue-mining work [24]. After removing 1,374 cross-link duplicates, excluding build-related threads (3,013) and non-English threads (2,668) across automated and manual passes, the analysis set contained **57,240** English-language classified threads.

LLM-based classification and manual validation. We used gpt-4o-2024-08-06 (temperature 0, fixed JSON schema, ∼$800 in API cost) to classify each thread into (failure mode, persistence) pairs and extract mode-specific parameters (error subtypes, data-type patterns, delay magnitudes). To assess reliability, two senior researchers independently annotated a stratified sample of **117** threads (five per non-empty stratum) with a third adjudicating disagreements; agreement was high (Cohen's $\kappa = 0.93$ for failure mode, 0.80 for persistence [14]).

Profile synthesis. Aggregating classifier outputs yields the empirical distribution in Table 2. Permanent failures predominate, with EXPLICIT_ERROR the largest share (10,393 of 14,829 classified failures); other content failures (INVALID_OUTPUT, INCORRECT_OUTPUT, MINOR_MISTAKE) are less frequent but operationally significant due to silent propagation risk; timing failures (DELAYED_RESPONSE, SILENT_DROP) span permanent and intermittent persistence. We package these distributions as a versioned, seedable fault profile consumed by MCP-Injector (Section 3).

[3] https://mcpmarket.com/server
[4] https://github.com/punkpeye/awesome-mcp-servers

Table 2: Joint distribution of failure modes and persistence categories across 57,240 mined threads. NON_FAILURE and UNCERTAIN persistence entries are classified but excluded from the injection profile.

Failure Mode	Interm.	Perm.	Trans.	Uncert.	Total
DELAYED_RESPONSE	51	157	1	24	233
EXPLICIT_ERROR	669	9,075	97	552	10,393
INCORRECT_OUTPUT	155	1,733	1	50	1,939
INVALID_OUTPUT	47	511	1	25	584
MINOR_MISTAKE	15	177	0	10	202
SILENT_DROP	167	963	4	83	1,217
UNCERTAIN	28	0	0	233	261
NON_FAILURE	0	0	0	42,411	42,411
Total	1,132	12,616	104	43,388	57,240

What is calibrated. The profile matches the relative frequency of (mode, persistence) cells among mined failure reports, not operational failure rates. Data-derived: the 18 selection weights (count/N, $N = 13{,}824$) and the six INTERMITTENT activation probabilities (Equation (2)). Hand-set: $d = 0.3$, the 30 s delay, the OP1/OP2 perturbation intensities, and TRANSIENT activation 1 (Appendix A). The profile is permanent-heavy: 91.3% permanent, 8.0% intermittent, 0.75% transient; EXPLICIT_ERROR/PERMANENT alone carries 65.6%, all explicit errors 71.2%, silent content modes 19.1%, timing modes 9.7%. Because GitHub issues over-represent persistent, reproducible failures, we treat it as a stress profile, although its shape is in line with the 61.4% Bohrbug share among 520 software faults from 18 JPL/NASA missions [9].

5 Experimental Protocol

5.1 Benchmark and Task Set

We evaluate on MCP-Bench [28], a multi-step tool-use benchmark with rubric-based LLM-as-a-judge scoring. Using its task-generation pipeline, we produced **25 single-server tasks** from 25 successfully connected MCP servers (out of 28 attempted), each exposing multiple tools. Tasks are presented to agents as *fuzzy* (naturalistic, underspecified) user prompts; the original concrete descriptions serve only as evaluation references. Distraction servers are disabled (`--distraction-count 0`) to isolate tool-call faults from irrelevant conversational noise.

5.2 Experimental Factors

The experiment crosses four factors in a full-factorial design.

LLM backbones. Two locally hosted open-weight models, **gpt-oss-20B** and **gpt-oss-120B**, served at their released MXFP4 precision with a fixed 16,000-token context window and default decoding parameters on consumer Apple Silicon via LM Studio (full hardware and serving configuration in Appendix I). No per-model prompt tuning is applied; differences are attributable to model scale alone.

Agent stacks. Three orchestration frameworks connect to the same MCP tool interface: (i) a **multi-round plan-and-execute** loop from MCP-Bench [28]; (ii) an **OpenAI Agents SDK** agent [21]; (iii) a **LangChain/LangGraph** pre-built agent [15]. Task prompts, tool schemas, context budget, and the per-task timeout/stopping policy are held constant across stacks; only intrinsic orchestration behavior (planning, retry logic, intermediate reasoning format) is allowed to vary. Appendix F documents these differences in detail.

Fault-injection conditions and repetitions. Each task runs under three proxy conditions: (i) **Proxy-off**: no interposition (true baseline); (ii) **OP0**: proxy with a null pass-through operator (exercises integration overhead); (iii) **OPP**: proxy with the 18-rule calibrated fault profile from Section 4 (derivation in Appendix A). Robustness comparisons use a *paired* within-task design: for every (task, seed) tuple we compute per-metric degradation $\Delta = \mathrm{OPP} - \mathrm{OP0}$. Three seeds per condition (fixing

benchmark, injector, and framework RNG seeds) yield $2 \times 3 \times 3 \times 3 \times 25 = \mathbf{1{,}350}$ experimental cells.

5.3 Execution and Run Policy

All 1,350 reported cells are governed by a single run policy: a 3,000 s per-task wall-clock timeout and a loop bound of 40 rounds (MCPBench `max_execution_rounds`), 40 turns (OpenAI `max_turns`), or 25 supersteps (LangChain, LangGraph default; Appendix F), which prevents unbounded hangs under silent-drop and delay faults. All OPP cells and every reported non-completion ran under these limits; fault-free cells that completed in earlier campaign rounds ran under tighter limits that they never reached (Appendix I), so the campaign history cannot inflate any reported non-completion. A run terminated by the timeout, by loop-bound exhaustion, or by an unhandled tool exception is classified as a *system-level failure*. The LLM-as-a-judge evaluator (gpt-4o-mini-2024-07-18 model) scores each completed run five times with a randomized rubric order to mitigate judge instability; we report median scores. Aggregate compute for the final 1,350 reported cells was approximately 58 wall-clock hours of agent execution plus about 18 hours of LLM-as-a-judge API time; the full five-round campaign history is in Appendix I.

5.4 Metrics

We report three metric families. **(i) Task-quality scores**: MCP-Bench's rubric-based judge rates task fulfillment, grounding, tool appropriateness, and parameter accuracy, enabling analysis beyond binary task completion. **(ii) System-level outcomes**: timeout and unhandled-error indicators, reported as failure rates per condition. **(iii) Latency decomposition**: end-to-end, agent-internal (LLM, planning, orchestration), and evaluation time, supporting both intrusiveness analysis under OP0 and fault-amplification analysis under OPP.

6 Results

We report three primary findings: calibrated MCP faults cause severe, pervasive quality degradation (Section 6.1); framework-level error handling plausibly matters more than model scale (suggestive evidence; Section 6.2); and scale does not purchase fault tolerance (Section 6.3). Two apparatus-validation checks precondition these findings: proxy non-intrusiveness (paired Wilcoxon, all Bonferroni-corrected $p \geq 0.05$ over 18 tests; Appendix B) and injection fidelity (the per-run fault draw follows the profile, and cells exposed to at least one fault degrade far more than unexposed cells; Appendix C). Mechanistic and task-level complements appear in Appendices D, G, and H.

6.1 Calibrated Faults Cause Severe, Pervasive Quality Degradation

We analyze 450 OPP cells (25 tasks $\times$ 3 seeds $\times$ 3 stacks $\times$ 2 models) paired against their OP0 counterparts. Cells that timed out or errored under OPP receive 0 in the primary analysis; a completed-only secondary analysis is reported alongside.

Completion rates. Under OP0, completion rates sit between 95 and 100% across all (stack, model) combinations (proxy-off baselines 93 to 100%). Under OPP, they fall to 88 to 97%, with the largest declines in OpenAI/120b (88%) and MCPBench/120b (91%). The 120b completes fewer tasks than the 20b in every stack (pooled 91.1% vs. 95.6%), a descriptive gap examined in Section 6.3; full per-cell rates appear in Appendix D (Table 4).

Score degradation. Pooled across all stacks and models, degradation is severe: task completion score drops 60% (8.08 to 3.23), tool selection 59%, planning effectiveness 67%, and the remaining six metrics 54 to 68% (full breakdown in Appendix D, Table 5). Every metric drops by 4.4 to 5.2 points out of 10 (paired 95% CIs within ± 0.4 points; rank-biserial $r \approx 0.9$; Benjamini-Hochberg corrected Wilcoxon $p < 0.001$). Restricted to cells that completed under both conditions, the pattern persists (57% drop in task completion), so degradation reflects genuine quality loss among successful runs and not only non-completion.

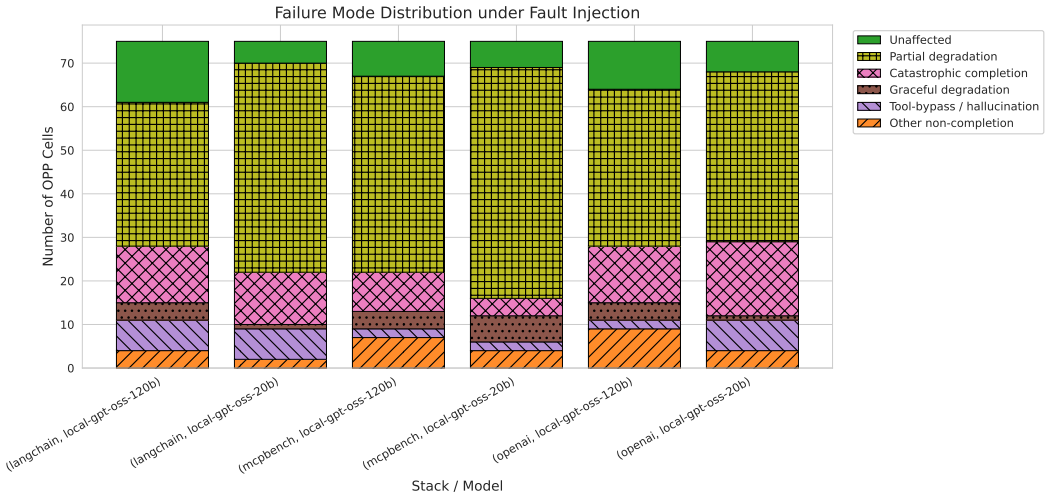


Figure 3: Failure mode distribution across all 450 OPP cells, stratified by stack and model. Partial degradation is the most common outcome (44 to 71% of cells); non-completion, catastrophic completion, graceful degradation, and tool bypass together account for 21 to 39%.

Failure mode taxonomy. Figure 3 classifies all 450 OPP cells into six behavioral categories. *Partial degradation* is the most common outcome (56% of cells; 44 to 71% across stack and model): agents typically complete tasks but with substantially reduced quality. *Unaffected* cells account for 7 to 19% per stack and model. The remaining 21 to 39% split across catastrophic completion, graceful degradation, tool bypass, and non-completion. Annotated case studies for each category appear in Appendix E.

Tool bypass is a baseline behavior that faults expose. A completed run counts as tool bypass when the judge rates task fulfillment at least 4 but tool appropriateness below 2: a plausible answer without tool evidence. Under OPP this covers 27 cells (6.0%): 14 of 150 for LangChain, 9 for OpenAI, and 4 for MCPBench. Faults are not the main cause. The same rule flags 21 OP0 and 16 Proxy-off cells, the LangChain and MCPBench counts are identical with and without faults, and only OpenAI rises (3 to 9 cells; paired exact McNemar $p = 0.07$). Bypass also explains much of the zero-dose set: 17 of the 27 bypass cells made no tool call, so no fault could fire, and they are 17 of the 55 zero-dose OPP cells (12.2%); excluding zero-dose cells raises the task-completion drop from 60% to 67%. These runs score well despite ungrounded answers. A LangChain/120b run of `unit_converter_000` computed every conversion from memory and scored 9.8 on task fulfillment and 9.3 on task completion, but 1.2 on tool appropriateness. Such answers inflate scores under faults while hiding what a deployment would care about, so we also report a *grounded-fulfillment* view that counts task fulfillment only when tool appropriateness is at least 2. Under OPP, runs below that threshold supply 16% of all fulfillment credit (OP0: 5%), so grounded fulfillment falls by 63% rather than 58% (paired drop -4.87 points, 95% CI $[-5.27, -4.47]$; OpenAI: 68% vs. 61%; Table 5).

6.2 Agent Stacks Shape Fault Response

The three agent stacks differ in how they surface tool errors to the model, producing measurably different resilience profiles. Figure 4 summarizes the key findings.

Completion and resilience. LangChain achieves the highest OPP completion rate (95 to 97%), OpenAI the lowest (88 to 95%), and MCPBench is in between (91 to 95%) (Figure 4a). Per-task score ratios ($\text{score}_{\text{OPP}}/\text{score}_{\text{OP0}}$) preserve the same ordering: LangChain has the highest median, MCPBench and OpenAI are lower with wider spread, and the ordering is stable whether failed runs are scored as zero or excluded (Figure 4b). A linear mixed-effects model with stack, condition, and their interaction as fixed effects and task as random intercept yields one significant and one marginal interaction: against a common OPP effect of -4.26 points, MCPBench loses a further 0.93 points relative to LangChain (95% CI $[-1.86, -0.01]$, $p = 0.048$) and OpenAI a further 0.84 ($[-1.77, 0.08]$,

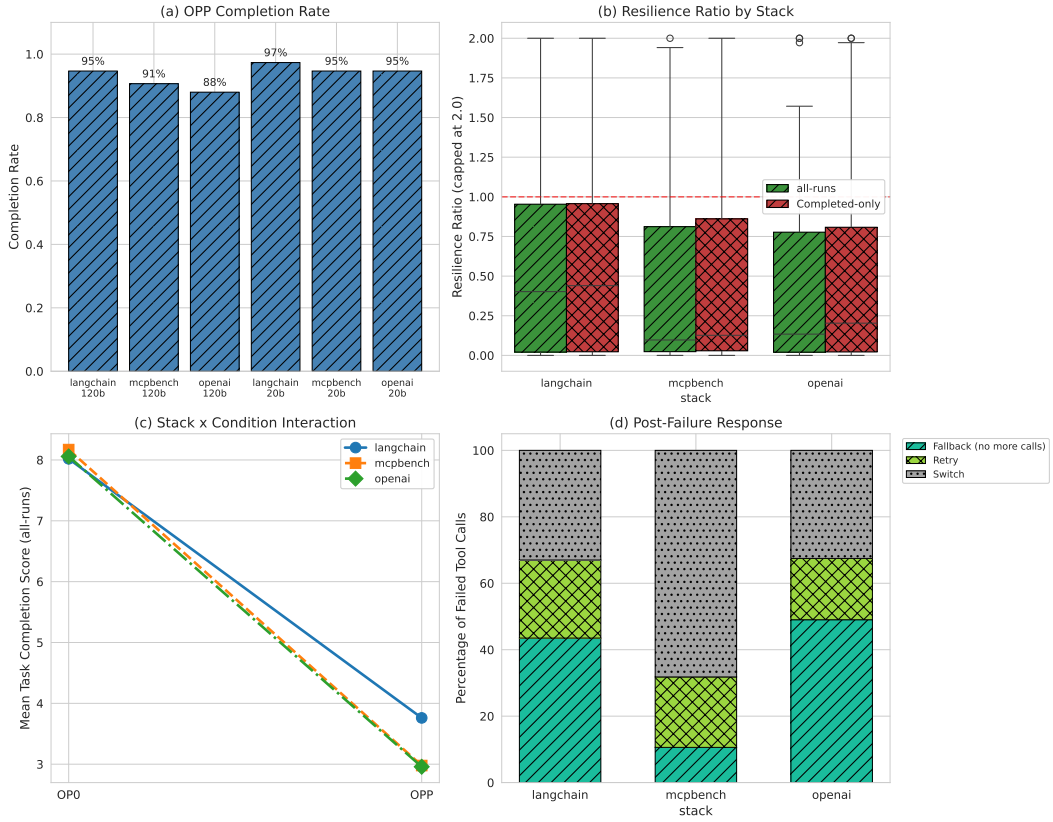


Figure 4: Stack sensitivity analysis. (a) OPP completion rates by stack and model. (b) Primary and completed-only resilience ratios by stack. (c) Stack×condition interaction: MCPBench and OpenAI degrade more steeply than LangChain. (d) Post-failure response strategy differs markedly across stacks.

$p = 0.074$) (Figure 4c). Both intervals reach close to zero, no multiplicity correction is applied, and MCPBench also receives more fault events per cell (Appendix C), so we treat the framework effect as suggestive rather than established.

Post-failure behavior. The three stacks respond qualitatively differently (Figure 4d, full analysis in Appendix G). MCPBench is *switch-dominant*: after a failure, its planner re-plans with a different tool on the next round, consistent with its plan-and-execute loop that re-issues the full `planned_tools` list each iteration. LangChain and OpenAI are *fallback-leaning*: both exit the tool loop more often than they retry, consistent with their ReAct-style loops terminating once the LLM emits a text-only message.

6.3 Scale Does Not Purchase Fault Tolerance

Does model scale confer fault resilience beyond raw capability? The resilience question asks whether the 120b *maintains a higher fraction of its own baseline* under OPP than the 20b does. On this benchmark, the two backbones have similar fault-free scores on completed runs (OP0: 8.37 vs. 8.13 task-completion score; Proxy-off: 8.11 vs. 8.30).

No detectable resilience advantage. In a mixed-effects model of the per-cell resilience ratio (OPP over OP0 task-completion score, non-completions scored zero; task random intercept), the model-size coefficient is indistinguishable from zero ($\beta_{\text{model}} = -0.009$, 95% CI $[-0.39, 0.37]$, $p = 0.96$), and neither model×stack interaction is significant ($p \geq 0.23$). The interval is wide: the data show no resilience benefit from scale for these two gpt-oss backbones under this serving setup, but cannot exclude a modest effect in either direction.

Decomposing the completion gap. Pooled OPP rates conflate three components. Fault-free quality is similar for the two backbones (above). Fault-free completion is already lower for the 120b (OP0: 96.9% [93.7, 98.5] vs. 99.1% [96.8, 99.8], Wilson 95% CIs; Proxy-off: 96.0% vs. 98.7%). The fault-attributable drop, paired over identical (stack, task, seed) cells, is -5.8 points for the 120b (95% CI $[-10.4, -1.4]$, exact McNemar $p = 0.015$) and -3.6 for the 20b ($[-7.2, -0.5]$, $p = 0.039$); the 2.2-point difference between them is not distinguishable from zero (95% CI $[-3.0, 7.5]$). The pooled OPP gap, 95.6% [92.0, 97.6] vs. 91.1% [86.7, 94.2] (difference 4.4 points, 95% CI $[-0.2, 9.3]$, Fisher exact $p = 0.088$), is therefore half baseline completion and half a larger but uncertain fault-attributable drop, and we report it as descriptive rather than as an effect of scale. The 120b is lower in every stack, most for OpenAI (88% vs. 95%) and MCPBench (91% vs. 95%), the two stacks where raw error messages reach the model most directly (Table 4). When the 120b does complete under OPP, its answers score higher (3.68 vs. 3.26), so capability and resilience rankings need not agree.

Candidate explanations. Greater fault exposure is consistent with the gap but does not establish it. Under OPP the 120b receives more fault injections per cell than the 20b in every stack (mean 4.6 vs. 3.9), it produces longer outputs under OP0 (2,063 vs. 1,711 output tokens per cell), and its retry loops on a failing tool are slightly longer (mean 4.5 vs. 3.9 consecutive retries among the 38 flagged runs; Appendix G). Its OPP non-completions (20 vs. 10) are wall-clock timeouts in all but two cases, not turn-budget exhaustion. The shared 16K-token context window and cross-stack differences in tool-call volume are co-equal alternative explanations (Section 7). Degradation is also strongly task-dependent: live APIs (game_trends, -97%) and deep sequential chains (national_parks, -82%) are far more vulnerable than short-chain computational tasks (unit_converter, -19%; Appendices D, G, H).

7 Limitations

Construct validity. The LLM judge is stochastic; five-fold scoring under randomized rubric order with median aggregation reduces but does not remove its noise, and the rubric rewards some ungrounded answers (Section 6.1). The fault profile inherits the reporting bias of GitHub issues, so we treat it as a stress profile (Section 4). Content operators omit schema drift, payload truncation, and partial result omission.

Internal validity. The proxy is non-intrusive (Appendix B), and the per-run fault draw follows the profile (Appendix C). Two explanations remain co-equal with fault exposure: the shared 16K-token context window, within which the 120b produces longer outputs, and tool-call volume (MCPBench cells receive 8.6 fault events on average against 2.3 for LangChain and 1.8 for OpenAI; 120b cells 4.6 against 3.9). Since stacks differ in both call volume and error handling, exposure and error handling remain partially entangled, and that longer tool-call chains amplify exposure is a hypothesis, not an established mechanism (Appendix H). Zero-dose cells (12.2%) and three seeds per cell limit power for per-task comparisons and rare interactions.

External validity. Both backbones come from one model family under one serving setup. MCP-Bench is easy without faults (OP0 completion 95 to 100%, mean task-completion score 8.08, per-task means 4.8 to 9.9), which limits what the results say about harder workloads. The study is powered for the ecosystem's 91%-permanent incident mix: TRANSIENT and INTERMITTENT entries produced 9 of 1,905 fired events, too few to characterize those regimes. The profile reflects the open-source MCP ecosystem as of September 2025; the stacks exclude multi-agent and human-in-the-loop workflows, and the 25 single-server tasks leave multi-server and longer-horizon settings to future benchmarks.

8 Broader impacts

MCP-Injector is a dual-use middleware: used as intended, it enables reproducible pre-deployment stress testing against a fault profile calibrated to the observed incident distribution of the public MCP ecosystem; as a transparent MCP interposition layer, its operator primitives (fabricated errors, delays, silent drops, perturbed payloads) could be repurposed adversarially by an attacker already positioned on the MCP path. Since the profile is shaped by the reporting bias of public GitHub artifacts (Section 4), it should be treated as a stress test rather than a certification instrument. We mitigate these risks by shipping OP0 as the default, tagging injected responses with a fault header

so cooperating loggers can distinguish synthetic from real faults, and scoping documentation to pre-deployment use in staging environments.

9 Related work

Tool-use benchmarks. Existing tool-use benchmarks [16, 30, 31, 27, 19, 28, 4, 17] measure success rates assuming tools respond correctly and on time; Xiong et al. [29] study parameter-filling failures *inside* the LLM rather than the external tool, and Agent-SafetyBench [33] and Ruan et al. [23] address adversarial threats rather than non-adversarial operational faults. ReliabilityBench [10] is closest to our work, evaluating fault tolerance under timeouts, rate limits, and schema drift; MCP-Injector differs in that its fault model is calibrated to the incident distribution mined from 1,774 MCP repositories, it models persistence schedules (permanent, transient, intermittent), and it operates as a transparent middleware requiring no agent modification.

Fault injection and chaos engineering. Classical fault injection evaluates system dependability by deliberately introducing faults at hardware, OS, or API boundaries [20, 5]. Chaos engineering [2] extends this philosophy to production distributed systems, and Yu et al. [32] survey its application to microservice architectures. Recent work applies chaos engineering principles to LLM-based systems: Owotogbe [22] proposes a framework for injecting agent communication failures in multi-agent systems, and Kikuta et al. [13] use LLMs to automate the chaos engineering cycle on Kubernetes. These efforts target infrastructure-level or inter-agent faults. MCP-Injector operates at a different layer, the MCP tool-call interface, injecting *semantic* faults (content errors, timing failures, schema violations) that exercise agent-level interpretation and recovery logic.

MCP ecosystem reliability. Hou et al. [11] and Huang et al. [12] catalogue MCP security risks and tool-invocation failure categories post hoc, and self-correction approaches such as Reflexion [25] improve robustness but assume tools are reliable; MCP-Injector instead provides a controlled, reproducible mechanism to *induce* realistic MCP failures and measure their impact on agent performance.

10 Conclusion

We presented MCP-Injector, a protocol-aware fault-injection middleware for MCP with a fault model calibrated to the observed incident distribution of the public MCP ecosystem (57,240 threads across 1,774 open-source repositories). Our 1,350-cell evaluation yields three findings. First, under this permanent-heavy stress profile, MCP faults cause severe quality degradation (60% drop in task-completion score; paired difference -4.85 points, 95% CI $[-5.22, -4.48]$), with *partial* degradation, not catastrophic failure, the most common outcome (56% of cells). Second, framework-level error handling appears to matter more than model scale (one significant and one marginal stack$\times$condition interaction, $p = 0.048$ and 0.074): stacks differ in whether faults cascade or are absorbed, although this evidence is suggestive. Third, for the two gpt-oss backbones tested, scale does not purchase fault tolerance: the 120B retains no larger share of its fault-free score under faults and completes descriptively fewer tasks, with greater fault exposure a candidate explanation rather than an established mechanism. Open directions: multi-server orchestration, longer-horizon tasks, and mitigations such as fault-aware planning.

Acknowledgments and Disclosure of Funding

We thank SURF for providing computational resources, under allocation EINF-16024, that supported part of the experimental work in this study. The authors declare no competing interests.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1 (1):11–33, 2004. doi: 10.1109/TDSC.2004.2.

[2] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016. doi: 10.1109/MS.2016.60.

[3] Jessica Castro, Nuno Laranjeiro, and Marco Vieira. Techniques and tools for runtime security monitoring and analysis of microservices. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, pages 191–193, 2023. doi: 10.1109/DSN-S58398.2023.00052.

[4] Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, et al. ACEBench: A comprehensive evaluation of LLM tool usage. In *Findings of the Conference on Empirical Methods in Natural Language Processing*, 2025.

[5] Domenico Cotroneo, Antonio Ken Iannillo, Roberto Natella, and Stefano Rosiello. Dependability Assessment of the Android OS through Fault Injection. *IEEE Transactions on Reliability*, 70(1):346–361, March 2021. ISSN 0018-9529, 1558-1721. doi: 10.1109/TR.2019.2954384. URL `http://arxiv.org/abs/1912.03490`. arXiv:1912.03490 [cs].

[6] Abul Ehtesham, Aditi Singh, and Saket Kumar. Enhancing clinical decision support and ehr insights through llms and the model context protocol: An open-source mcp-fhir framework. In *2025 IEEE World AI IoT Congress (AIIoT)*, pages 0205–0211, 2025. doi: 10.1109/AIIoT65859.2025.11105280.

[7] Qi Feng, Guowei Zhang, and Kaize Chen. Design and implementation of a delivery robot system for natural language interaction. In *2025 7th International Conference on Internet of Things, Automation and Artificial Intelligence (IoTAAI)*, pages 739–744, 2025. doi: 10.1109/IoTAAI66837.2025.11213046.

[8] In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. Pie: A programmable serving system for emerging llm applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 415–430, 2025.

[9] Michael Grottke, Allen P. Nikora, and Kishor S. Trivedi. An empirical investigation of fault types in space mission system software. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 447–456, 2010. doi: 10.1109/DSN.2010.5544284.

[10] Aayush Gupta. ReliabilityBench: Evaluating LLM agent reliability under production-like stress conditions. *arXiv preprint arXiv:2601.06112*, 2026.

[11] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model context protocol (MCP): Landscape, security threats, and future research directions. *ACM Transactions on Software Engineering and Methodology*, 2025.

[12] Donghao Huang, Gauri Malwe, and Zhaoxia Wang. When agents fail to act: A diagnostic framework for tool invocation reliability in multi-agent LLM systems. *arXiv preprint arXiv:2601.16280*, 2026.

[13] Daisuke Kikuta, Hiroki Ikeuchi, and Kengo Tajiri. LLM-powered fully automated chaos engineering: Towards enabling anyone to build resilient software systems at low cost. In *International Conference on Automated Software Engineering*, 2025.

[14] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[15] LangChain. Langgraph: Build resilient language agents as graphs. `https://github.com/langchain-ai/langgraph`, 2024. Accessed: 2026-04-13.

[16] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs, October 2023. URL `http://arxiv.org/abs/2304.08244`. arXiv:2304.08244 [cs].

[17] Wenrui Liu, Zixiang Liu, Elsie Dai, Wenhan Yu, Lei Yu, and Tong Yang. MCPAgentBench: A real-world task benchmark for evaluating LLM agent MCP tool use, 2025.

[18] Qing Luo, Weizhen Zeng, Manni Chen, Gang Peng, Xiaofeng Yuan, and Qian Yin. Self-attention and transformers: Driving the evolution of large language models. In *2023 IEEE 6th International conference on electronic information and communication technology (ICEICT)*, pages 401–405. IEEE, 2023.

[19] Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for General AI Assistants, November 2023. URL `http://arxiv.org/abs/2311.12983`. arXiv:2311.12983 [cs].

[20] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)*, 48(3):1–55, 2016.

[21] OpenAI. Openai agents sdk. `https://github.com/openai/openai-agents-python`, 2025. Accessed: 2026-04-13.

[22] Joshua Owotogbe. Assessing and enhancing the robustness of LLM-based multi-agent systems through chaos engineering. In *IEEE/ACM International Conference on AI Engineering (CAIN)*, 2025.

[23] Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. Identifying the Risks of LM Agents with an LM-Emulated Sandbox, May 2024. URL `http://arxiv.org/abs/2309.15817`. arXiv:2309.15817 [cs].

[24] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, pages 724–734, 2014.

[25] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

[26] Luis Miguel Vieira Da Silva, Aljosha Köcher, and Felix Gehlhoff. Beyond formal semantics for capabilities and skills: Model context protocol in manufacturing. In *2025 IEEE 30th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, 2025. doi: 10.1109/ETFA65518.2025.11205601.

[27] Jize Wang, Ma Zerun, Yining Li, Songyang Zhang, Cailian Chen, Kai Chen, and Xinyi Le. Gta: a benchmark for general tool agents. *Advances in Neural Information Processing Systems*, 37: 75749–75790, 2024.

[28] Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and Eugene Siow. MCP-Bench: Benchmarking Tool-Using LLM Agents with Complex Real-World Tasks via MCP Servers, August 2025. URL `http://arxiv.org/abs/2508.20453`. arXiv:2508.20453 [cs].

[29] Qian Xiong, Yuekai Huang, Ziyou Jiang, Zhiyuan Chang, Yujia Zheng, Tianhao Li, and Mingyang Li. Butterfly effects in toolchains: A comprehensive analysis of failed parameter filling in LLM tool-agent systems. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2025.

[30] Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. On the Tool Manipulation Capability of Open-source Large Language Models, May 2023. URL `http://arxiv.org/abs/2305.16504`. arXiv:2305.16504 [cs].

[31] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

[32] Senyao Yu, Huayao Wu, Xintao Niu, and Changhai Nie. A systematic literature review on fault injection testing of microservice systems. *IEEE Transactions on Services Computing*, 2025.

[33] Hanrong Zhang, Jingyuan Huang, Kai Mei, Yifei Yao, Zhenting Wang, Chenlu Zhan, Hongwei Wang, and Yongfeng Zhang. Agent security bench (ASB): Formalizing and benchmarking attacks and defenses in LLM-based agents. In *International Conference on Learning Representations*, 2024. arXiv:2410.02644.

A Fault Profile Calibration Details

This appendix documents how each field of the calibrated OPP profile consumed by MCP-Injector (file `fault_modes/fault_mode_1.json`) is obtained from the mined corpus of Section 4. The profile contains 18 entries, one per combination of six operators (OP1 through OP6) and three injectable persistence types (INTERMITTENT, PERMANENT, TRANSIENT). Each entry specifies three kinds of values: (i) a selection weight (`probability`), (ii) a persistence schedule parameter (`activation_p`, and for INTERMITTENT also `deactivation_p`), and (iii) operator-specific content parameters (`duration`, `delta_ratio`, `probability_per_word`). We describe how each is set, separating values that are mechanically derived from the mined distribution (Table 2) from hyperparameters that are fixed on principled grounds. Table 3 lists every entry alongside its source counts.

Mapping failure modes to operators. The six operators implement one mined failure mode each: DELAYED_RESPONSE maps to OP5, EXPLICIT_ERROR to OP4, INCORRECT_OUTPUT to OP2, INVALID_OUTPUT to OP3, MINOR_MISTAKE to OP1, and SILENT_DROP to OP6. The UNCERTAIN mode row and the UNCERTAIN persistence column of Table 2 are excluded from the profile because they carry no actionable (mode, persistence) label. The NON_FAILURE row is also excluded, since non-failures are represented by the null injector (OP0) used in the no-fault baseline OP0.

Selection weight. For each (operator, persistence) cell, the selection weight is

$$\text{probability} = \frac{\text{count}}{N}, \tag{1}$$

where $N = 13{,}824$ is the sum of all injectable cells in Table 2 (that is, the three injectable persistence columns for the six mined failure modes, excluding UNCERTAIN). This preserves the empirical joint distribution of (failure mode, persistence) as a probability mass function over the 18 operator-persistence entries, with the rare UNCERTAIN mass reassigned proportionally. For example, EXPLICIT_ERROR/PERMANENT has 9,075 threads, yielding $9075/13824 \approx 0.6565$; MINOR_MISTAKE/INTERMITTENT has 15 threads, yielding $15/13824 \approx 0.0011$. At injection time, MCP-Injector samples one entry per candidate fault episode with these weights, so the long-run frequency of every (mode, persistence) pair matches its mined prevalence.

INTERMITTENT activation probability. Equation (2) was designed for a two-state schedule that alternates between faulty and correct phases with per-call transition probabilities $a = \texttt{activation_p}$ and $d = \texttt{deactivation_p}$, whose stationary faulty fraction is $a/(a+d)$; setting that fraction to the proportion of recurring (non-transient) observations of the mode classified as INTERMITTENT rather than PERMANENT gives:

$$\frac{a}{a+d} \;=\; \frac{\text{INT}}{\text{INT} + \text{PERM}} \qquad \Longrightarrow \qquad a \;=\; d \cdot \frac{\text{INT}}{\text{PERM}}. \tag{2}$$

We fix $d = 0.3$ across all entries (see below), which determines a uniquely for each operator. For instance, DELAYED_RESPONSE has $\text{INT}/\text{PERM} = 51/157$, giving $a = 0.3 \cdot 51/157 \approx 0.0975$. All six INTERMITTENT activation probabilities in the profile are reproduced by Equation (2) to four decimal places. The shipped injector, however, uses these parameters differently: once an INTERMITTENT entry is drawn, each call fires independently with probability a, and from the next call on the entry is released with probability d per call, after which a new entry is drawn from the profile. The faulty fraction while the entry is held is therefore a rather than $a/(a+d)$, and an episode spans $1 + 1/d \approx 4.3$ calls on average, so it is expected to fire $a(1 + 1/d) \leq 0.42$ times. The six activation probabilities are data-derived in the sense of Equation (2), but under the implemented schedule INTERMITTENT entries fire far less often than the mined INT/(INT+PERM) ratios imply, consistent with the 6 INTERMITTENT events in the fired-faults log (Appendix C).

TRANSIENT activation probability. TRANSIENT faults, by definition, fire on a single tool call and then reset (Section 3.2). Once a TRANSIENT entry is selected, the corresponding call is faulty with certainty, so $\texttt{activation_p} = 1.0$ for every TRANSIENT entry. The mined frequency of each (mode, TRANSIENT) cell is reflected in the selection weight, not in the schedule parameter.

PERMANENT schedule. A PERMANENT entry has no activation parameter: once the entry is selected, the operator fires on every subsequent matching call (Section 3.2).

Choice of $d = 0.3$ for INTERMITTENT deactivation. Equation (2) identifies only the ratio a/d from data. The mined threads do not provide reliable episode-length statistics (GitHub issues rarely report call-level burst lengths), so d is a free hyperparameter. In the two-state design it sets the mean faulty burst to $1/d \approx 3.3$ calls; in the shipped injector it sets how long a drawn INTERMITTENT entry is held ($1 + 1/d \approx 4.3$ calls on average). Either way the episode is short relative to the execution loops (up to 25 or 40 rounds per task), so a single INTERMITTENT episode does not monopolize a task execution. Choosing a common d for all operators keeps the calibration one-dimensional per mode and ensures that cross-operator comparisons of INTERMITTENT effects are not confounded by different episode lengths. The loop bounds themselves differ across stacks and are documented in Appendix F.

OP5 delay duration. The MCP HTTP transport timeout is 60 seconds (Section 5.1, default in `benchmark_config.yaml`). We set `duration` $= 30$ seconds: half the HTTP timeout, which is an order of magnitude above typical healthy tool latency (a few seconds) and therefore clearly disruptive, but strictly below the transport cutoff so that OP5 remains distinct from OP6 (SILENT_DROP) and from OP4 (EXPLICIT_ERROR) at the protocol boundary. Longer values would cause transport-level timeouts and collapse DELAYED_RESPONSE into SILENT_DROP; shorter values would risk being absorbed by the agent's natural per-call variance and not register as a delay.

OP1 and OP2 content-perturbation parameters. OP1 (MINOR_MISTAKE) and OP2 (INCORRECT_OUTPUT) both act word-by-word on the response, but realize qualitatively different failure modes. OP1 mutates each selected word by a single-character substitution of magnitude controlled by `delta_ratio`; OP2 replaces each selected word by a random value of the same type. The parameters are chosen to reflect the semantic separation between the two modes. OP1 uses `probability_per_word` $= 0.08$ and `delta_ratio` $= 0.05$, yielding sparse, low-magnitude perturbations that leave most of the response intact, consistent with the MINOR_MISTAKE label (small deviation, structurally valid). The 5% delta level is the canonical minor-perturbation intensity used in perturbation-robustness literature. OP2 uses `probability_per_word` $= 0.35$, which corrupts roughly one third of the response, consistent with INCORRECT_OUTPUT (semantically wrong but structurally valid). We avoid values close to 1 because a fully garbled response typically fails downstream JSON parsing, collapsing OP2 into OP3 (INVALID_OUTPUT). The 0.08 vs. 0.35 gap preserves a clear intensity ordering between the two content operators; the mined corpus does not support finer tuning because threads rarely quantify the numeric extent of a "minor" versus "wrong" response.

OP3, OP4, OP6 parameters. These operators take no tunable parameters. OP3 deterministically casts the response to a type that violates the tool's schema, OP4 raises a structured MCP error, and OP6 suspends the call indefinitely. Their behavior is fully specified by the persistence schedule and requires no additional calibration.

Summary. Equation (2) and $\text{probability} = \text{count}/N$ are the two deterministic rules that map the mined distribution onto the JSON profile. The remaining values ($d = 0.3$; `duration` $= 30$; `delta_ratio` $= 0.05$; `probability_per_word` $\in \{0.08, 0.35\}$) are fixed hyperparameters whose roles are to (i) set the temporal scale of INTERMITTENT episodes, and (ii) separate content operators at semantically meaningful intensities without collapsing them into adjacent modes. Together these choices produce a fully specified, seedable fault profile whose selection weights are traceable to Table 2.

B Proxy Non-Intrusiveness: Full Analysis

The experimental design requires that the proxy with null injection (OP0) is outcome-equivalent to the no-proxy baseline (proxy-off). We test this by comparing task-quality scores and execution latency across all six (stack, model) combinations.

Table 3: Derivation of every entry in the OPP profile (`fault_mode_1.json`). Selection weights are count/N with $N = 13{,}824$. INTERMITTENT activation probabilities follow Equation (2) with $d = 0.3$; TRANSIENT activation is fixed at 1.0. Small residuals reflect four-decimal rounding in the shipped JSON.

Operator (mode)	Persistence	Count	prob.	act. p	Derivation of act. p
OP5 (DELAYED_RESPONSE)	INTERMITTENT	51	0.0037	0.0975	$0.3 \cdot 51/157$
	PERMANENT	157	0.0114	n/a	n/a
	TRANSIENT	1	0.0001	1.0	fixed
OP4 (EXPLICIT_ERROR)	INTERMITTENT	669	0.0484	0.0221	$0.3 \cdot 669/9075$
	PERMANENT	9075	0.6565	n/a	n/a
	TRANSIENT	97	0.0070	1.0	fixed
OP2 (INCORRECT_OUTPUT)	INTERMITTENT	155	0.0112	0.0268	$0.3 \cdot 155/1733$
	PERMANENT	1733	0.1254	n/a	n/a
	TRANSIENT	1	0.0001	1.0	fixed
OP3 (INVALID_OUTPUT)	INTERMITTENT	47	0.0034	0.0276	$0.3 \cdot 47/511$
	PERMANENT	511	0.0370	n/a	n/a
	TRANSIENT	1	0.0001	1.0	fixed
OP1 (MINOR_MISTAKE)	INTERMITTENT	15	0.0011	0.0254	$0.3 \cdot 15/177$
	PERMANENT	177	0.0128	n/a	n/a
	TRANSIENT	0	0.0000	1.0	fixed (unused)
OP6 (SILENT_DROP)	INTERMITTENT	167	0.0121	0.0520	$0.3 \cdot 167/963$
	PERMANENT	963	0.0697	n/a	n/a
	TRANSIENT	4	0.0003	1.0	fixed

Score equivalence. For each (stack, model) pair, we run paired Wilcoxon signed-rank tests on three composite metrics (task completion, tool selection, planning effectiveness) between proxy-off and OP0, pairing by (task, seed). After Bonferroni correction across 18 tests (6 pairs $\times$ 3 metrics), no comparison reaches significance at $\alpha = 0.05$. The smallest uncorrected p is mcpbench/20b on planning effectiveness ($p_{\text{uncorr}} = 0.003$, $r = 0.40$, Bonferroni $p = 0.050$), followed by mcpbench/20b on tool selection ($p_{\text{uncorr}} = 0.006$, $r = 0.39$, Bonferroni $p = 0.10$); all three mcpbench/20b deltas are small and negative (-0.27 to -0.45 points), so a slight proxy effect in that cell cannot be excluded. Mean score deltas are small: the largest absolute difference is 0.55 points (openai/120b task completion), with no consistent direction across stacks, confirming that the proxy does not systematically bias scores.

Latency overhead. Mean execution-time deltas (OP0 minus proxy-off) range from -22 s (langchain/120b) to $+7$ s (mcpbench/120b) in absolute seconds, and from -12% (openai/20b) to $+21\%$ (langchain/20b) as a fraction of the proxy-off baseline. The negative delta for LangChain reflects run-to-run variance in LLM inference time rather than proxy-induced speedup. Expressed as per-tool-call overhead, the proxy adds an estimated 0.1 to 0.5 s per tool invocation, which is negligible relative to LLM inference latency (typically 2 to 15 s per turn). Figure 5 shows the distribution of latency deltas by stack.

Conclusion. The proxy satisfies the non-intrusiveness requirement: OP0 is a valid fault-free baseline, and all subsequent OP0-vs-OPP comparisons are not confounded by instrumentation artifacts.

C Injection Fidelity: Full Analysis

This section checks how the specified profile translates into realized fault events under OPP and whether fault exposure is sufficient to support causal claims. We define the *fault dose* of an experimental cell as the number of fault injection events fired by the proxy during that cell's execution, computed by counting the cell's entries in the fired-faults log. Dose ranges from zero (when the agent makes no tool call, or draws an INTERMITTENT entry that never fires) to several dozen, depending on tool-call volume and the persistence state of each triggered fault.

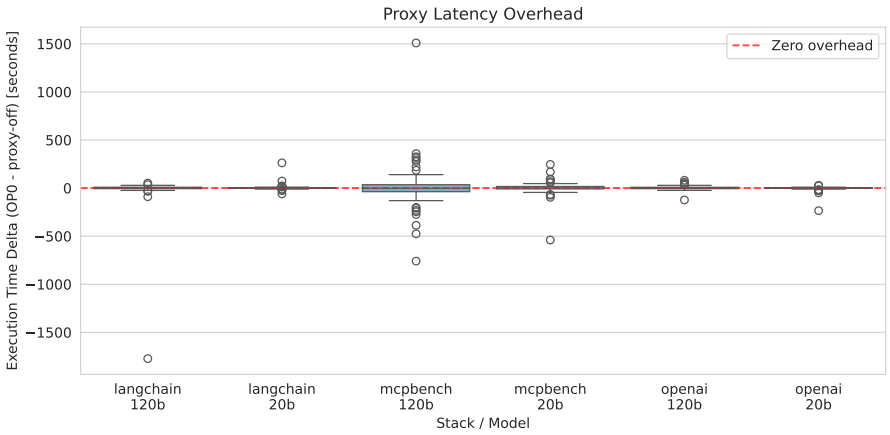


Figure 5: Distribution of execution-time deltas (OP0 − proxy-off) per task, by stack. Distributions are centered near zero with no systematic positive bias, confirming minimal proxy overhead.

Distribution fidelity. We compare the specified (fault operator × persistence) selection weights with the realized frequencies in the fired-faults log: 1,905 fired events from the 395 OPP cells with at least one injection (Figure 6). The per-event realized mix departs detectably from the specified weights ($\chi^2 = 684.5$, $\mathrm{df} = 16$ over the 17 entries with non-zero weight, $p < 0.001$; Cramér's $V = 0.15$; total variation distance 0.20). EXPLICIT_ERROR/PERMANENT remains the dominant entry (65.6% specified, 59.5% realized), but INCORRECT_OUTPUT/PERMANENT (OP2) rises from 12.5% to 28.9%; PERMANENT entries account for 99.5% of fired events against 91.3% of the weight, and INTERMITTENT entries (8.0% of the weight) fire only 6 times (0.3%). The departure reflects how the profile is consumed rather than a sampling error. Selection weights apply per draw, not per event: the injector draws an entry at the first tool call of a run, and a PERMANENT draw then fires on every subsequent call, so each such run contributes as many events as it makes tool calls. Event volume per draw depends on the entry and the stack: runs under OP2/PERMANENT, whose perturbed outputs return as successes, average 7.5 fired events, against 4.2 under EXPLICIT_ERROR/PERMANENT, a gap that holds within every stack; and MCPBench runs contribute 68% of all events (mean 8.6 per cell, versus 2.3 for LangChain and 1.8 for OpenAI). INTERMITTENT entries fire on a given call only with activation probability $a \leq 0.10$, and the median OPP cell makes two tool calls, so these entries rarely fire. At the level of the draw itself, the first fired entry of each run is consistent with the specified weights renormalized over the PERMANENT and TRANSIENT entries (391 runs; Monte Carlo $p = 0.22$, $V = 0.06$), although OP2/PERMANENT is drawn more often than expected (74 runs against 53). The profile thus fixes what is drawn, while persistence and agent behavior determine how many faults each draw delivers. Analyses that condition on exposure (zero-dose cells and exposure versus dose, below) use realized doses from the fired-faults log, and the OPP versus OP0 contrasts of Section 6.1 compare conditions as assigned, so neither depends on the per-event mix matching the profile.

Zero-dose runs. Of 450 OPP cells, 55 (12.2%) received no fault. Every profile entry targets all tools and every PERMANENT or TRANSIENT draw fires on its first call, so a cell is zero-dose only if the agent made no tool call (29 cells, 17 of them classified as tool bypass) or drew an INTERMITTENT entry that never fired (26 cells, consistent with the 8% INTERMITTENT selection weight; Appendix A). These cells are distributed across all stacks (LangChain: 21, MCPBench: 15, OpenAI: 19) and are concentrated in tasks where alternative solution paths exist (unit_converter, math_mcp, wikipedia). Including zero-dose cells in the primary analysis dilutes the measured fault effect; excluding them amplifies the degradation from -60% to -67% in the task completion score.

Exposure and dose. Exposure to at least one fault is strongly associated with degradation: OPP cells with one or more fired faults score 2.72 on task completion (OP0: 8.15 for the same cells), against 6.93 for zero-dose cells (OP0: 7.61). The *number* of faults is not. In a linear mixed-effects model `score` $\sim$ `fault_dose + stack + (1 | task_id)` fit by REML to the 366 completed OPP cells with nonzero dose, the dose coefficient is small and, if anything, positive for all nine metrics (task

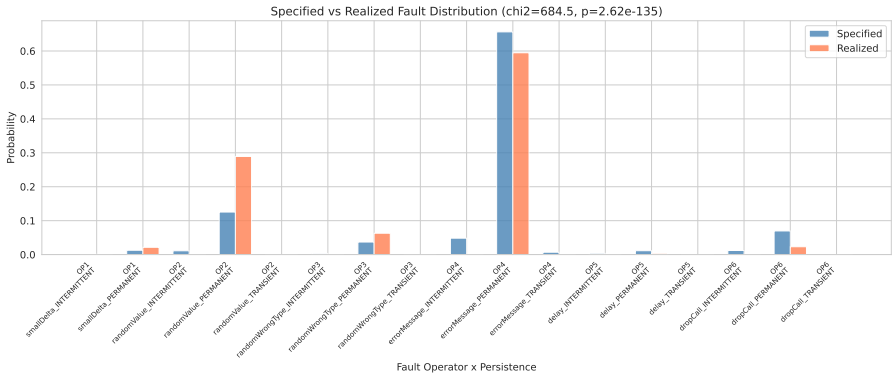


Figure 6: Specified selection weights vs. realized share of fired events (1,905 events from 395 OPP cells) across (operator $\times$ persistence) entries. Because a PERMANENT draw fires on every subsequent tool call, entries that sustain longer tool-call sequences (notably OP2/PERMANENT) are over-represented per event, and INTERMITTENT entries rarely fire. The title reports the per-event goodness-of-fit test.

completion: $+0.06$ points per fired fault, 95% CI $[-0.004, 0.12]$; including zero-dose cells: $+0.01$, $[-0.05, 0.07]$). Two features of the design explain this. Most runs receive a single PERMANENT draw that fires from the first tool call (see distribution fidelity above), so degradation sets in at first exposure; and dose is confounded with tool-call volume, since agents that keep calling tools, for instance under silent content faults, accumulate more events while still producing partial answers. Zero-dose cells are not a randomized control either: 29 of the 55 made no tool call, and their fault-free scores are lower. We therefore read the evidence as an exposure effect rather than a dose-response relationship.

D Supplementary Analyses of Fault Impact

Per-cell completion rates. Table 4 reports task completion rates for every (model, stack, condition) cell, expanding the headline numbers in Section 6.1. OP0 rates are 95 to 100% and proxy-off rates 93 to 100% across all (stack, model) cells; the largest OPP drops are for the 120b under MCPBench and OpenAI, although per-stack intervals overlap widely.

Table 4: Task completion rates (%) by model, stack, and proxy condition, with Wilson 95% CIs ($n = 75$ cells per stack row, 225 per pooled row). Per-stack intervals overlap widely; the largest OPP drops are for the 120b under MCPBench and OpenAI.

Model	Stack	Proxy-off	OP0	OPP
20b	LangChain	100.0 [95.1, 100.0]	100.0 [95.1, 100.0]	97.3 [90.8, 99.3]
	MCPBench	100.0 [95.1, 100.0]	100.0 [95.1, 100.0]	94.7 [87.1, 97.9]
	OpenAI	96.0 [88.9, 98.6]	97.3 [90.8, 99.3]	94.7 [87.1, 97.9]
	Pooled	98.7 [96.2, 99.5]	99.1 [96.8, 99.8]	95.6 [92.0, 97.6]
120b	LangChain	94.7 [87.1, 97.9]	96.0 [88.9, 98.6]	94.7 [87.1, 97.9]
	MCPBench	100.0 [95.1, 100.0]	100.0 [95.1, 100.0]	90.7 [82.0, 95.4]
	OpenAI	93.3 [85.3, 97.1]	94.7 [87.1, 97.9]	88.0 [78.7, 93.6]
	Pooled	96.0 [92.6, 97.9]	96.9 [93.7, 98.5]	91.1 [86.7, 94.2]

Per-metric score degradation. Table 5 reports the full nine-metric degradation summary referenced in Section 6.1.

Per-task score heatmap. Figure 7 shows percentage degradation for each of the 25 tasks across all nine evaluation metrics, pooled across stacks, models, and seeds. Degradation ranges from -19% (unit_converter, task completion) to -98% (game_trends, grounding). Tasks involving live APIs or

Table 5: Score degradation under fault injection (OPP vs. OP0), pooled across stacks and models. Paired drops have 95% CIs within ± 0.4 points and Benjamini-Hochberg corrected Wilcoxon $p < 0.001$ for every metric. †Task fulfillment counted only when tool appropriateness is at least 2 (Section 6.1).

Metric	OP0	OPP	Δ	% Drop
Task completion	8.08	3.23	−4.85	−60
Tool selection	8.10	3.36	−4.74	−59
Planning eff.	7.51	2.49	−5.02	−67
Task fulfillment	8.07	3.37	−4.70	−58
Grounded fulfillment†	7.70	2.83	−4.87	−63
Grounding	8.09	3.10	−5.00	−62
Tool appropriateness	8.21	3.81	−4.40	−54
Parameter accuracy	8.00	2.91	−5.08	−64
Dependency awareness	7.94	2.75	−5.19	−65
Parallelism eff.	7.08	2.23	−4.85	−68

deep sequential chains are consistently the most vulnerable, while short-chain computational tasks show the most resilience.

Completed-only analysis. Restricting to cells that completed under both OP0 and OPP, task completion score drops by 57% (from 8.26 to 3.53), compared to 60% in the primary analysis. The gap between all-in and completed-only is modest (3 percentage points), indicating that non-completion, while present, is not the dominant driver of score degradation; quality loss among completed runs accounts for most of the effect.

Case studies. We present detailed case studies for all six failure categories in Appendix E, with representative reasoning traces illustrating how agents cope with injected faults.

E Failure Mode Case Studies

To ground the failure mode taxonomy (Section 6.1), we present annotated case studies for each of the six categories. All examples are drawn from the langchain/120b condition under OPP (seed 1), which exhibits the widest behavioral diversity. Scores are reported on the 0 to 10 scale; "nan" indicates system failure (non-completion).

Non-completion. `car_price_evaluator_000` and `metropolitan_museum_000` both failed outright, producing no scorable output (all metrics nan). In both cases, the agent encountered persistent access-denied errors and exhausted its turn budget retrying. The reasoning trace for `car_price_evaluator_000` reveals repeated attempts with minor parameter variations: "*Since previous attempts denied access, maybe still can try? We have only three tools all denied previously. Might be permission issue; perhaps we cannot get data. Could stop.*" The agent recognized the futility of retrying but continued anyway, a pattern consistent with the absence of explicit fault-type diagnosis noted in Appendix G.

Tool-bypass / hallucination. `openapi_explorer_000` completed with a task score of 4.4 but tool appropriateness of only 0.8: the agent produced a detailed comparison of the GitHub and GitLab REST APIs entirely from parametric knowledge, never successfully calling the designated OpenAPI exploration tool. More strikingly, `unit_converter_000` scored 9.8 on task fulfillment (9.3 task completion) by computing unit conversions from memory without any tool call, receiving only 1.2 on tool appropriateness; since it made no call, no fault reached it. These cases are qualitatively concerning because a downstream consumer would see plausible, high-quality output with no indication that the designated tools were never consulted.

Graceful degradation. `google_maps_000` exemplifies this category: the agent acknowledged tool unavailability and offered alternative approaches ("*It looks like I'm currently unable to query the nearby-search tool. If you'd like, I can try a different approach*"), but received 0.0 across all metrics

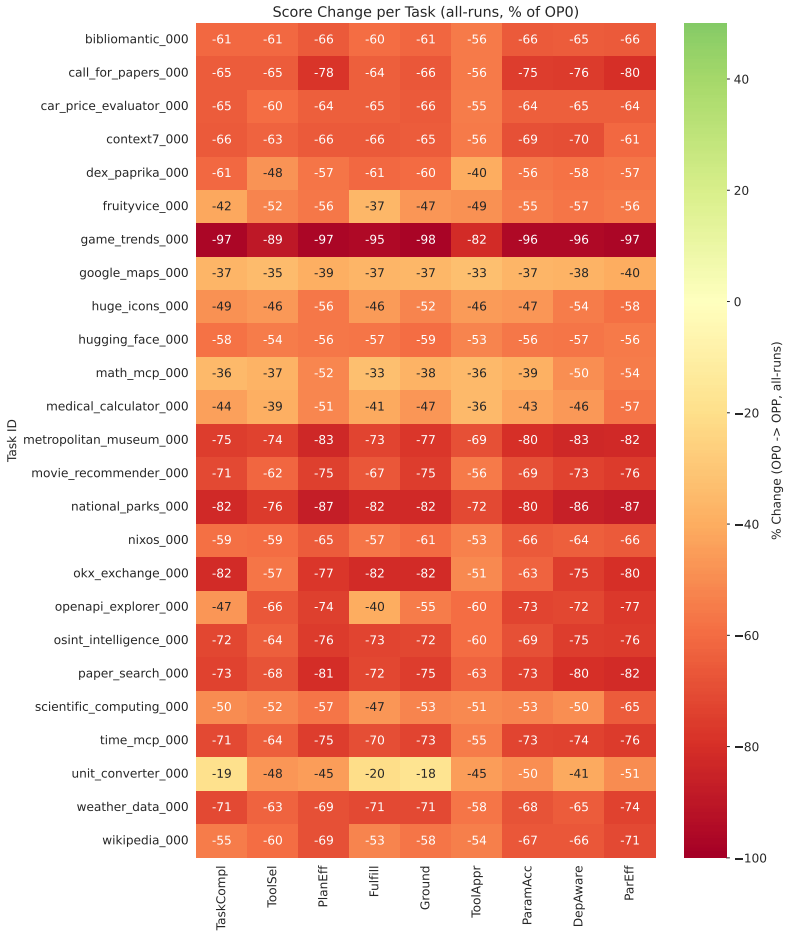


Figure 7: Per-task score degradation (OPP vs. OP0, % change) across all nine evaluation metrics. Tasks are ordered alphabetically. Darker cells indicate more severe degradation.

because it produced no tool-grounded answer. `context7_000` similarly degraded gracefully, falling back to parametric knowledge about Express middleware with a task score of 0.2. The reasoning trace shows explicit recognition of the problem: "*Since resolve-library-id failed due access denied, maybe cannot use tools? We might skip and answer from knowledge.*" These cases highlight a tension in current evaluation rubrics: scores penalize tool non-use even when the agent's behavior is arguably adaptive.

Catastrophic completion. `bibliomantic_000` and `metropolitan_museum_000` (seed 1) both completed but scored 0.0 across all metrics. The agent produced polite refusals ("*I'm sorry, but I don't have the ability to run the I Ching-related tools*") that technically constitute a response but provide no task-relevant content. The reasoning traces for `bibliomantic_000` are particularly revealing, showing four consecutive planning iterations where the agent hypothesizes parameter format issues: "*The previous attempts failed due to output format issues; maybe we need proper parameters. For i_ching_divination tool expects query optional string.*" At no point does the agent recognize that the fault is permanent rather than a transient parameter mismatch.

Partial degradation. `dex_paprika_000` (task score 0.4, tool appropriateness 2.2) and `fruityvice_000` (task score 1.9, tool appropriateness 4.0) represent the modal outcome under fault injection. In both cases, the agent made some successful tool calls but produced incomplete or partially incorrect answers. `fruityvice_000` illustrates a common pattern: the agent retrieved partial data about fruit nutritional content, supplemented it with parametric knowledge of vari-

able accuracy, and produced a mixed-quality response that scores well on structure but poorly on grounding.

Unaffected. `call_for_papers_000` (task score 9.4, tool appropriateness 9.8) and `huge_icons_000` (task score 10.0, tool appropriateness 10.0) completed with near-perfect scores despite running under OPP. For `call_for_papers_000`, the reasoning trace shows the agent quickly pivoting after tool failure: "*Since get_events is denied maybe we can't use? … User wants info on upcoming AI conferences … Since tool calls failed, I must rely on my knowledge.*" Despite falling back to parametric knowledge, the agent produced a comprehensive, accurate answer because the task domain (well-known conference information) is well-represented in training data. `huge_icons_000` succeeded because the injected faults did not affect the specific tool calls required for the task (the fault was semantically benign for that call pattern). These cases underscore that resilience is task-dependent: the same fault profile can be devastating for one task and irrelevant for another.

Cross-cutting observations. Three patterns recur across categories. First, agents universally treat failures as transient, retrying with minor parameter variations rather than diagnosing fault persistence. Second, the quality of fallback behavior depends heavily on whether the task domain overlaps with the model's training data: API-dependent tasks with specialized schemas degrade catastrophically, while general-knowledge tasks degrade gracefully or not at all. Third, current evaluation rubrics do not distinguish between "correctly identified tool unavailability and provided best-effort answer" and "failed to use tools," conflating adaptive and maladaptive behaviors under the same low score.

F Agent Stack Orchestration Details

This appendix documents the intrinsic orchestration behaviors that differ across the three agent stacks and that we allow to vary by design (Section 5.2). Task prompts, tool schemas, LLM backbones, decoding parameters, context budget, per-task wall-clock timeout, and the fault-injection layer are held constant. The differences below are relevant for interpreting the post-failure behavior in Section 6.2 and the retry-loop analysis in Appendix G. We retain the labels **MCPBench**, **OpenAI**, **LangChain** from the main text; these follow the naming of the reference implementations rather than any claim about the orchestration pattern per se.

MCPBench executor. The reference MCP-Bench agent [28] is a multi-round *plan-and-execute* loop rather than a per-step reason-act-observe loop. On every round, a planner LLM call is asked to emit a strict JSON object with three keys: a free-text `reasoning` field, a Boolean `should_continue` gate, and a list `planned_tools` of independent tool calls. The executor then dispatches the planned calls concurrently with a per-server semaphore. Tool-call errors are surfaced back to the next planner call as part of an *accumulated information* buffer, which is LLM-summarized when it approaches the context budget. Retries exist at the LLM-call level (task/round nesting, token-budget fallbacks, format-fix re-prompts) but not at the tool-call level; whether to retry or switch tools is the planner LLM's decision.

OpenAI (Agents SDK). We use the OpenAI Agents SDK's `Runner.run` with `tool_choice="required"` on the chat-completions API, which gives the model native OpenAI function calling and parallel tool calls by default. The SDK runs a single message loop: each turn produces an assistant message optionally containing one or more `tool_calls`, the SDK executes them and appends `tool`-role messages with the results, and the loop terminates when the assistant message contains only text. MCP tools are exposed through a thin adapter that maps tool errors to a `CallToolResult` with `isError=True`, so the LLM sees a structured error message on the next turn and can retry, switch, or stop. No custom retry logic is added on top of the SDK.

LangChain (`create_agent`). The LangChain stack uses the v1 `create_agent` abstraction, which compiles a ReAct state machine onto LangGraph: alternating `agent` (LLM) and `tools` nodes until the LLM stops emitting `tool_calls` [15]. Multiple tool calls in a single assistant message are executed in parallel by `ToolNode` (internally, `asyncio.gather`). A lightweight tool-monitoring middleware wraps each tool invocation so that exceptions are caught and returned as a `ToolMessage`

payload, analogous to the `isError` path in the OpenAI adapter. As in the other stacks, any retry or tool-switching is decided by the LLM on the next agent step.

Loop bounds. For the MCPBench and OpenAI stacks, we cap the orchestration loop at 40 iterations (`max_execution_rounds=40` and `max_turns=40` respectively). The LangChain stack inherits LangGraph's default `recursion_limit` of 25 supersteps, which corresponds to roughly 12 agent/tool cycles. This asymmetry is a property of the stack's defaults rather than a controlled variable, and it is conservative for our analysis: a tighter loop could only *decrease* LangChain's observed retry and tool-switching rates, so it cannot explain the fact that LangChain exhibits the *highest* OPP completion rate in Section 6.2. We also cap each run at a shared 3000-second wall-clock timeout. The timeout binds before either MCPBench or LangChain saturates its loop bound; a small fraction of OpenAI runs (14 of 450, about 3%) do reach `max_turns=40` before the timeout fires.

Implications for stack sensitivity and behavioral adaptation. Table 6 summarizes the intrinsic orchestration differences across the three stacks. The stacks differ most in how the decision to stop further tool calls is represented, and this shapes the post-failure profile reported in Section 6.2. MCPBench drives the loop through a per-round planner that re-plans the full tool list on every round, so after a fault the natural continuation is to plan a different tool; the result is a switch-dominant profile (68% switch, 21% retry, 11% fallback). OpenAI and LangChain both encode "stop" implicitly as "emit an assistant message with no tool calls", so once the LLM runs out of hypotheses about the failing tool the path of least resistance is to verbalize a final answer; both stacks are fallback-leaning (49% and 44% fallback, respectively). LangChain occupies the middle ground on tool-switching (33%) and shows the highest retry rate (24%), consistent with its ReAct loop re-entering the agent node after each tool error. The richer LLM-call retry machinery in the MCPBench stack, combined with its higher per-run tool-call count (mean 8.3 under OPP versus 2.5 for LangChain and 1.9 for OpenAI; Appendix G), also explains why the most severe retry-loop cases in Appendix G concentrate in the MCPBench agent stack.

Table 6: Intrinsic orchestration differences across the three stacks. Items listed as "LLM decides" indicate the stack has no built-in mechanism and delegates the decision to the model on the next orchestration step.

	MCPBench	**OpenAI (Agents SDK)**	**LangChain** (`create_agent`)
Orchestration pattern	Multi-round plan-and-execute with JSON planner output	Single message loop with native function calling	ReAct state graph on LangGraph (agent/tools nodes)
Tool invocation	Structured JSON prompt (`planned_tools` list)	Native function calling (`tool_choice` required)	Native tool calling on `ChatOpenAI`
Parallel tool calls	Explicit per-round, per-server semaphore	SDK-level fan-out	`asyncio.gather` in `ToolNode`
Tool-error handling	Error text added to accumulated information buffer	Returned as `isError` tool result	Caught by middleware, returned as tool message
Tool-call retry policy	LLM decides (next round)	LLM decides (next turn)	LLM decides (next agent step)
Stop decision	Explicit `should_continue` Boolean per round	Implicit (assistant message with no tool calls)	Implicit (AIMessage with no tool calls)
Loop bound	40 rounds	40 turns	25 supersteps (LangGraph default)
Reasoning format	Free text in JSON `reasoning` field plus accumulated buffer	Native assistant/tool message history	AI/tool message list on shared state

G Behavioral Adaptation: Reasoning Traces and Retry Loops

Fault-response classification. For each fault-injected tool call under OPP we classify the agent's immediate response into one of four mutually exclusive actions based on the subsequent tool-call

trace: *retry* (the next tool call targets the same tool that just failed), *switch* (the next tool call targets a different tool), *fallback* (no further tool call is issued and the run completes) or *timeout* (no further tool call is issued and the run does not complete). Across 1,302 fault-adjacent calls the three stacks produce distinct behavioral signatures (Figure 8a). MCPBench is switch-dominant: 68% of failures are followed by a call to a different tool, with only 11% resulting in fallback and 21% in retry. LangChain splits 44% fallback, 33% switch, 24% retry. OpenAI is fallback-leaning: 49% fallback, 33% switch, 18% retry. No stack records any timeout in this classification: every fault-adjacent call is followed either by another tool call or by a completed run terminating in text. Pooled across stacks, 57% of failures are followed by a switch, 22% by fallback, and 21% by retry; the pooled picture is dominated by MCPBench simply because its plan-and-execute loop issues far more tool calls per run (mean 8.3 under OPP, versus 2.5 for LangChain and 1.9 for OpenAI; Figure 8b). A paired Wilcoxon comparison of tool-call counts between OP0 and OPP is non-significant within every stack ($p \geq 0.09$), so the fault injection does not appreciably change how much the agent tries to use tools; it changes *which* tool the agent calls next.

Reasoning trace analysis. Examining chain-of-thought traces from MCPBench/120b runs (the only stack that exposes an explicit `reasoning` field on every planner call) reveals a recurring pattern: the model observes a tool failure, hypothesises a parameter or format issue ("maybe need correct usage", "proper parameters"), and either re-issues the same call with a minor variation or pivots to a nearby tool. Precise fault diagnosis, for instance distinguishing a permanent error from a transient one, is absent in all examined traces (see `rq6_reasoning_excerpts.md` for the curated excerpts). Agents therefore treat all failures as potentially transient, which is consistent with the switch- and retry-heavy response distribution above.

Retry loops. We separately flag runs in which the agent issues three or more consecutive calls to the same failing tool. Out of 38 such runs across all OPP conditions, 25 occur in MCPBench, 7 in LangChain, and 6 in OpenAI. The bias toward MCPBench is consistent with its richer LLM-call retry machinery (Appendix F), which gives the model the most opportunities to re-issue the same tool call before the planner aborts. The longest chain is `scientific_computing_000` (mcp-bench/120b/seed_3), with 12 consecutive retries over 22 tool calls and 47,761 tokens; the most token-expensive chain is the same task on 20b/seed_1 (52 tool calls, 107,849 tokens).

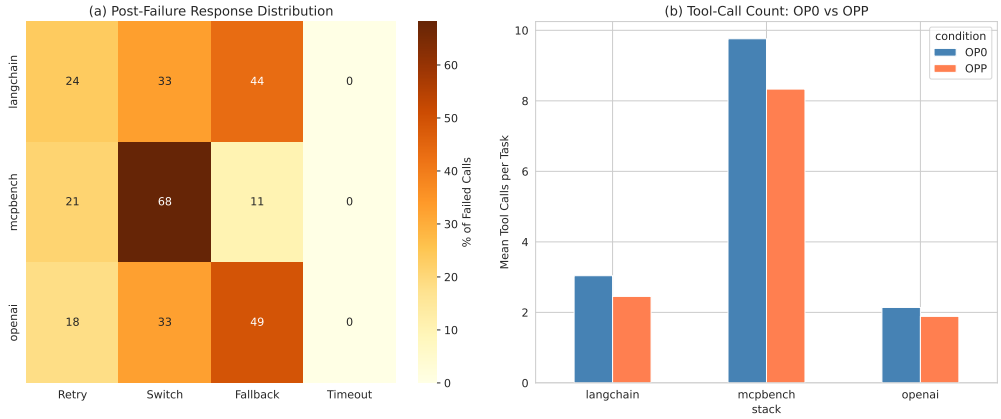


Figure 8: Post-failure behavior by stack. (a) Distribution of the immediate agent response after each fault-adjacent tool call, in percent of failed calls per stack; MCPBench is switch-dominant while LangChain and OpenAI are fallback-leaning. (b) Mean tool calls per task under OP0 versus OPP; within each stack the difference is small and non-significant (paired Wilcoxon, $p \geq 0.09$).

H Task-Domain Sensitivity

Task structural features. We extract two structural features for each of the 25 tasks: *chain depth* (number of distinct tool invocations in the dependency analysis, ranging from 6 to 29) and *domain class* (one of six categories: stateless retrieval, pure computation, multi-step sequential, stateful financial, real-time API, developer tooling).

Chain depth and resilience. We first examine the bivariate association between chain depth and per-task resilience, where the per-task resilience ratio is the median of OPP/OP0 task-completion scores across stacks, models, and seeds (Figure 9b). The Spearman correlation is $\rho = 0.29$ ($p = 0.17$, $n = 25$), positive but not significant. Chain depth, however, is itself correlated with mean fault dose ($\rho = 0.47$, $p = 0.02$), since longer dependency chains present more tool-call opportunities for faults to fire on. Partialling out fault dose via rank regression leaves a small and non-significant residual association ($\rho_{\text{partial}} = 0.35$, $p = 0.09$). We therefore do not claim that chain depth causally drives vulnerability in this dataset; we only report that the simple bivariate picture is confounded by fault exposure.

Domain-level comparison. Task-median resilience ratios vary substantially across the six domain classes (Figure 9a,c). Stateful financial tasks (dex_paprika, okx_exchange) are the least resilient (median 0.04, $n = 2$), followed closely by developer-tooling tasks (context7, hugging_face, nixos, osint_intelligence; median 0.05, $n = 4$) and real-time API tasks (game_trends, google_maps, time_mcp, weather_data; median 0.06, $n = 4$). Pure-computation tasks (math_mcp, scientific_computing; median 0.60, $n = 2$) and multi-step sequential tasks (median 0.61, $n = 5$) are the most resilient; stateless-retrieval tasks sit in between (median 0.18, $n = 8$). A Kruskal-Wallis test across the six domain classes is marginal ($H = 9.75$, $p = 0.08$). With as few as two tasks per class for pure computation and stateful financial, this comparison is underpowered; we report it as descriptive evidence that certain task profiles are reliably harder under OPP, not as a confirmatory domain-level effect.

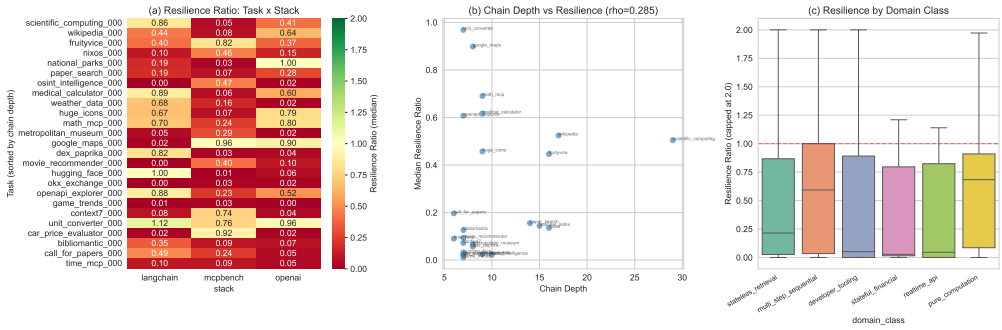


Figure 9: Task-domain sensitivity. (a) Per-task resilience ratio (median of OPP/OP0 across seeds) by stack, with tasks sorted by chain depth. (b) Task-median resilience ratio versus chain depth, labeled by task; the observed Spearman correlation is $+0.29$, $p = 0.17$. (c) Distribution of per-cell resilience ratios (capped at 2.0) by domain class; real-time API, stateful financial, and developer-tooling tasks cluster at the low-resilience end.

I Reproducibility Details

Serving hardware and software. Both backbones (gpt-oss-20B, gpt-oss-120B) run locally on a single Apple Mac Studio equipped with an M3 Ultra system-on-chip (28-core CPU, 60-core GPU, 32-core Neural Engine, 96 GB unified memory, 1 TB SSD), served through LM Studio 0.3.39 at their released MXFP4 precision. No additional quantization is applied, and no per-model prompt tuning is used; cross-model differences in our results are therefore attributable to model scale alone.

Campaign execution history. The full campaign underlying the 1,350 reported cells was executed over five successive rounds. Each round re-ran cells that had failed in prior rounds due to misconfigurations or timeouts. The first two rounds used a 1,500 s timeout and 20 execution rounds; from the third round on, the limits were raised to the final policy of Section 5.3. Of the 1,350 reported cells, 875 fault-free (Proxy-off and OP0) cells come from the first two rounds, and all of them completed without reaching the tighter limits (longest run 1,449 s, at most 19 MCPBench rounds). The remaining 25 fault-free cells and all 450 OPP cells ran under the final policy, as did all 51 reported non-completions. Cumulative compute across the five-round campaign, including all re-runs of cells that had failed under tighter timeout or round limits, was approximately 237 wall-clock hours

on the Mac Studio. Aggregate compute for the final 1,350 reported cells alone was approximately 58 wall-clock hours of agent execution plus about 18 hours of LLM-as-a-judge API time.

J Licenses of Used Assets

The paper relies on several public assets, all credited in the text and bibliography and used in compliance with their licenses. **MCP-Bench** [28] is the benchmark suite (Apache License, Version 2.0); we fork it and carry forward upstream attribution. The **OpenAI Agents SDK** [21] and **LangChain/LangGraph** [15] are both distributed under the MIT License. The **gpt-oss-20B** and **gpt-oss-120B** model backbones are released by OpenAI under the Apache License, Version 2.0; they are served locally via LM Studio 0.3.39. Bundled MCP server implementations under `mcp_servers/` retain their per-server upstream licenses (mostly MIT). The calibration corpus is mined via the GitHub REST API from 1,774 public repositories listed in *MCP Market* and *Awesome MCP Servers*, in compliance with GitHub's terms of service; raw issue and pull-request narratives are deliberately not redistributed, and reviewers can regenerate the raw corpus using their own `GITHUB_TOKEN`. Our own contributions are released under the Apache License, Version 2.0 (source code) and the Creative Commons Attribution 4.0 International license (CC-BY-4.0) for data artifacts (fault profiles, per-task result JSONs, fired-faults logs, and per-thread classifier outputs). A path-by-path mapping of licenses, the upstream attribution required by Apache 2.0 section 4(d), and a list of pinned commit SHAs and release tags for the third-party code dependencies used in the reported campaigns are documented in the artifact repository (`LICENSES-README.md`, `NOTICE`, and `REPRODUCING.md`).

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction (Section 1) state three contributions, each supported by a dedicated section: (1) MCP-Injector, the protocol-aware, drop-in MCP fault-injection middleware with seedable persistence schedules, is described in Section 3; (2) the MCP fault model, calibrated to the observed incident distribution of the public MCP ecosystem and mined from 57,240 threads across 1,774 open-source repositories, is detailed in Section 4; and (3) the 1,350-cell benchmark study across two backbone sizes, three agent stacks, and three fault injection conditions is reported in Section 6 (with apparatus-validation, mechanistic, and task-level analyses in Appendices B, C, D, G, and H), where the quantitative headline findings cited in the abstract (severe score degradation, tool bypass as a baseline behavior that faults expose, no resilience benefit from scale, and suggestive stack-mediated resilience) are each backed by analyses in the main results section and the supporting appendices. Scope boundaries (non-adversarial threat model, MCP tool-call boundary only, two backbone sizes, English-language mining corpus) are stated explicitly across Sections 2, 4, and 5 so that the claims reflect what the experiments actually test.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 7 is a dedicated Limitations section that discusses construct, internal, and external validity. For construct validity, we acknowledge that the LLM-as-a-judge evaluator is stochastic (mitigated by five evaluations per run with randomized rubric order and median scoring) and rewards some ungrounded answers, that the fault profile mined from GitHub issues and pull requests inherits their reporting bias and is therefore treated as a stress profile, and that the content operators omit schema drift, payload truncation, and partial result omission. For internal validity, we state that the shared 16K-token context window and cross-stack differences in tool-call volume are co-equal alternative explanations for the reported differences, that exposure and error handling remain partially entangled, that the claim that longer tool-call chains amplify exposure is a hypothesis, and that zero-dose cells (12.2% of OPP cells) and three seeds per cell limit power for per-task comparisons and rare interactions. For external validity, we flag that both backbones come from one model family under one serving setup, that MCP-Bench is easy without faults (OP0 completion 95 to 100%), that the study is powered for the ecosystem's 91%-permanent incident mix rather than for transient or intermittent regimes, that the fault profile reflects the open-source MCP ecosystem as of September 2025, and that the stacks and the 25 single-server tasks do not cover multi-agent, human-in-the-loop, multi-server, or longer-horizon settings. The paper rests on three load-bearing assumptions: a non-adversarial threat model and the MCP tool-call boundary as the correct injection point (both stated in Section 2), and the representativeness of the mined fault profile (stated in Sections 4 and 7). If the non-adversarial assumption is violated, an intelligent adversary may exploit patterns (timing side channels, crafted payloads) that our calibrated profile does not capture, so the measured degradation would understate worst-case behavior; if the tool-call boundary is not the dominant actuator for a given deployment, other

MCP surfaces (prompts, resources, streaming) may constitute additional and uncharacterized failure sources; if the open-source mining corpus is not representative of production usage, enterprise deployments (with private tools, stricter SLAs, different failure distributions) may experience degradation profiles that differ from those reported here. Computational efficiency is not a central concern for MCP-Injector, which is a transparent MCP middleware rather than a compute-bound algorithm and scales linearly with the number of tool calls per task (independent of any training-data notion of dataset size); we nonetheless characterize its runtime cost in Appendix B, where OP0 adds roughly 0.1 to 0.5 s per tool invocation (latency deltas ranging from -12% to $+21\%$ across stacks) with no Bonferroni-significant score difference from proxy-off. Privacy and fairness considerations do not directly apply, as MCP-Injector operates on synthetic tool-call traffic in non-human-subject benchmark tasks and neither collects user data nor makes decisions about people; the mined corpus draws only on public GitHub issue and pull-request metadata from open-source MCP repositories, with no personal or sensitive content retained. Together these scope boundaries bound the generalizability of specific quantitative findings (e.g., exact degradation percentages), while the qualitative conclusions (no resilience benefit from scale, partial degradation as the most common outcome, tool bypass as a baseline behavior) are likely more robust than the suggestive stack-level differences.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper presents a systems artifact (MCP-Injector), an empirically mined MCP fault model, and an empirical benchmark study; it does not claim theoretical results, theorems, lemmas, or proofs. The load-bearing assumptions of the empirical study (non-adversarial threat model, MCP tool-call boundary as the injection actuator, representativeness of the mined profile) are stated and discussed in Section 2 and Section 7.

   Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper discloses the full recipe needed to regenerate the main experimental results. Section 3 describes the MCP-Injector middleware (interposition model in Figure 1, four-stage Observe/Decide/Act/Record pipeline, the six fault operators plus the null passthrough in Table 1, the three persistence schedules, scope and composition rules, and the seeded randomness that drives the fault schedules, with every fired fault logged). Section 4 and Appendix A specify how the OPP fault profile was calibrated (57,240 threads across 1,774 open-source MCP repositories, classification pipeline, and the 18 injection rules with persistence semantics that constitute the profile). Section 5 then gives the full-factorial design ($2 \times 3 \times 3 \times 3 \times 25 = 1{,}350$ cells), the backbones (gpt-oss-20B and gpt-oss-120B, both publicly available open-weight models served with a 16,000-token context and default decoding), the serving hardware (a single Apple Mac Studio with M3 Ultra SoC, 28-core CPU, 60-core GPU, 96 GB unified memory), the three agent stacks (documented in detail in Appendix F so that plan format, parallelism, retry behavior, and loop bounds are all made explicit), the 25 single-server MCP-Bench tasks with distraction servers disabled, the per-task timeout and loop bounds governing all reported cells (3,000 s wall-clock; 40 rounds, 40 turns, or 25 supersteps depending on the stack), the judge configuration (gpt-4o-mini-2024-07-18, five evaluations per run with randomized rubric order and median aggregation), the paired within-task comparison $\Delta = \text{OPP} - \text{OP0}$, and the three seeds that fix the benchmark, injector, and framework-level RNGs. Appendices B, C, D, G, H, and E report the supplementary analyses, statistical tests, and behavioral traces underlying each headline finding. Because the backbones are open-weight, MCP-Bench is a public benchmark, and every relevant parameter of the injector, run policy, and judge is specified, a reader with the same models, tasks, and fault profile can reconstruct the experimental matrix from the paper alone. Access to the MCP-Injector middleware, the fault profile JSON files, the campaign scripts, the per-task result JSONs, and the analysis notebooks is addressed in the reply to the next question.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case

of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.

- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The public artifact at `https://gitlab.com/uniluxembourg/snt/sedan/papers/mcp-injector`, archived at `https://doi.org/10.5281/zenodo.23573765`, contains the MCP-Injector proxy middleware, the calibrated fault profile (`fault_modes/fault_mode_1.json`), the campaign and baseline runners, the full 1,350-cell evidence base under `analysis/results_formatted/`, the fault-mining pipeline under `mining/`, and the analysis notebook that produces every figure and table in the paper. The top-level `README.md` documents environment setup (LM Studio, conda, API keys), a single-cell quickstart, and full-campaign loops; `REPRODUCING.md` maps each paper figure and table to the notebook cell or script that produced it. Code is released under Apache 2.0 and data artifacts under CC-BY-4.0 (see `LICENSE`, `DATA-LICENSE`, and `LICENSES-README.md`).

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Section 5 specifies the benchmark (25 single-server MCP-Bench tasks with distraction servers disabled), the four-factor full-factorial design (two backbones, three agent stacks, three proxy conditions, three seeds, 25 tasks, 1,350 cells), the backbone configuration (gpt-oss-20B and gpt-oss-120B served at MXFP4 precision via LM Studio 0.3.39 on a single M3 Ultra Mac Studio with a 16,000-token context window and default decoding), the three proxy conditions (Proxy-off, OP0, OPP with 18 injection rules from the calibrated profile), the paired within-task design $\Delta = \text{OPP} - \text{OP0}$, the run policy (a 3,000 s per-task timeout and a stack-specific loop bound of 40 rounds, 40 turns, or 25 supersteps, with runs exceeding either limit classified as system-level failures), the judge configuration (gpt-4o-mini-2024-07-18, five evaluations per run with randomized rubric order and median aggregation), and the three metric families (task-quality scores, system-level outcomes, latency decomposition). Run-policy and judge hyperparameters follow MCP-Bench [28], which is the source of the benchmark itself. Appendix F details stack-level orchestration differences (plan format, parallelism, retry logic, loop bounds) and Appendix A details the 18-rule OPP profile and how it was calibrated. No model training is performed (open-weight backbones are used off the shelf), so optimizer and training-hyperparameter details are not applicable.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Every main-claim finding is reported with an effect size and a 95% confidence interval or an explicit significance test, and distributions are displayed as boxplots (median, IQR, whiskers) rather than symmetric error bars because most quantities of interest (bounded completion rates, resilience ratios, long-tailed latencies) are skewed or bounded. Score degradation under fault injection (Section 6.1 and Appendix D) uses paired Wilcoxon signed-rank tests across nine metrics with Benjamini-Hochberg correction (all corrected $p < 0.001$; paired drops are reported with 95% CIs). Proxy non-intrusiveness (Appendix B) uses paired Wilcoxon tests across 18 (stack, model, metric) combinations with Bonferroni correction (all corrected $p \geq 0.05$, largest effect size $r = 0.40$). Exposure versus dose (Appendix C) is examined with a linear mixed-effects model `score` $\sim$ `fault_dose + stack + (1 | task_id)`, reported with 95% CIs. Stack and model scaling effects (Sections 6.2 and 6.3) use mixed-effects models reported with 95% CIs (stack $\times$ condition: $\beta = -0.93$ $[-1.86, -0.01]$ for MCPBench and -0.84 $[-1.77, 0.08]$ for OpenAI; model size: $\beta_{\text{model}} = -0.009$ $[-0.39, 0.37]$). Task-characteristic analyses (Appendix H) use Spearman correlations, partial Spearman via rank regression, and a Kruskal-Wallis test across six domain classes. The factors of variability captured by the analyses are made explicit: the paired within-task design $\Delta = \text{OPP} - \text{OP0}$ (Section 5) removes task-difficulty variance and isolates condition-induced variance, three seeds per cell capture LLM decoding stochasticity, the random intercept `(1 | task_id)` in the mixed-effects model captures task-level heterogeneity, and the judge stochasticity is controlled by scoring each run five times with a randomized rubric order and taking the median. The nonparametric tests (Wilcoxon, Spearman, Kruskal-Wallis) make no Normality assumption on the scores; the mixed-effects models assume Normal residuals and Normal random intercepts, which we do not test formally. All p-values are computed via standard library functions (SciPy for Wilcoxon, Spearman, and Kruskal-Wallis; statsmodels for the mixed-effects models; the exposure-versus-dose model of Appendix C was fit with an equivalent REML implementation that

reproduces the statsmodels estimates of the stack $\times$ condition model), multiple comparisons are controlled with Bonferroni or Benjamini-Hochberg as appropriate, and effect sizes are reported alongside the p-values (matched-pairs rank-biserial r for Wilcoxon, Spearman ρ for correlations, β coefficients for the mixed-effects model). Figure and table captions restate the test and correction used (e.g., Table 5 reports Benjamini-Hochberg corrected Wilcoxon tests).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The paper discloses the compute needed to reproduce every reported experiment. All agent and backbone inference runs on a single Apple Mac Studio (M3 Ultra system-on-chip, 28-core CPU, 60-core GPU, 32-core Neural Engine, 96 GB unified memory, 1 TB SSD) serving gpt-oss-20B and gpt-oss-120B through LM Studio 0.3.39 at MXFP4 precision with a 16,000-token context window (Section 5.2). Two external API services are additionally used: the LLM-as-a-judge evaluator (gpt-4o-mini-2024-07-18) is called five times per completed cell via the OpenAI API (Section 5.3), and the fault-mining classifier (gpt-4o-2024-08-06) was invoked once to label 57,240 threads at a total API cost of approximately USD $800 (Section 4). Per-cell compute is bounded by a 3,000 s per-task wall-clock timeout and a stack-specific loop bound (Section 5.3); observed per-cell end-to-end and agent-internal execution times are reported in Appendix B. The final 1,350 reported cells required approximately 58 wall-clock hours of agent execution on the Mac Studio plus about 18 hours of LLM-as-a-judge API time (Section 5.3); cumulative Mac Studio compute across the five-round campaign was approximately 237 wall-clock hours. The full research project used more compute than the final reported cells: as disclosed in Section 5.3, the campaign was executed over five successive rounds in which earlier-round cells that had failed due to misconfiguration or timeout were re-run under progressively larger timeout and round limits; a small amount of additional compute was spent on pilot tuning of timeouts and proxy configuration, but it is negligible relative to the final campaign.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.

- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The authors have reviewed the NeurIPS Code of Ethics and the research conforms to it. The study does not involve human subjects, crowdsourcing, or user interaction, so IRB and compensation requirements do not apply (Questions 14 and 15 answer [N/A]). The only external corpus used is public GitHub issue and pull-request metadata from 1,774 open-source MCP repositories (57,240 threads), from which an aggregate 18-rule fault profile is derived; no personal or sensitive content is retained. Experiments are run on off-the-shelf open-weight backbones (gpt-oss-20B and gpt-oss-120B) with synthetic tool-call traffic on public MCP-Bench tasks, so no personally identifiable information is processed. The released artifacts (the MCP-Injector middleware, the calibrated fault profile `fault_mode_1.json`, the campaign runners, the analysis notebook, and the 1,350-cell evidence base) are published at `https://gitlab.com/uniluxembourg/snt/sedan/papers/mcp-injector`, archived at `https://doi.org/10.5281/zenodo.23573765`, under Apache 2.0 (code) and CC-BY-4.0 (data), with licenses and intended use documented in `LICENSE`, `DATA-LICENSE`, and `LICENSES-README.md`. The threat model is explicitly non-adversarial (Section 2), MCP-Injector is a transparent local MCP proxy with no credential handling or lateral-movement capability, and the fault profile reproduces naturally occurring open-source failure patterns rather than novel attack payloads, which bounds the dual-use surface. Transparency obligations are met by full disclosure of the experimental recipe, statistical tests with multiple-comparison corrections, both primary (all-runs) and completed-only analyses, and an explicit discussion of construct, internal, and external validity threats (Section 7). The submitted version was fully anonymized; this camera-ready version is de-anonymized as required.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: The paper includes a dedicated Broader impacts section (Section 8) covering positive impacts (reproducible pre-deployment stress testing of tool-calling agents and a fault profile calibrated to the observed incident distribution of the public MCP ecosystem), risks under intended use (the proxy as a high-privilege surface for tool-call data), miscalibration risks (the profile should be treated as a stress test rather than a certification instrument), and misuse risks (the operator primitives repurposed as an adversarial MCP middleware), together with three mitigations: null-injector default, fault-header tagging of injected responses, and pre-deployment-only framing.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [Yes]

    Justification: We release the MCP-Injector proxy and a text-only empirical fault-mode dataset mined from public GitHub issues and pull requests. We do not release any pre-trained language model, image generator, or scraped image dataset. The proxy is explicitly framed under a non-adversarial threat model (Section 2) and is architecturally scoped to MCP deployments the operator already controls (the agent must be reconfigured to route through the proxy, which provides no mechanism to attack third-party servers, harvest credentials, or persist beyond the session). The released artifact includes a usage note restricting use to research and dependability evaluation, logs every fault activation for auditability, and ships aggregated thread metadata and failure-mode classifications rather than the raw scraped corpus.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All external assets used in the paper (the MCP-Bench benchmark, the OpenAI Agents SDK, LangChain/LangGraph, the gpt-oss model backbones, the bundled MCP server implementations, and the GitHub-mined calibration corpus) are credited in the text and bibliography and used in compliance with their licenses. Appendix J lists each asset together with its license (Apache 2.0, MIT, or CC-BY-4.0) and notes that mining of

the calibration corpus respects GitHub's terms of service. The artifact repository ships `LICENSES-README.md` (path-by-path license map), `NOTICE` (Apache 2.0 section 4(d) upstream attribution), and `REPRODUCING.md` (pinned commit SHAs and release tags for the third-party code dependencies used in the reported campaigns).

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: The paper releases four new assets, each documented alongside the assets in the public artifact repository `https://gitlab.com/uniluxembourg/snt/sedan/papers/mcp-injector` (archived snapshot: `https://doi.org/10.5281/zenodo.23573765`): the MCP-Injector proxy middleware and its wiring in the benchmark runner, the calibrated fault profile `fault_modes/fault_mode_1.json` (and its null-injector counterpart `fault_mode_0.json`), the per-thread classifier outputs from the 57,240-thread / 1,774-repository mining pipeline under `mining/`, and the full 1,350-cell campaign result corpus under `analysis/results_formatted/` together with the notebook that produces every paper figure and table. Documentation is shipped with the assets: `mcp-bench/README.md` (artifact summary, environment and API-key setup, LM Studio configuration, quickstart, full-campaign loops, standalone injector launch path, and verification of fired faults), `mcp-bench/REPRODUCING.md` (per-figure and per-table mapping to the producing notebook cell or script), `fault_modes/README.md` (operator and persistence semantics), and `mining/README.md` (September 2025 snapshot date, scope statement, rate-limit and roughly USD $800 OpenAI budget, cold-regeneration recipe, and cross-reference to Section 7). License metadata is shipped as `LICENSE` (Apache 2.0 for code), `DATA-LICENSE` (CC-BY-4.0 for fault profiles, fired-fault logs, per-task result JSONs, and per-thread classifier outputs), `NOTICE` (upstream attribution to `accenture/mcp-bench`, and the MCP-Injector authors for our additions, listed in `AUTHORS`), and `LICENSES-README.md` (a path-to-license map); per-file SPDX headers declare the same terms on new sources. No human-subject consent applies: the mined corpus is drawn from public GitHub issue and pull-request threads, and we redistribute only thread identifiers, repository identifiers, classification labels, persistence categories, thread titles, and one-sentence classifier rationales, with raw issue or pull-request bodies, comment chains, and review-comment chains excluded, so third-party repository licenses do not propagate into our CC-BY-4.0 scope. The release carries no development history: it is published as a release snapshot, with author home-directory paths, operational logs, and local editable-install entries removed, and an automated sweep confirms that no credentials, API keys, or local working notes are present in the published tree.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.

- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or human subjects. All experiments use automated runs of open-weight backbones (gpt-oss-20B and gpt-oss-120B) on public MCP-Bench tasks with an LLM-as-a-judge evaluator (gpt-4o-mini-2024-07-18), and the fault-mode calibration corpus is classified automatically by gpt-4o-2024-08-06 from public GitHub issue and pull-request metadata. The only human labeling is the inter-annotator reliability audit reported in Question 16 and Section 4 (Cohen's $\kappa = 0.93$ for failure mode, $\kappa = 0.80$ for persistence), which was performed by the authors on 117 public GitHub threads as a quality check on the automated classifier; no crowdworkers were recruited or compensated, and no data about the annotators was collected. This is consistent with the [N/A] for Question 15 (IRB) and with the Code-of-Ethics declaration in Question 9.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The study involves no human subjects, interviews, surveys, or crowdsourcing, so IRB review (or equivalent) was not required. The fault-profile dataset underlying MCP-Injector is mined programmatically from public open-source MCP repositories (issue and discussion threads), and is analyzed only in aggregate (fault-category proportions and persistence distributions); no personal, private, or identifiable data are collected or released. All dependability experiments are fully automated: agent stacks operate on MCP tool calls, LLM backbones are served locally, and evaluation is performed by an automated judge. No participants were recruited, compensated, or exposed to risk. The inter-annotator reliability audit reported in Question 16 and Section 4 was performed by the authors on 117 public GitHub threads; it does not constitute human-subjects research, and no data about the annotators was collected.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are a core, non-standard component of this work in three distinct roles, each documented in the paper. (i) Agent backbones under test: gpt-oss-20B and gpt-oss-120B are the subjects under evaluation, served locally via LM Studio 0.3.39 at MXFP4 precision with a 16,000-token context window and default decoding across three agent stacks (MCPBench multi-round plan-and-execute, OpenAI Agents SDK, LangChain/LangGraph), as specified in Section 5.2. (ii) LLM-as-a-judge evaluator: gpt-4o-mini-2024-07-18 scores each completed run five times with randomized rubric order and median aggregation, as specified in Section 5.3. (iii) Fault-mining classifier: gpt-4o-2024-08-06 was invoked (temperature 0, fixed JSON schema) to classify 57,240 threads from 1,774 open-source MCP repositories, yielding 14,829 labeled (failure mode, persistence) failure instances, at a total API cost of approximately USD $800, as described in Section 4 and Appendix A; the 18-rule OPP fault profile (`fault_modes/fault_mode_1.json`) is derived directly from these classifier outputs and is therefore a load-bearing product of LLM use, with reliability supported by an independent human audit (Cohen's $\kappa = 0.93$ for failure mode, $\kappa = 0.80$ for persistence). Any use of LLMs for writing or editing assistance did not influence the scientific claims, experimental design, statistical analyses, or conclusions, and is excluded from this declaration per the NeurIPS LLM policy.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.