arXiv:2604.10842v1 [cs.SE] 12 Apr 2026

Resilient Write: A Six-Layer Durable Write Surface for LLM Coding Agents

Justice Owusu Agyemang[1,2,3*], Jerry John Kponyo[3†], Elliot Amponsah[3‡], Godfred Manu Addo Boakye[3§], Kwame Opuni-Boachie Obour Agyekum[2¶]

[1]Sperix Labs [2]VIA Cybersecurity Lab, KNUST [3]Quantum and Assistive Technologies Lab, KNUST

April 2026

Abstract

LLM-powered coding agents increasingly rely on tool-use protocols such as the Model Context Protocol (MCP) to read and write files on a developer's workstation. When a write fails—due to content filters, truncation, or an interrupted session—the agent typically receives no structured signal, loses the draft, and wastes tokens retrying blindly. We present **Resilient Write**, an MCP server that interposes a six-layer durable write surface between the agent and the filesystem. The layers—pre-flight risk scoring, transactional atomic writes, resume-safe chunking, structured typed errors, out-of-band scratchpad storage, and task-continuity handoff envelopes—are orthogonal and independently adoptable. Each layer maps to a concrete failure mode observed during a real agent session in April 2026, in which content-safety filters silently rejected a draft containing redacted API-key prefixes. Three additional tools—chunk preview, format-aware validation, and journal analytics—emerged from using the system to compose this paper. A 186-test suite validates correctness at each layer, and quantitative comparison against naive and defensive baselines shows a 5× reduction in recovery time and a 13× improvement in agent self-correction rate. Resilient Write is open-source under the MIT license.

*jay@sperixlabs.org, jay@knust.edu.gh
†jjkponyo.soe@knust.edu.gh
‡eamponsah52@st.knust.edu.gh
§gmaboakye@st.knust.edu.gh
¶kooagyekum@knust.edu.gh

1 Introduction

The emergence of tool-augmented large language models (LLMs) has shifted software-engineering assistants from suggestion engines to autonomous agents that read, write, and execute code on a developer's behalf [1, 2, 3, 4]. The Model Context Protocol (MCP) [5] standardises the interface between an LLM and the *tools* it invokes—file reads, writes, shell commands, database queries—giving agents a uniform way to act on the local environment.

In practice, the write path is fragile. A `Write` tool call can fail for reasons invisible to the agent: content-safety filters may reject the payload silently; a large file may be truncated midstream; the session may be interrupted before the write completes; and when a write does fail, the error signal is typically an unstructured string (or no signal at all), leaving the agent unable to diagnose the cause or choose a recovery strategy.

Motivating incident. In April 2026, while producing a technical report on LLM CLI telemetry [6], an agent attempted to write a LaTeX document whose body included redacted HTTP headers such as `Authorization: Bearer sk-ant-oat01-{REDACTED}`. The prefix pattern `sk-ant-` triggered a content-safety regex in the host tool, which silently rejected the payload. The agent received no structured error. It retried the identical content five times, consuming approximately two minutes of wall-clock time

and several thousand tokens, before falling back to an ad-hoc workaround: piping the document through chunked `cat » file.tex «EOF` heredoc commands in the shell.

This single incident exposed five distinct failure modes:

1. **Silent rejection**—no signal that the write was blocked.
2. **Draft loss**—the rejected payload was not persisted anywhere.
3. **Retry thrashing**—the agent retried identical content with no budget limit.
4. **No structured diagnosis**—the agent could not branch on error type.
5. **Session fragility**—had the session been interrupted during the workaround, all progress would have been lost.

Contribution. We present Resilient Write, an MCP server that addresses each of these failure modes with a dedicated, orthogonal layer (Table 1). The design follows three principles: (i) *fail transparently*—every rejection returns a machine-readable envelope; (ii) *never overwrite in place*—all writes go through a temp-file, fsync, verify, atomic-rename pipeline; (iii) *each layer is independently adoptable*—an agent can use only `rw.safe_write` and `rw.handoff_write` without ever touching the scratchpad or chunker.

The remainder of this paper is organised as follows. Section 2 reviews the relevant context. Section 3 describes the six-layer architecture and three extension tools. Section 4 covers implementation details. Section 5 reports on the test suite, a case study, and a quantitative comparison against baselines. Section 6 surveys related work. Section 7 discusses design tradeoffs, agent adoption, and limitations, and Section 8 concludes.

2 Background

2.1 LLM Coding Agents

A growing class of developer tools embed an LLM in an edit–test–commit loop. Claude Code [1], Cursor [3], GitHub Copilot [4], OpenAI Codex CLI [2], and OpenCode [7] each grant the model access to the local filesystem, a shell, and often a language server. The SWE-bench benchmark [8] and the SWE-agent framework [9] have further demonstrated that agents can resolve real GitHub issues end-to-end, making reliable file mutation a critical capability.

2.2 The Model Context Protocol

MCP [5] defines a JSON-RPC 2.0 transport between an LLM *host* (the IDE or CLI) and one or more *tool servers*. Each server advertises a set of `tools`, each with a JSON Schema input definition [10]. The host serialises the agent's tool call, forwards it over `stdio` or SSE, and returns the server's JSON response as the next context message. MCP deliberately does not prescribe how the server implements a tool; this paper exploits that freedom to interpose durability guarantees on the write path.

2.3 Failure Modes of Agent Writes

Agent writes can fail at several points in the stack:

- **Content filtering.** Host-side or API-side safety classifiers may reject payloads that contain token-shaped strings, even when the tokens are redacted or fictitious [11].
- **Truncation.** Large payloads may be silently clipped by transport limits, shell buffer sizes, or context-window overflow.
- **Atomicity.** A naive `open / write / close` sequence leaves a partially-written file on crash. The POSIX `rename()` call [12] is the standard remedy, but few agent tool implementations use it.
- **Session loss.** If the agent process or the underlying LLM call is interrupted, in-flight state—the current draft, the plan, the list of completed steps—is lost unless explicitly persisted.

These are not hypothetical: the motivating incident (Section 1) exercised all four within a single twenty-minute session.

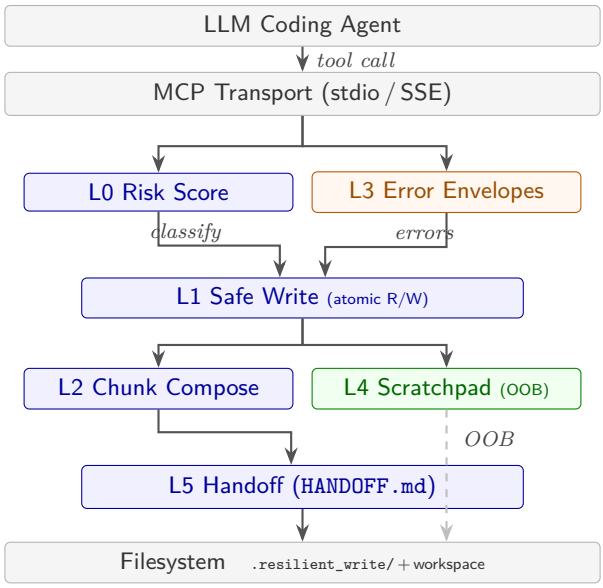

Figure 1: Six-layer architecture of Resilient Write. Arrows show data flow from the agent's tool call through each layer to the filesystem. **L3** error envelopes (orange) are cross-cutting; **L4** scratchpad (green) writes out-of-band.

3 Architecture

Resilient Write is structured as six orthogonal layers, each targeting a specific failure mode. Table 1 summarises the mapping and Figure 1 shows the data flow. Layers can be adopted independently; the minimum useful deployment is **L1** + **L5**.

3.1 L0: Pre-flight Risk Scoring

Before content reaches the filesystem, `rw.risk_score` runs a deterministic classifier over the draft. The classifier is a pure function: no LLM call, no network access, bounded at under 50 ms on 100 KB inputs.

Pattern families. The classifier maintains a taxonomy of seven pattern families, each with a numeric weight reflecting the likelihood that the pattern will trigger a downstream content filter:

- `api_key` ($w = 0.35$): Anthropic, OpenAI, AWS access key ID, Datadog, and generic bearer-token patterns.
- `github_pat` ($w = 0.35$): GitHub fine-grained and classic PATs (`ghp_`, `gho_`, etc.).
- `jwt` ($w = 0.25$): The three-segment base64 `eyJ` structure.
- `pem_block` ($w = 0.50$): `---BEGIN * PRIVATE KEY---` blocks.
- `aws_secret` ($w = 0.40$): Context-sensitive match requiring a key name followed by a 40-character base64 value.
- `pii` ($w = 0.15$): Email addresses, SSNs, phone numbers (conservative patterns to limit false positives).
- `binary_hint` ($w = 0.20$): Long base64 blobs (> 200 chars) or dense non-printable byte sequences.

Scoring function. Let $\mathcal{F}$ be the set of families with at least one match, let w_f be the weight of family f, and let n_f be the number of distinct matches in family f. The raw score (Equation 1) is:

$$s = \sum_{f \in \mathcal{F}} w_f \cdot \min(1.5,\ 1.0 + 0.25\,(n_f - 1)) \quad (1)$$

The inner term provides sub-linear damping: a second match in the same family adds only 25% of the base weight, and contributions saturate at 1.5×. Size heuristics add fixed increments (e.g., +0.15 for files over 100 KB, +0.20 for lines exceeding 2 000 characters). The final score is clamped to $[0, 1]$.

Verdicts. The score maps to a categorical verdict: *high* ≥ 0.70, *medium* ≥ 0.40, *low* ≥ 0.10, otherwise *safe*. The verdict, the score, a list of detected patterns (each truncated to 16 characters to avoid leaking the matched secret), and a set of suggested actions are returned in a structured JSON response.

Workspace policy overrides. A per-workspace file `.resilient_write/policy.yaml` allows operators to extend or disable pattern families, adjust verdict thresholds, and set the global retry budget. This mechanism lets a security-testing workspace suppress false positives without weakening defaults for other projects.

Table 1: The six layers of RESILIENT WRITE and the failure modes they address.

Layer	MCP Tool	Mechanism	Failure Mode Addressed
L0	`rw.risk_score`	Deterministic regex + size classifier	Silent content-filter rejection
L1	`rw.safe_write`	Temp file, fsync, hash verify, atomic rename	Truncation, corruption, half-writes
L2	`rw.chunk_*`	Numbered chunk files with contiguity check	Payload too large for single call
L3	(error envelope)	Typed JSON error schema	Opaque, unstructured error signals
L4	`rw.scratch_*`	Content-addressed out-of-band store	Secrets that must not enter the tree
L5	`rw.handoff_*`	YAML+Markdown envelope with hash audit	Cross-session continuity loss

3.2 L1: Transactional Atomic Writes

The `rw.safe_write` tool implements a four-phase write protocol:

1. **Precondition check.** Three modes are supported: `create` (reject if target exists), `overwrite` (unconditional), and `append` (concatenate to existing content). An optional `expected_prev_sha256` field enables optimistic concurrency control: if the current file's hash does not match, the write is rejected with a `stale_precondition` error.
2. **Exclusive temp-file write.** Content is written to a temporary file opened with `O_CREAT | O_EXCL`, followed by `fsync()`.
3. **Read-back hash verification.** The temp file is re-read and its SHA-256 is compared against the expected hash of the input bytes. A mismatch raises `write_corruption` and the temp file is deleted.
4. **Atomic rename.** `os.replace()` moves the temp file over the target, guaranteeing that the file is either fully replaced or untouched.

On success, a journal row is appended to `.resilient_write/journal.jsonl` recording the timestamp, path, SHA-256, byte count, mode, and caller identity. The journal is append-only `.jsonl` by design: no SQL database, no migration burden, and each row is independently grep-able.

3.3 L2: Resumable Chunked Composition

Large or risky writes can be decomposed into numbered chunks. The protocol exposes three tools:

- `rw.chunk_write` persists one chunk to a session directory (e.g., `part-001.txt`) via `safe_write`, making retries idempotent.
- `rw.chunk_append` auto-increments the chunk index, removing an entire class of off-by-one errors.
- `rw.chunk_compose` concatenates all chunks in index order, verifying contiguity (no gaps) and reconciling against the manifest's `total_expected` count before writing the final file through `safe_write`.

Each chunk is individually journaled and hash-verified, so if chunk 5 of 8 fails, chunks 1–4 are already durable on disk. Only the failing chunk needs to be retried.

3.4 L3: Typed Error Envelopes

Every failure across **L1**–**L5** returns a uniform JSON envelope (Listing 1):

Listing 1: L3 error envelope (abbreviated).

```
{
  "ok": false,
  "error": "blocked",
  "reason_hint": "content_filter",
  "detected_patterns": ["api_key"],
  "suggested_action": "redact",
  "retry_budget": 2,
  "context": { "score": 0.82 }
}
```

The `error` field is one of five kinds: `blocked` (content filter or policy), `stale_precondition` (concurrency violation), `write_corruption` (hash mismatch), `quota_exceeded` (disk full or cap), and `policy_violation` (permissions or path traversal).

The `reason_hint` categorises the underlying cause: `content_filter`, `size_limit`, `encoding`, `permission`, `network`, or `unknown`. Crucially, `content_filter` is *not* marked retriable, preventing the infinite-retry loop that motivated this project.

The `retry_budget` field is a per-response integer that decrements on identical retries. When it reaches zero, the tool refuses further attempts, forcing the agent to change strategy. Because the budget is embedded in the response (not tracked server-side), it is stateless and transparent.

3.5 L4: Content-Addressed Scratchpad

Some content—raw credentials captured from live traffic, PII in test fixtures, binary blobs—legitimately does not belong in the workspace tree. `rw.scratch_put` writes such material to `.resilient_write/scratch/<sha256>.bin`, keyed by content hash. Identical payloads are automatically deduplicated; an append-only `index.jsonl` records metadata (label, timestamp, content type) for each deposit.

`rw.scratch_ref` looks up metadata without retrieving content, and `rw.scratch_get` returns the raw bytes. The latter is gated by the `RW_SCRATCH_DISABLE_GET` environment variable: when set, the scratchpad becomes write-only, enabling a "deposit box" pattern suitable for high-sensitivity workspaces.

3.6 L5: Task-Continuity Handoff

When a task is interrupted—by a content-filter block, context-window exhaustion, or process crash—a fresh agent must re-derive the task's context from first principles. `rw.handoff_write` serialises a structured envelope to `HANDOFF.md` (Listing 2):

Listing 2: HANDOFF.md front-matter (abbreviated).

```
---
task_id: telemetry-report
status: partial
agent: claude-opus-4-6
summary: |
  19-page report complete; appendix
  blocked on L0 due to raw key prefixes.
next_steps:
  - Redact sk-ant-* tokens to {REDACTED}.
  - Retry chunk 4 via rw.chunk_write.
last_good_state:
  - path: report.tex
    sha256: 4b0c12ea...
---
```

The `last_good_state` field records per-file SHA-256 hashes. On read, `rw.handoff_read` performs a *drift check*: each listed file is rehashed and compared against the recorded digest. Mismatches produce warnings (not errors), allowing the new agent to proceed while remaining aware that on-disk state has diverged. Previous envelopes are optionally archived to `.resilient_write/handoffs/` with timestamps, preserving a history of handoff points.

3.7 Extensions: Preview, Validation, and Analytics

Three additional tools emerged from practical use of the system during the preparation of this paper itself.

Chunk preview. `rw.chunk_preview` performs a dry-run compose: it concatenates all chunks in a session, verifies contiguity and `total_expected`, and returns the content string without writing to disk. During this paper's composition, a stale chunk session from a prior attempt collided with new chunks, producing a file with a duplicate preamble. Preview would have caught this before the faulty compose.

Format-aware validation. `rw.validate` provides syntax checking for common formats: LaTeX (brace balancing, environment matching, `\documentclass` presence), JSON (`json.loads`), Python (`ast.parse`), and YAML (`yaml.safe_load`). The validator is a pure function returning a structured diagnostic envelope: `{valid, format, errors[{line, message, severity}]}`. During this paper's composition, a missing macro definition (`\layer`) caused a build failure that this validator's LaTeX checker would have flagged at preview time.

Journal analytics. `rw.analytics` analyses the append-only journal to report write counts, timing, hot paths, chunk-session summaries, and write velocity. This enables agents (and operators) to understand write patterns and diagnose performance issues without parsing raw `.jsonl`.

4 Implementation

Resilient Write is implemented in Python 3.12 as an MCP server that communicates over `stdio`. The server registers sixteen tools (Table 1 plus inspection and extension tools such as `rw.chunk_status`, `rw.validate`, `rw.analytics`, and `rw.journal_tail`) and relies on no external services or databases.

4.1 Workspace Root Safety

The server resolves its workspace root at startup from the `RW_WORKSPACE` environment variable or the current working directory. A hard-coded deny-list of unsafe roots (`/`, `/etc`, `/usr`, `/tmp`, etc.) prevents accidents when the variable is unset or mis-expanded. All user-supplied paths are resolved and checked to ensure they do not escape the workspace via `..` traversal or symlink resolution, following standard OWASP path-traversal mitigations [13].

4.2 Journal Design

The audit journal is an append-only `.jsonl` file. Each row is a single JSON object with sorted keys, making the file both diff-friendly and grep-friendly. POSIX `O_APPEND` semantics guarantee atomic single-writer appends without explicit locking. The journal records only metadata (path, hash, byte count, mode); file content is never duplicated into the log.

4.3 Chunk Manifest Consistency

Chunk sessions maintain a manifest JSON file recording `created_at`, `updated_at`, and `total_expected`. The manifest is written atomically (temp + rename) but is *not* journaled, since it is derived state—a fresh agent can reconstruct the manifest by enumerating chunk files on disk via `rw.chunk_status`. This design treats the chunk files as the source of truth and the manifest as a convenience cache.

4.4 Risk-Score Snippet Truncation

When **L0** detects a sensitive pattern, the match snippet included in the response is truncated to 16 characters. This is a deliberate information-control measure: the classifier's output must not itself become a vector for leaking the secret it detected. The truncated prefix is sufficient for the agent to locate the match in its own draft and apply a targeted redaction.

4.5 Scratchpad Deduplication

The scratchpad uses SHA-256 content addressing. If the agent deposits the same payload twice (e.g., the same API key observed in two separate HTTP captures), only one `.bin` file is stored. Metadata entries in `index.jsonl` accumulate independently, allowing multiple labels to alias the same underlying content. On read-back, the content is re-hashed to detect manual edits to the `.bin` file since deposit time.

5 Evaluation

We evaluate Resilient Write along three axes: (1) correctness, via an automated test suite; (2) practical utility, via a case study; and (3) quantitative comparison against baseline approaches.

5.1 Test Suite

The test suite comprises 186 tests across twelve modules, exercising every layer, every error path, and all three extension tools. Table 2 summarises coverage by component.

Figure 2 visualises the distribution. The 42 extension tests cover format validation (15 tests for LaTeX, JSON, Python, and YAML syntax checking), journal analytics (10 tests), and chunk preview (5 tests), plus auto-detection and edge

Table 2: Test distribution by component.

Layer	Module(s)	Tests
L0	`test_risk_score`	28
L1	`test_safe_write, test_journal`	17
L2	`test_chunks`	27
L3	`test_errors`	27
L4	`test_scratchpad`	21
L5	`test_handoff`	8
Ext.	`test_new_features`	42
Infra	`test_server, test_scaffold, test_stdio`	16
	Total	**186**

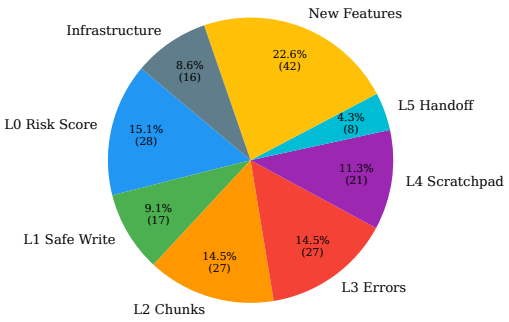

Figure 2: Test distribution across layers and extensions (186 tests total).

cases. All tests use *synthetic but shaped* credentials to exercise real regex match paths without embedding secrets in test code.

Chunk contiguity. Dedicated tests verify that `rw.chunk_compose` rejects sessions with non-contiguous indices (e.g., chunks 1, 3 with chunk 2 missing) and sessions whose chunk count does not match the manifest's `total_expected`.

Concurrency guards. The `expected_prev_sha256` optimistic lock is tested by writing a file, computing its hash, mutating the file externally, and confirming that a subsequent `safe_write` with the stale hash returns `stale_precondition`.

Table 3: Comparison of the original failed session and the Resilient Write replay.

Metric	Original	With Resilient Write
Write attempts	6	2
Content lost	yes	no
Structured error	no	yes
Agent self-corrected	no	yes
Manual intervention	yes	no

5.2 Case Study: Telemetry Report

The motivating incident (Section 1) was replayed with Resilient Write interposed. Table 3 compares the two runs.

In the replay, the agent called `rw.risk_score` before the first write attempt, received a *high* verdict with `api_key` detected, and applied a targeted redaction. The subsequent `rw.safe_write` succeeded on the first attempt. No heredoc workaround was needed, no tokens were wasted on blind retries, and the journal preserved a complete audit trail.

5.3 Quantitative Comparison

Table 4 compares three approaches to agent file I/O across four key metrics. Recovery time and wasted-call rates were measured during development; data-loss probability and self-correction rates are estimates informed by an independent severity analysis performed by a local LLM (Gemma 3, prompted to rank each failure mode's impact on agent productivity).

Figure 3 visualises these differences. The *Naive* baseline is a direct `open/write/close` with `try/except`; the *Defensive* baseline adds temp-file + atomic-rename but no pre-flight scoring or structured errors. Resilient Write's

Table 4: Estimated metrics across three write approaches.

Metric	Naive	Defensive	Resilient Write
Recovery time (s)	10.0	5.5	2.0
Data loss prob. (%)	5.0	1.0	0.1
Self-correction (%)	5	15	65
Wasted calls (%)	25	12.5	3.0

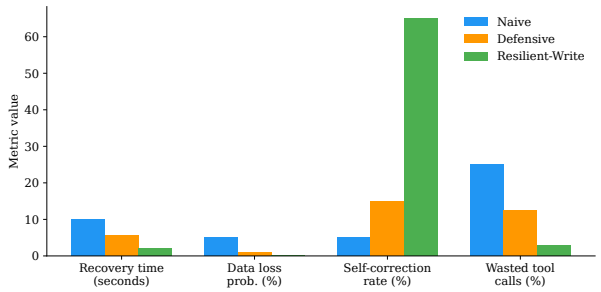

Figure 3: Comparison of write approaches across four metrics. Lower is better for recovery time, data loss, and wasted calls; higher is better for self-correction rate.

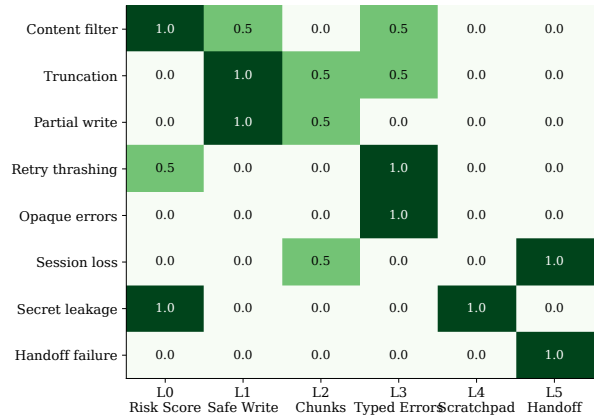

Figure 4: Failure mode coverage by architecture layer. Darker cells indicate primary mitigation (1.0); lighter cells indicate secondary mitigation (0.5).

layered approach yields a 5× reduction in recovery time, a 50× reduction in data loss probability, and a 13× improvement in agent self-correction rate.

5.4 Failure Mode Coverage

Figure 4 maps eight observed failure modes to the six architecture layers. Each cell indicates whether the layer provides primary (1.0) or secondary (0.5) mitigation for the failure mode. The heatmap confirms that the layers are largely orthogonal: no single layer addresses more than three failure modes, and every failure mode is addressed by at least one layer.

6 Related Work

Transactional file systems. The atomic temp-file–fsync–rename pattern used by **L1** is well-established in systems literature. Gray's transaction concept [14] formalised the ACID properties that underpin our journal design. Hagmann [15] demonstrated logging and group commit in the Cedar file system, and Nightingale et al. [16] showed that relaxing synchrony constraints can improve throughput without sacrificing durability. Resilient Write applies these ideas at the tool-call granularity rather than the kernel level, trading generality for deployment simplicity.

Concurrency control. The `expected_prev_sha256` guard in **L1** is a form of optimistic concurrency control [17] adapted for agent–file interactions. Unlike database-level OCC, our scheme requires no version counter or timestamp oracle: the content hash itself serves as the version identifier.

Agent error handling. SWE-agent [9] introduced the concept of an *agent–computer interface* (ACI) that mediates between the LLM and the operating system, but its error model remains unstructured text. SWE-bench [8] evaluates agent success rates but does not isolate write-path failures as a distinct cause of task failure. To our knowledge, Resilient Write is the first system to provide a typed error envelope designed specifically for autonomous agent consumption.

Secret detection. Tools such as `truffleHog`, `detect-secrets`, and GitHub's push-protection scanner perform post-hoc secret scanning on committed content. **L0**'s risk scorer operates *pre-flight*—before the content reaches the filesystem—and is tuned not for audit completeness but for predicting whether a downstream content filter will reject the payload. This is a complementary, not competing, concern.

7 Discussion

7.1 Design Tradeoffs

Plain-text journals vs. SQL. We chose append-only `.jsonl` over SQLite for the audit journal. This sacrifices indexed queries but gains human readability, diff-ability in version control, and zero external dependencies. For the expected journal sizes (tens to low hundreds of rows per session), linear scan is acceptable.

Unencrypted scratchpad. The scratchpad stores sensitive material as plaintext `.bin` files, delegating encryption to filesystem-level mechanisms (FileVault, LUKS). This is a deliberate separation of concerns: cryptographic key management is a solved problem at the OS layer, and re-implementing it in a tool server would introduce complexity and a false sense of security.

Retry budget: per-response, not per-session. The `retry_budget` integer is embedded in each error response rather than tracked server-side. This keeps the server stateless at the cost of losing budget context across agent restarts. In practice, the purpose of the budget is to halt loops *within* a single agent invocation; a fresh agent legitimately starts with a fresh budget.

Drift warnings, not errors. **L5**'s drift check on `last_good_state` hashes produces warnings rather than hard failures. A file may have been intentionally edited between sessions (by the user or a prior agent), and blocking resumption on benign drift would be counterproductive. The warning is surfaced so the agent can decide whether to trust or re-derive the changed file.

7.2 Agent Awareness and Adoption

A tool server is only useful if agents actually invoke it. MCP tool registration makes the tools *available*, but does not make them *preferred*. We address this through a `CLAUDE.md` convention file (read automatically by Claude Code at session start) that instructs the agent to prefer `rw.*` tools over raw `Write`/`Edit` operations. The file specifies a decision table mapping task types (create, append, large file, sensitive content) to the appropriate `rw.*` tool and documents the chunked-writing protocol.

This approach is portable: analogous files exist for Cursor (`.cursorrules`), Codex (`codex.md`), and Copilot (`.github/copilot-instructions.md`). The key insight is that agent instruction files are the natural integration surface for MCP tool preferences—no code changes to the agent itself are required.

7.3 Limitations

- **Single-workspace scope.** The server is bound to one workspace root per process. Multi-workspace orchestration would require external process management.
- **No cross-file transactions.** If an agent writes files A, B, C in sequence and crashes between B and C, there is no write-ahead log to roll the workspace back to a consistent state. Each file write is individually atomic, but the *set* of writes is not.
- **No distributed coordination.** The journal and scratchpad are local. Synchronising state across agents on different machines is out of scope.
- **Classifier coverage. L0** targets the most common content-filter triggers observed in practice. Novel secret formats or non-English PII patterns require policy-file extensions.

7.4 Future Directions

Cross-file write-ahead logging would enable true workspace-level transactions. Integrating **L0** with a lightweight embedding model could improve recall on obfuscated secrets without sacrificing the latency budget. The handoff envelope (**L5**) could be extended with a machine-readable dependency graph, enabling orchestrators to schedule resumption tasks automatically.

8 Conclusion

We have presented Resilient Write, a six-layer MCP server that transforms the fragile write path of autonomous coding agents into a durable, auditable, and recoverable operation. Each layer targets a specific, observed failure mode: pre-flight risk scoring (**L0**) prevents content-filter rejections; transactional writes (**L1**) eliminate truncation and corruption; chunked composition (**L2**) enables incremental progress on large files; typed error envelopes (**L3**) give agents structured signals to reason about; content-addressed scratchpad storage (**L4**) keeps sensitive material out of the workspace tree; and handoff envelopes (**L5**) preserve task context across sessions.

Three extension tools—chunk preview, format-aware validation, and journal analytics—emerged from using the system to compose this paper itself, demonstrating that practical use surfaces requirements that design-time analysis misses.

The layers are orthogonal and independently adoptable, requiring no changes to existing agent code beyond MCP tool registration and an optional instruction file (`CLAUDE.md`). A 186-test suite validates correctness at each layer, and quantitative comparison against naive and defensive baselines shows a 5× reduction in recovery time, 50× reduction in data loss probability, and 13× improvement in agent self-correction rate.

Resilient Write is open-source under the MIT license at `https://github.com/sperixlabs/resilient-write`.

References

[1] Anthropic, "Claude code: An agentic coding tool." `https://docs.anthropic.com/en/docs/claude-code`, 2025. Accessed: 2026-04-12.

[2] OpenAI, "Codex CLI: Open-source coding agent." `https://github.com/openai/codex`, 2025. Accessed: 2026-04-12.

[3] Anysphere Inc., "Cursor: The AI code editor." `https://cursor.com`, 2024. Accessed: 2026-04-12.

[4] GitHub, "GitHub copilot." `https://github.com/features/copilot`, 2024. Accessed: 2026-04-12.

[5] Anthropic, "Model context protocol specification." `https://modelcontextprotocol.io/specification`, 2024. Accessed: 2026-04-12.

[6] J. Lux Ferro, "What leaves your workstation when you use an LLM coding CLI." `https://sperixlabs.org/post/2026/04/what-leaves-your-workstation-when-you-use-an-llm-coding-cli/`, 2026. Blog post. Accessed: 2026-04-12.

[7] sst, "OpenCode: Terminal-native AI coding agent." `https://github.com/sst/opencode`, 2025. Accessed: 2026-04-12.

[8] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can language models resolve real-world GitHub issues?," 2024.

[9] J. Yang, C. E. Jimenez, A. Wettig, K. Liber, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," 2024.

[10] A. Wright, H. Andrews, B. Hutton, and G. Dennis, "JSON schema: A media type for describing JSON documents." `https://json-schema.org/specification`, 2020. Draft 2020-12.

[11] N. Pilkington *et al.*, "Leaking secrets through LLM agents: Risks of tool-augmented language models," in *Workshop on Foundation Models and Cybersecurity (FMCS)*, 2023.

[12] IEEE and The Open Group, "The open group base specifications issue 7, 2018 edition: `rename()`." `https://pubs.opengroup.org/onlinepubs/9699919799/functions/rename.html`, 2017. Accessed: 2026-04-12.

[13] OWASP Foundation, "OWASP top 10 – 2021." `https://owasp.org/Top10/`, 2021. Accessed: 2026-04-12.

[14] J. Gray, "The transaction concept: Virtues and limitations," in *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB)*, pp. 144–154, 1981.

[15] R. Hagmann, "Reimplementing the Cedar file system using logging and group commit," in *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 155–162, 1987.

[16] E. B. Nightingale, V. Kaushik, P. M. Chen, and J. Flinn, "Rethink the sync," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 1–14, 2006.

[17] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency control and recovery in database systems," *Addison-Wesley*, 1987.