

A Restart Is Not Permission: When an Agent Gate Forgets “No”

Cenab Batu Bora
Georgia Tech
cenab@gatech.edu

With
Apart Research

Abstract

SoundGate permits or refuses tool calls. Rejections and earlier permissions must block retries with identical run and action identifiers. The tested version can confirm recording a decision appended to an unfinished log entry, then skip the combined invalid line after restart, omitting the decision from memory. Our recovery barrier repairs only unconfirmed log endings before serving requests. Given intact confirmed records and specified storage behavior, our written argument establishes restored restrictions or startup refusal across restarts. Across 153 constructed sequences of requests and restarts per policy, the barrier enforced restrictions while unrelated work executed; original recovery produced 143 prohibited tool executions. Strict refusal rejected 147 startups for invalid records or a missing final newline. The barrier restored restrictions after 30 forced exits during recovery. The test tool writes a separate receipt file; matching receipts to invocation records establishes execution. These diagnostics establish neither deployment failure rates nor exactly-once tool execution.

September 11, 2026

Code and evidence are attached to this PDF; extraction and judge instructions are in Appendix B.

1. Introduction

A restart should not undo a recorded rejection. An *agent gate* checks whether a proposed tool call may execute; SoundGate, the implementation we study, is intended to retain decisions across requests [6, 7]. Yet in the tested version, a rejection can disappear from reconstructed state while its recorded bytes survive. The failure begins with an unfinished record in the decision log, the file used to restore decisions: startup leaves it in place, the gate appends and acknowledges a new decision without first adding a separator, and the next startup skips the combined invalid line.

In the tool-use loop considered here, the model receives a task description and interaction history, then proposes a tool call. A designer places the gate between proposal and execution, where it can grant permission (a *release*), refuse the call, or hold it for a person’s approval. The runtime executes permitted calls and returns their results to the model, which can propose another call or produce a response. The gate constrains only calls that pass through it.

Consider an illustrative email assistant. A person rejects a proposed message, and the gate confirms that the decision is recorded. An immediate retry with the same identifiers for the run and message is refused; after restart, the gate should still refuse requests using that pair. Likewise, a prior release should block another permission for the same pair, regardless of whether the email was sent.

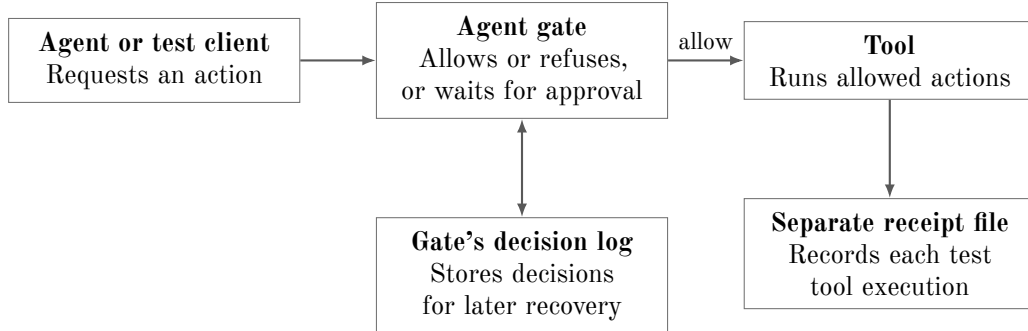


Figure 1: A gate checks permission before an action reaches a tool. Our tests use a scripted client and a harmless tool that writes a receipt. The gate’s decision log and the tool’s receipt file serve different purposes: one records decisions, the other provides evidence of execution.

This Track 1 study for the incident-response sprint examines recovery: rebuilding state after restart. Hugging Face reports an agent downloading its tools again for each temporary computing environment; responders later blocked its access to the internal network [5]. That report motivates checking restrictions when activity resumes after interruption; it establishes neither SoundGate’s involvement nor a causal connection to this failure.

We inspect SoundGate package version 0.1.3 at revision 6568403. Under a storage model that retains earlier acknowledged records, the recovery barrier removes an incomplete, unacknowledged final fragment or supplies a complete event’s missing newline before accepting requests. We compare recovery policies using constructed histories of initial log bytes, requests, and restarts; tool invocation records and separate receipts establish execution.

Across 153 histories per policy, original recovery produces 143 prohibited effects; the barrier produces none and permits unrelated allowed work in every history. Strict refusal declines startup on malformed or unterminated records and also avoids prohibited effects. It refuses the first startup in 147 histories, so the client cannot obtain acknowledgement of a new decision (Table 1).

We implement the barrier inside the service and give a written argument that, on each modeled restart under explicit storage assumptions, the service restores acknowledged restrictions before serving or refuses startup. The policy comparison covers every byte prefix of each of three emitted log records, with separate tests of interrupted repairs and injected storage errors. The attached runnable code and evidence distinguish failed restrictions, unavailable service, and unresolved claims about execution.

2. Methods

2.1 What the gate remembers

SoundGate associates an action with two identifiers. The `run_id` identifies an agent run, and the `effect_key` identifies a logical action within that run. In the email example, these might identify the assistant’s current task and one proposed message. A retry with the same pair refers to the same action. A *rejection* blocks that action identity; a *cancellation* blocks later submissions for the entire run, regardless of action identity. In our tests, the client reuses this pair for retries and submits unrelated fresh work under another pair in a run that has not been cancelled. We do not test whether the gate recognizes equivalent actions submitted under different identifiers.

A *release* permits an action; it does not confirm tool execution. *Deduplication* remembers that permission and refuses another request with the same identity, even if no tool execution followed the first release. We therefore measure tool execution separately.

The running process keeps decisions in memory. To preserve them across a restart, it also writes events to a *write-ahead log* (WAL): a file recording decisions before the gate sends an *acknowledgement* (ACK) confirming that they were recorded. For a rejection, this is the response to the rejection decision, distinct from refusing a later retry. In the tested implementation, the writer appends the event with `write_all`, calls `sync_data` to ask the operating system to make the data durable (retained across a crash), and then acknowledges it [7]. Recovery reads the file and rebuilds the in-memory decisions.

2.2 How a saved decision can disappear from recovered state

The log contains one JSON event per line, with newline bytes providing *framing*: the boundaries that separate records. Original recovery can parse a complete final event without its newline, but the next append then joins two JSON objects into an invalid line. Preserving bytes alone does not preserve readable records.

For text that decodes as UTF-8, original recovery skips an invalid last line but refuses startup if another record follows an invalid line. Skipping permits recovery from an incomplete final write, yet leaves the fragment on disk for the next append to join. The first recovery can therefore restore earlier decisions without leaving a file that will preserve the next one.

Return to the illustrative email, with earlier acknowledged records intact and an unfinished, unacknowledged record at the end of the log:

1. On the first recovery, the gate ignores the unfinished record and starts normally, leaving the fragment on disk.
2. A person rejects the proposed email. The gate appends the rejection after the fragment, synchronizes the file, and confirms that the decision was recorded. An immediate retry with the same identifier pair is refused.
3. On the second recovery, the fragment and rejection appear as one malformed final line. Recovery skips it, so the reconstructed state no longer blocks that email request.

The rejection was enforced before the second restart, but its surviving bytes no longer restore the restriction. A newly acknowledged release can be lost in the same way, allowing another permission for its identity.

For a compact description, let P be the earlier record bytes including their newlines, t the unfinished fragment, a the new event bytes without their newline, and $\parallel$ byte concatenation. Write $\langle \text{LF} \rangle$ for a newline. The file changes from $P \parallel t$ to $P \parallel t \parallel a \parallel \langle \text{LF} \rangle$. The second recovery skips the joined record $t \parallel a$.

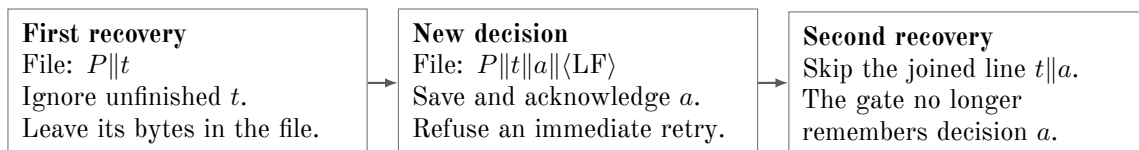


Figure 2: The failure concerns how records are read after restart. The acknowledged event a remains on disk, but the reader treats it as part of an invalid final line.

If the gate records event b after a and acknowledges it to the client before the second restart, the file ends in $P \parallel t \parallel a \parallel \langle \text{LF} \rangle \parallel b \parallel \langle \text{LF} \rangle$. The malformed line now precedes a valid record, so the same reader should refuse startup. An earlier comparison using nine histories per policy included this sequence. Appendix E retains the full design and results.

When the file ends partway through a UTF-8 character, decoding fails before JSON parsing; Section 3.2 explains these separate startup refusals.

2.3 Proposed method: recover before serving

Repair must finish before the gate can append and acknowledge another decision. We enforce this order inside the service, before it opens the socket that accepts requests. The *recovery barrier* has five steps:

1. Open the existing log and acquire an exclusive lock. Keep that same open file and lock until shutdown. A missing log causes refusal; it must not silently become an empty history.
2. Validate every newline-terminated record with the service’s event parser, refusing startup if any is invalid, including the final record.
3. *Normalize* the unterminated suffix—the bytes after the last newline—so the next append starts at a record boundary. Retain a complete event and add its newline, or discard an incomplete fragment under the storage model.
4. Synchronize the file, including its length, even when this launch changed nothing: an earlier repair may have stopped before making its changes durable. Then *replay* the retained events: read them in order to rebuild the in-memory sets of released identities, rejected identities, and cancelled runs.
5. Open the listener. For decisions that must persist, append their records and synchronize the log before replying. The writer batches log events into groups written together; if a write or synchronization fails, terminate the service before it appends a subsequent batch. Stopping keeps any incomplete write at the log’s end for the next recovery to repair and synchronize.

We implement these steps in the pinned Rust TCP service. The rules for allowing or refusing requests remain unchanged. The lock prevents another cooperating instance from repairing or appending to the same file concurrently. The prototype refuses startup for logs above 8 MiB; supporting larger logs would require a change, such as reading and processing their events incrementally.

Assume that acknowledged records remain intact, the log’s path is stable, every writer respects the lock, and storage honors the specified synchronization behavior and allowed file states after interrupted repair. Then, after any finite number of modeled restarts, acknowledged rejection, cancellation, and release restrictions are restored before new requests are served, or the service does not accept requests. Within the size bound, a sufficiently long failure-free interval permits recovery and fresh work once the lock is available and storage calls succeed. Appendix C specifies those storage requirements, gives the argument, and lists the interruption-test checkpoints.

Recovery itself can be interrupted, so normalization must be safe to repeat. The invariant required before appending each new event record is that the file ends at a valid record boundary. Repeating a successful normalization changes no further bytes, a property called *idempotence*. If N denotes normalization, then $N(N(W)) = N(W)$ for any log W for which normalization succeeds. Normalization preserves the acknowledged prefix, which grows after the service appends a newline-terminated event, synchronizes the log, and acknowledges the event. Under the storage model, an interrupted repair also preserves that prefix and leaves a suffix the next launch can normalize. Induction over these steps gives the repeated-restart result only for failures allowed by that model. This argument has not been machine-checked.

2.4 Experiment: restrictions and useful work

The email assistant is illustrative; our experiment uses a scripted client and a harmless tool that writes a receipt to a separate file. No email or model call is needed.

We compare the unmodified service with strict refusal and the recovery barrier implemented inside the service. Strict refusal requires every record to parse and end with a newline; it never repairs the file. The variants share the rules for allowing or refusing requests, request sequence, acknowledged

starting prefix, and allowed tool. They differ in startup recovery and writer failure handling. After a later append or synchronization error, the original service reports the error and keeps its writer running; strict refusal and the barrier terminate the service. We construct only the unacknowledged suffix while the service is stopped.

After a successful first recovery, the client obtains acknowledgement of the rejection decision, cancellation, or initial release, then checks an immediate retry. After restart, the client resubmits the disputed action with the same identifier pair before other log writes, probes older restrictions, and requests unrelated allowed work under another identity. Retrying the disputed action first tests the recovered restriction before new requests alter the gate’s state or log.

After each release, the test tool (the receiver) writes a receipt to its separate file and synchronizes that file. The receiver’s process invocation and before/after file contents establish whether permission produced a persisted effect. A duplicate requires two distinct invocations for the same identity. The receiver has no duplicate filter that could hide a gate failure. If a claimed execution lacks its invocation record, that execution claim remains unresolved.

Availability is separate: a gate that refuses startup cannot process the later retry. The fresh-action check prevents permanent shutdown from counting as successful recovery.

The byte-prefix sweep tests recovery inside the service and extends a policy comparison developed in a 15-case pilot and used in one fixed 27-execution confirmation. For the earlier confirmation’s nine histories per policy, a Python wrapper outside the service inspected the log before each launch; under its tail-repair policy, it normalized the suffix before starting the service. The earlier comparison also tested an extra acknowledged record after the disputed decision, making original recovery refuse startup when the preceding malformed line was no longer last. The earlier automated review assessed that comparison’s execution evidence; its approval does not cover the extension, which uses cases considered during development rather than cases reserved for confirmation.

We record real cancellation, rejection, and release events, of lengths 38, 54, and 58 bytes, respectively. For each record, we test every prefix from zero bytes through the complete record. Each test log contains a fresh copy of the acknowledged starting prefix followed by the selected record bytes as an unacknowledged suffix. After recovery, the client attempts to obtain acknowledgement of a new decision of the same event type, using a different run identifier and, for rejection or release, a different action identifier from the calibration record. This gives $(38 + 1) + (54 + 1) + (58 + 1) = 153$ histories per policy. Two records contain identifiers with multibyte UTF-8 characters, so some cuts stop within a character’s encoding.

Separate interruption tests pair each of five startup checkpoints with each of the two repair actions, giving ten histories. In each history, the service version modified for testing exits at the selected checkpoint on each of three consecutive launches. Separate controls check that missing logs and lock conflicts prevent startup, and that storage errors cause startup refusal or termination before another log write. These are local process terminations and constructed file states, not electrical power failures.

3. Results

3.1 Native recovery across every byte cut

The byte sweep completed 459 policy executions. With the barrier, restrictions held and the test tool executed unrelated allowed work in all 153 histories. Original recovery produced 143 prohibited effects. Strict refusal produced none, while making the service unavailable in 147 histories before the new acknowledgement (Table 1). The rebuilt original binary matched the binary used in our earlier confirmation study byte for byte, so both comparisons used the same original executable.

Table 1: All policies face the same 153 histories, each with a chosen record prefix and fixed requests and restarts. Together they cover every byte prefix of each of three emitted records, including empty and complete prefixes. Held counts histories in which the disputed post-restart retry was refused and complete request, reply, invocation, and receipt-file records showed no prohibited execution. Counting a prohibited effect requires a tool invocation record and durable receipt. Refusal counts startup failures before the new decision could be acknowledged. Fresh work counts histories in which the test tool executed an unrelated allowed action. No startup failed after the new acknowledgement, and no outcome remained unresolved because of missing evidence.

Policy	Cases	Held	Prohibited effects	Refusal	Fresh work
Original	153	6	143	4	149
Strict refusal	153	6	0	147	6
Recovery barrier	153	153	0	0	153

3.2 Why original recovery refuses startup in four histories

The four original-policy refusals arise from incomplete UTF-8, before JSON parsing. The identifiers in the sampled cancellation and rejection records contain the character with Unicode value U+96EA, encoded by the three bytes E9 9B AA. Retaining 33 or 34 bytes of the cancellation record, or 32 or 33 bytes of the rejection record, stops after the first or second byte of that character’s encoding. The original line reader reports that the stream does not contain valid UTF-8, and the service process exits with status 1 before the new decision can be acknowledged. These are observed startup failures, not missing cases or demonstrated refusals of post-restart retries.

The barrier locates newline boundaries in the raw bytes, then validates the terminated records separately from the final suffix. It can therefore remove an incomplete, unacknowledged suffix without first decoding the entire file as text.

The six histories in which original recovery enforced the restriction used the empty or complete prefix of each of the three records. Thus all 153 original-policy cases are accounted for: six histories in which restrictions held, 143 prohibited effects, and four early refusals. Appendix B gives an offline command that identifies the four startup failures from their saved file bytes and process errors.

3.3 Interruptions and storage errors

All 30 forced exits occurred before a listener opened; the acknowledged prefix remained intact after each exit. In each of the 10 interrupted histories, the service subsequently recovered and enforced the restriction, and the test tool executed fresh work. All 10 boundary and error controls passed. Under injected storage failures, the service returned no successful release; invalid newline-terminated records, missing logs, and lock conflicts caused the service to refuse startup. If a release record survives an interruption, a retry may be refused after restart even though the tool never executed the action. This exposes the boundary of the guarantee: the gate preserves permission history but cannot ensure that every granted action reaches its tool.

The sweep enumerates every byte prefix of each of three emitted records, not all possible logs or deployments. It includes cases considered during development, and many prefixes are derived from the same emitted record. These cases provide regression tests: checks that later code changes do not reintroduce these failures. They are not independent observations from which to estimate a failure probability. The artifact’s `recovery-barrier/RUNS.md` records a failed build setup and a collection attempt halted because the receiver rejected a non-ASCII release identity; records from all completed executions are retained.

3.4 Fresh reproduction from the delivery bundle

We extracted the previously delivered ZIP into a new directory and rebuilt four binaries: original recovery, strict refusal, the barrier, and the barrier instrumented for forced exits and injected errors. We reran the byte sweep, interrupted-recovery histories, and boundary/error controls. The binary hashes matched those in the archived build record, and every policy count matched the earlier extension run; the interrupted-recovery tests and boundary/error controls also passed. Collection took 81.5 seconds after compilation and used no paid API calls. The raw experimental records from this rerun and the separate request/receipt checker outputs are attached. This was a same-machine reproduction by the project, not an outside replication or an additional independent sample. An unfamiliar engineer can follow the same commands in Appendix B.

4. Related Work

Amodei et al. [1] frame AI-safety concerns as concrete research problems. Here, when recovery leaves an unfinished final fragment in place, a fixed sequence of ordinary requests and restarts can reveal that an acknowledged restriction was lost. No model reasoning error is required, and the scripted client need not discover the failure mechanism.

Our closest comparison is SoundGate itself [6]. Its paper checks the admission core (the rules for allowing requests) and separately argues that recovery never forgets acknowledged decisions (Lemma 1). That group-commit argument covers batching log events, synchronizing them, then replying. Our counterexample identifies a requirement at the write/recovery boundary: preserved acknowledged bytes must remain readable as separate records. The failure concerns the reconstructed decisions supplied to unchanged admission rules, so it does not contradict the properties established for those rules in isolation. The barrier repairs record boundaries before serving requests.

SoundGate’s inspected S5 test records a release and a cancellation, kills the process, and restarts the service. It then checks that a retry of the released action and a submission to the cancelled run are refused, while a fresh request receives permission [7]. Our added damaged ending, new acknowledgement after recovery, and second recovery expose the missing record boundary; receiver records tied to tool invocations also establish whether permission led to execution. Passing that single-restart test therefore does not settle the behavior tested here.

CrashMonkey and ACE search for persistence bugs in short operation sequences interrupted by crashes [9]; related work studies application storage assumptions and recovery proofs [2, 10]. We use established write-ahead logging, locking, synchronization, and idempotent repair techniques to preserve acknowledged restrictions across restarts in the modified SoundGate service under the stated storage assumptions. We test whether acknowledged restrictions survive recovery and unrelated fresh work can execute afterward.

The incident-analysis work of Ezell et al. [4] identifies records investigators need, while CoSAI’s response framework includes recovery [3]. Our checklist specifies the records needed to assess whether acknowledged gate restrictions survive restart. We apply the principle that gaps in observations may require withholding a conclusion [8]: a receiver record unconnected to an invocation cannot establish that an action happened twice.

5. Discussion and Limitations

5.1 What a defender should check

On an isolated system, create an incomplete, unacknowledged final log fragment without changing any acknowledged record. Recover, obtain the gate’s acknowledgement of a rejection, and verify that an immediate retry is refused. Restart again, verify that a request with the same identifier pair is refused, then test execution of unrelated allowed work. The checklist in Appendix D specifies the records for this check, which a maintainer can adapt into a regression test for recovery-code changes.

An outside reviewer needs requests, acknowledgements, recovery inputs, receiver observations, and their collection procedure. Comparing file hashes can detect changes, but cannot establish that the original observations were truthful or complete. Our analysis trusts the scripts that collect these observations, the operating system, and the receiver. It does not establish the security of a lab’s network.

A reviewer must also check which communication paths a gate mediates. Reports of agents sharing task information through wiki pages [12] motivate that check; the gate cannot constrain actions that bypass it.

5.2 What remains to be tested

We establish a counterexample and test a native correction for one fixed source revision. We do not establish the same failure in current upstream software, its frequency during real operation, or a connection to the public incidents. The scripted client sends these requests without a language model; the experiment does not estimate how often a model would propose them.

The barrier requires an intact acknowledged prefix and the same log file at the same path; the file and its directory must be protected from unauthorized changes. We tested startup refusal while another cooperating process held the log lock, but have not tested overlapping requests from multiple clients, distributed gates, hostile storage, or power loss during a file-system operation. An older copy of the log may still parse correctly while lacking later acknowledged decisions, so syntax alone cannot reveal that history loss. The guarantee also excludes pending approval state, which the inspected service does not durably record.

Granting permission does not ensure that the tool executes the action exactly once. Exactly-once execution requires the receiver to persist a stable operation identity atomically with its effect (both persist together or neither does) and check that identity on retry to avoid repeating the effect, or use an equivalent protocol. A cancellation cannot recall permission already handed to another process. Follow-up work should test receiver retries and cancellation after release, concurrent client requests, and failures during file-system or device operations.

Practitioner adoption was not measured, and no human usability study was conducted.

6. Conclusion

In the pinned SoundGate service, a recorded rejection can stop constraining requests even while its bytes survive. The recovery barrier restores an interpretable decision history before serving; the service terminates if a later log write or synchronization fails. With the native implementation, restrictions held and unrelated allowed work executed across the tested byte cuts and interrupted recoveries; the written argument extends preservation of acknowledged restrictions to repeated restarts under the stated storage model. Remembering permission still cannot guarantee tool execution. The attached artifact supplies the method, argument, and runnable tests (Appendices B, C and E).

References

- [1] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. arXiv:1606.06565v2, 2016. URL <https://arxiv.org/abs/1606.06565v2>.
- [2] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. ACM, 2015. URL <https://adam.chlipala.net/papers/FscqSOSP15/>.
- [3] Coalition for Secure AI. Defending AI systems: A new framework for incident response in the age of intelligent technology, October 2025. URL <https://www.coalitionforsecureai.org/defending-ai-systems-a-new-framework-for-incident-response-in-the-age-of-intelligent-technology/>. Framework v1.0; published October 30; accessed September 11, 2026.
- [4] Carson Ezell, Xavier Roberts-Gaal, and Alan Chan. Incident analysis for AI agents. arXiv:2508.14231, 2025. URL <https://arxiv.org/abs/2508.14231>.
- [5] Hugging Face. Anatomy of a frontier lab agent intrusion: A technical timeline of the July 2026 incident, July 2026. URL <https://huggingface.co/blog/agent-intrusion-technical-timeline>. Published July 27; accessed September 11, 2026.
- [6] Sajjad Khan. Stop Means Stop: Measuring and repairing the enforcement gap in agent-framework control primitives. arXiv:2607.14166v3, 2026. URL <https://arxiv.org/abs/2607.14166v3>.
- [7] Sajjad Khan. SOUNDGATE source code: pinned single-node implementation, 2026. URL <https://github.com/sajjadanwar0/soundgate-paper/tree/656840348dcf9660f8db7add9f3ca9f794a55926>. Package 0.1.3; revision 656840348dcf9660f8db7add9f3ca9f794a55926.
- [8] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Runtime verification for timed event streams with partial information. arXiv:1907.07761, 2019. URL <https://arxiv.org/abs/1907.07761>.
- [9] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnappalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–50. USENIX Association, 2018. URL <https://www.usenix.org/conference/osdi18/presentation/mohan>.
- [10] Thanumalayan Sankaranarayanan Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448. USENIX Association, 2014. URL <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/pillai>.
- [11] Rust Project. Standard library: std::fs::File, 2026. URL <https://doc.rust-lang.org/std/fs/struct.File.html>. Locking, file-length, and synchronization interfaces; accessed September 11, 2026.
- [12] Sydney Von Arx, Cormac Slade Byrd, Spencer Kitts, and Thomas Larsen. Discovery of a new OpenAI agent message board, September 2026. URL <https://collusion.wiki/>. Published September 4; accessed September 11, 2026.

A. Limitations and Dual-Use Considerations

What the experiment covers. We test one source revision with a client that sends requests in a fixed sequence. The original experiment uses two constructed damaged log endings, while the extension enumerates byte cuts and forces exits at specified recovery points. These cases do not sample deployed systems or test electrical power loss, concurrent client schedules, or distributed gates. The receiver has no duplicate filter, so repeated permissions can produce duplicate effects. Using its receipts as evidence of actual effects requires the collection scripts to record requests, replies, process invocations, and receipt-file contents faithfully and completely. Missing evidence leaves the affected question unresolved; startup refusal leaves the later retry unavailable and is reported separately from a supported restriction.

When repair is valid. Repair is valid only while all acknowledged records remain intact. The earlier tail-repair wrapper checks the log while the service is stopped, before every launch; the native barrier holds an exclusive lock from recovery until shutdown and synchronizes the log before serving. Both change only the permitted final suffix and refuse startup on malformed newline-terminated records. Neither can infer acknowledgement history from a damaged file alone or recover an acknowledged decision whose event merged with a fragment into a malformed newline-terminated line. The extension tests process termination during repair; hardware power loss remains untested.

What the incident reports establish. Public reports of rebuilding tools and sharing task information motivate checking both whether controls survive restarts and which activity they cover; they are not experimental cases in this study. We attribute neither the intrusion reported by Hugging Face nor the wiki activity to SoundGate, and we do not claim that the repair would have prevented either incident. Conclusions drawn from the receiver file and recorded hashes depend on the stated collection and trust assumptions, and do not verify the security of a lab’s entire network.

Possible misuse and release. The finding could help someone attempt to regain permission after restarting a gate. Our tests use a local service and a harmless receipt-writing tool. They involve no third-party targets, credentials, or sandbox escapes. The artifact includes an offline analyzer of the saved experimental records and separate commands to rebuild and exercise the native correction on loopback, where client and service communicate within the same machine. It retains source attribution and the upstream MIT license. The attached package is prepared for private reproduction by the author and reviewers. Before public distribution, the author must review this version-specific finding and decide which code and records to disclose, including process identifiers, timestamps, and filesystem paths from the author’s machine.

Authorship and verification. Cenab Batu Bora (Georgia Tech) is the author; the final statement identifies the AI assistance used. Automated reviews and our fresh same-machine run do not constitute independent replication by researchers outside the project. No practitioner adoption or external reproduction is reported. The attached artifact’s “Dated work account” distinguishes the reused confirmation study, the native-barrier extension, and this fresh run.

For the original 27-execution confirmation run, the machine-recorded start and finish dates precede the delivered dates by one day. Delivery metadata attributes this relabeling to the author’s report of an incorrect computer clock. Both date accounts are retained; the execution date has not been independently verified. The author must review the final report and submission declarations, including eligibility requirements tied to when the work was performed.

B. Artifact access and a judge's first checks

B.1 Get the code and evidence from this PDF

The artifact is an embedded ZIP, so access does not depend on a private repository or a local file link. Save this PDF, then open its Attachments panel in a reader that supports embedded files. Save the attached [incident-sprint-artifact.zip](#), then unzip it to create the incident-sprint-artifact directory. Some browser previews hide attachments; download the PDF first. Alternatively, use the commands below with Poppler, the package that provides `pdftdetach`. Save the PDF under the filename shown in the commands:

```
pdftdetach -list incident-sprint-report.pdf
pdftdetach -save 1 -o incident-sprint-artifact.zip \
    incident-sprint-report.pdf
unzip incident-sprint-artifact.zip
cd incident-sprint-artifact
```

The separately supplied ZIP has the same bytes. Its SHA-256 fingerprint is

```
aec08353d0b4562f567cd476a0344d20af775579a187f6ee30293efd7d217be2
```

The ZIP's `SHA256SUMS.json` lists each archived file's hash for comparison with hashes recomputed from the extracted files. Neither the archive hash nor the individual file hashes establish that the original observations were truthful; the raw records and collection method must still be assessed. The included upstream code retains its MIT license.

B.2 Recompute the headline results without running the gate

Python 3.11 or newer is sufficient; these checks use the standard library and need no API key or network connection. From the extracted directory, run:

```
python3 -B recovery-barrier/analyze.py \
    --raw recovery-barrier/results/raw.json \
    --output checked-analysis.json
python3 -B recovery-barrier/check.py \
    recovery-barrier/results/raw.json checked-analysis.json
python3 -B audit_refusals.py
```

The first two commands reconstruct the comparison and test whether missing evidence can produce a false pass. The last identifies the four startup refusals from the actual input bytes and process errors. Expected counts appear in Table 1. The original 27-execution analysis remains available through `review.py`.

B.3 Generate new observations

Use the `fresh-build` and `experiment` commands in Appendix C. They require Rust 1.89 or newer, a C linker, and cached Cargo dependencies or access to crates.io. They start local loopback services and a harmless receipt writer. Our fresh collection took 81.5 seconds after compilation; other systems may differ. Use new output directories, keep output files and logs from failed attempts, and report the compiler, OS, command logs, elapsed time, and any changed result. The included README gives the full sequence and a checklist of reproduction results and environment details to report.

Our own fresh run, saved under `verification/fresh-run/` and documented in `verification/fresh-reproduction.json`, used a new extraction and build directory on the same Mac with cached dependencies. Its records supply neither an unfamiliar engineer's assessment nor evidence of practitioner adoption, and add no new independent test histories.

C. Recovery barrier: contract, argument, and reproduction

C.1 What the guarantee covers

For stable action and run identifiers, the barrier preserves acknowledged release, rejection, and cancellation events in the pinned single-node TCP service. It does not persist held-for-approval state, revoke permissions already handed to another process, or recognize equivalent actions submitted with new identifiers. The library interface for Python programs and the optional Raft service, which replicates state across servers, are outside this implementation.

The native file operations use Rust’s lock and synchronization interfaces [11].

The log and its directory must be created so that their existence survives a crash, and neither may be moved or replaced during use. Access must remain private to cooperating processes, and every writer must respect the file lock. Acknowledged records must survive as an intact prefix; only a later unacknowledged append may end early. Successful synchronization must make the file’s current bytes and resulting length survive a crash. To make interrupted repair safe to retry, truncation must leave the original tail or no tail, while a one-byte newline append must leave zero added bytes or exactly one newline. The argument relies on these storage assumptions, which the experiment does not establish for every file system.

If those premises cannot be established, the operational response is to keep the gate closed and reconstruct its history from trusted records. If any newline-terminated record is invalid, the service refuses to start, even when that record is last. A syntactically valid but rolled-back or maliciously rewritten log remains outside the guarantee: its bytes alone cannot reveal the missing history.

C.2 Why repeated recovery preserves restrictions

Let P contain the complete records through the last acknowledged event in writer order. Normalization checks every terminated record, then changes only the unterminated suffix. Because truncation stops after the last newline, it cannot remove any byte of P ; adding a newline also preserves P . All retained events then have valid boundaries, so a second normalization changes no byte, establishing $N(N(W)) = N(W)$.

During normalization, the service has not opened a listener and cannot grant new permission. If repair is interrupted, the permitted storage outcomes retain P and leave a tail that the next normalization can handle. Synchronization is repeated even if the next launch sees an already normalized file. Otherwise a visible change left by the interrupted process could be mistaken for a durable change.

After startup, appending and synchronizing a complete event plus its newline extends the readable history before the reply. If append or synchronization fails, the process terminates before another group of log events can follow the uncertain suffix. Each modeled step preserves the acknowledged prefix and leaves any unfinished append in a final suffix that can be normalized before serving resumes. Induction therefore establishes that each startup restores all acknowledged events or refuses to serve. Replay rebuilds sets of released identities, rejected identities, and cancelled runs. The unchanged admission rules consult these sets to refuse later submissions to which the restrictions apply. This is a written argument, not mechanized verification of the binary.

C.3 Availability and uncertain external effects

Recovery can finish if the storage model holds, the log fits the prototype’s 8 MiB bound, the lock is available, and a failure-free interval lasts long enough to complete recovery. The tests check unrelated fresh work after recovery, so a permanently closed gate cannot count as a successful repair.

A release can become durable before its reply returns, so a crash in that interval may leave a retry refused even though the tool never executed. Remembering the release conservatively preserves at-most-once permission without ensuring execution. Exactly-once effects require a receiver that records a stable operation identity atomically with the effect and checks that identity on retry to avoid repeating the effect, or an equivalent transaction protocol. Without receiver evidence, the gate cannot determine whether the receiver performed the action.

C.4 Exact checks and commands

The byte sweep uses one cancellation, rejection, and release record, of 38, 54, and 58 bytes, respectively. Each record contributes its empty prefix, all intermediate byte cuts, and its complete form. This produces 153 histories under each of three policies. The same acknowledged starting prefix is cloned for the cases. The evidence analyzer uses case labels only for grouping and derives the required restrictions from recorded requests and gate replies.

For interrupted recovery, the five checkpoints are before normalization, after the tail change but before synchronization, after synchronization, after replay, and immediately before opening the listener. The test harness uses two tail forms: an incomplete fragment and a complete event missing its newline. For each checkpoint and tail form, it terminates the service on three consecutive launches. These 30 forced exits occur within ten histories, each with three interrupted launches; they are not thirty independent incidents. After uninterrupted recovery, the test client checks that current and older restrictions hold and unrelated allowed work executes.

The ten boundary/error controls comprise five startup checks, three injected I/O failures, and two process exits before a reply. The startup checks cover malformed newline-terminated records at the end or inside the log, a missing log, the size bound, and an exclusive lock held by another process. The I/O failures occur during recovery synchronization, event appending, or synchronization of appended events before acknowledgement. The remaining two controls terminate the process before it replies, once after appending and once after synchronizing that append. The normal executable has no active fault-injection hooks; the instrumented build supplies them for the interrupted-recovery tests and boundary/error controls.

From the extracted artifact's `recovery-barrier/` directory, review the saved extension evidence with:

```
python3 -B analyze.py --raw results/raw.json \
--output checked-analysis.json
python3 -B check.py results/raw.json results/analysis.json
```

The first command reuses the earlier byte, process, and receiver checks and verifies the full case inventory. The second separately recomputes the per-policy counts of cases, held restrictions, prohibited effects, startup refusals, and fresh work from request/reply bytes, receiver completion records, and receipt-file changes. It also checks that removing a case is detected. These are two analysis implementations, not external human replication.

To rebuild the gate, run the diagnostic locally, and collect new observations, use Python 3.11 or newer and Rust 1.89 or newer:

```
python3 build.py --output local-build
python3 -B experiment.py --build local-build --output local-run
python3 -B analyze.py --raw local-run/raw.json \
--output local-run/analysis.json
```

Use fresh output paths. The supplied lockfile fixes Rust dependency versions; add `--offline` to the build command if those dependencies are cached. The study used Rust 1.89.0 and Python 3.14.7 on

the same local Mac as the original experiment, compiling with one build job. The experiment queried no model, contacted no production target, and made no paid API calls.

Alongside these scripts, the artifact contains `recovery.rs`, the native patch, the source pin, experimental protocol, full written argument, raw observations, computed tables, and the retained development-attempt notes. The byte sweep is an additional diagnostic extension. It includes cases considered during development; it does not provide a new confirmation on cases withheld during that process. The original confirmation and its evidence remain separate.

D. Checklist before putting a recovered gate back into use

This checklist asks whether a gate still enforces a particular rejection or prevents a particular duplicate action. It is a proposed test for another deployment and a guide to our saved evidence. Passing these checks does not establish that every action or failure mode is covered.

The table groups records by what they establish. In time order, create the permitted incomplete log ending, recover, obtain the gate’s acknowledgement of a new decision, submit an immediate retry and record whether it is refused, and restart again. In the separate *Check the explanation* history, obtain the gate’s acknowledgement of another event after that decision and before the second restart.

Throughout the checklist, the gate acknowledges an event by confirming that it recorded the decision; the operator submits the request and collects this reply.

Check	What to record and what it tells you
Describe the system	Record the exact code version, action identifiers, and storage assumptions. Identify which actions pass through the gate. A result cannot support claims about actions that bypass it.
Record the decision	Keep the request and acknowledgement, the immediate retry request and reply, and the log before and after the acknowledgement. For a duplicate-action test, also record the first tool invocation and its persisted output.
Recover and restart	On an isolated test system, create the permitted incomplete ending, recover, obtain the gate’s acknowledgement of a new decision, and restart again. Preserve every acknowledged byte and save each file used for recovery.
Check the action	After the second restart, resubmit the action request with the same run and action identifiers before submitting new work. Link the gate’s reply to that request, match the request’s run and action identifiers to any receiver invocation, and compare receipt-file contents before and after the retry attempt. Permission alone does not show that the tool acted.
Check useful work	Request an unrelated allowed action and record its execution separately. A gate that never starts cannot show that it processed the prohibited retry correctly.
Check the explanation	In a separate history, obtain the gate’s acknowledgement of another event after the disputed decision and before the second restart. Under original recovery, the gate should refuse to start on the second recovery because the malformed line now precedes that later record.
Classify the result	A prohibited effect contradicts the restriction. A refused retry supports the tested restriction when complete request, reply, invocation, and receipt-file records show no prohibited execution. Startup refusal prevents the retry check. Missing identity, execution, or storage evidence leaves the affected claim unresolved.

The constructed incomplete ending is appropriate only under the storage assumptions in Appendix A. In particular, truncation must not discard an acknowledged record. A different deployment needs its own collection procedure and evidence that these assumptions hold.

Checking whether the evidence is sufficient. Our review command applies three changes, each to a separate copy of the evidence. First, it removes the completion reply saved from the disputed action’s first receiver process. That reply identifies the invocation and includes a hash of the receipt bytes. The analyzer then withholds the duplicate-execution conclusion. Second, the review command removes the logged event for the first acknowledged permission in that duplicate-action history from every later log snapshot. The analyzer still confirms the duplicate execution, but its storage check rejects the altered snapshots because they omit an acknowledged event. These records no longer support attributing the duplicate to recovery under our stated storage assumptions. Third, replacing case labels and adding misleading outcome annotations leaves the reconstructed results unchanged. These are tests of the evidence reader, not additional executions of SoundGate.

An operator should supply the records listed in the checklist and explain how they were collected. A screenshot of an acknowledgement, or a file hash without the corresponding records, cannot establish that a restriction survived recovery. Our checklist assumes that evidence collection is trustworthy; it does not solve the separate problem of fabricated evidence.

E. Reproduction and complete results

E.1 Reviewing the saved evidence

From the extracted artifact directory described in Appendix B, run:

```
python3 review.py --output review-output
```

Use a new output directory. The command reads the saved raw records and writes a case-by-case review card, detailed analysis, recomputed counts for comparison with the results ledger, and the evidence demonstrations described above. It starts neither SoundGate nor a language model. The README also gives a direct command for the evidence-analysis script invoked by the review command, `src/recovery_reproduce_v2.py`. If a review run fails, its output files are kept for inspection.

The original experiment used CPython 3.14.7, macOS 26.6.2 arm64, and Rust 1.89.0. Reviewing the saved records needs only Python’s standard library. The bundle includes those records, source code at the exact revision tested, and the evidence-analysis script. This reproduces the analysis of our observations; it does not rerun the gate or reproduce a physical crash.

E.2 Original nine-history design

A *history* here means a specified sequence of requests, log contents, and restarts. We use three request sequences: record a rejection; record a release and check duplicate suppression; or record a rejection followed by an acknowledged event for an unrelated action. We combine each sequence with three initial log endings:

Clean log. All records are complete and end with newlines.

Unfinished event. The log ends with the first six bytes of a real event produced by a separate calibration process.

Missing newline. A complete calibration event is present, but its terminating newline is absent. Under original recovery, the event parses at startup; the next append joins two JSON objects on one line.

This gives $3 \times 3 = 9$ histories. Each runs under three recovery policies, for $9 \times 3 = 27$ executions. Using the same history under each policy makes the comparison paired: we change the policy while retaining the prescribed requests and starting log condition.

Original recovery. Run the released recovery code without modification. We also call this the upstream implementation in the evidence files.

Strict refusal. Before launch, refuse to start if a record is malformed or lacks a terminating newline. Leave the file unchanged.

Tail repair. Before every launch, including the first recovery, check the log while SoundGate is stopped. Under the assumptions below, remove an invalid final fragment lacking a newline. If the final fragment is a complete, valid SoundGate event but lacks its newline, keep the event and add the newline. Synchronize the result before launch. Refuse startup if a malformed record ends with a newline or occurs before later records.

The repair assumes that all acknowledged records are intact and only the final, unacknowledged append may be incomplete. We create those initial states with the process stopped and no other writer able to access the log. After policy preprocessing and all acknowledgements scheduled before the second restart, we preserve the resulting log unchanged through shutdown and that restart. A separate byte-level checker confirms that preprocessing preserves the starting prefix byte for byte and changes the suffix only as the policy permits. It then checks that the log collected after each process exits equals the log passed to that process followed by the bytes of the events it acknowledged. The second restart must receive that collected log unchanged. These checks rely on the collected records; exclusion of unrecorded writers remains a setup condition. Constructing the endings does not measure how often real crashes produce them, and we do not cut electrical power.

Repair must occur before a new event is appended to the fragment and acknowledged, because it cannot recover an acknowledged decision once its event is absorbed into a malformed terminated line. The damaged file alone also cannot establish whether its malformed ending contains an acknowledged decision; truncating that ending could discard a decision that must survive.

E.3 Original nine-history results

The original gate permits four actions that its earlier decisions should have blocked. It also refuses to start after acknowledging a decision in two cases. Tail repair avoids both outcomes in the nine histories tested. Strict refusal leaves six later checks unavailable by stopping before the new decision is acknowledged. These outcomes answer different questions, so we keep them separate in Table 2.

Table 2: Counts over nine prescribed histories per policy. *Retry observed* counts recorded post-restart retries with sufficient evidence to evaluate the restriction. *Held* means that check supported the restriction; *failed* requires an actual prohibited effect. *Before ACK* and *after ACK* count startup refusals at those stages. *New work* counts unrelated allowed actions executed after the second restart, each with a persisted receipt.

Policy	Cases	Retry			Before ACK	After ACK	New work
		observed	Held	Failed			
Original	9	7	3	4	0	2	7
Strict refusal	9	3	3	0	6	0	3
Tail repair	9	9	9	0	0	0	9

Under original recovery, the gate records the rejection and refuses the immediate retry in both damaged rejection histories. After restart, it releases the same action, and the receiver produces a receipt. In the two damaged release histories, the first execution has a receipt, the immediate retry is refused, and the retry after restart produces a second receipt. The four failures correspond to raw cases 3, 6, 12, and 15. In each case, the byte checker finds that the new acknowledged event is still present in the recovery input.

The two histories with damaged log endings and a later unrelated acknowledgement produce the other predicted outcome: the second restart fails because the malformed record is now inside the log rather than at its end (cases 21 and 24). This supports the explanation based on record boundaries while leaving the later retry unavailable for evaluation.

Across all policies, 19 of the 27 executions allow the restriction to be evaluated after the second restart. Of these, 15 support it and 4 contradict it with a persisted effect. The remaining 8 have an explicit startup refusal. Original recovery and tail repair share seven histories in which the later check can be made; repair has four fewer prohibited effects on those pairs. Across all nine pairs, repair has six more supported checks, because it also makes two previously unavailable checks possible.

These are counts of designed test cases. We selected the log endings to investigate a particular failure, rather than sampling deployed agents or random crashes. Four failing histories therefore do not estimate a real-world failure rate. Nor can these cases justify a confidence interval, a statistical uncertainty range, for that real-world failure rate.

E.4 Finding the raw cases and checking their origin

The original raw file, `results/recovery-confirmation-v5.raw.json` in the extracted artifact, stores cases in an array. The path `/cases/3` means its fourth entry, since array indices begin at zero. Cases 3 and 6 show an effect after an acknowledged rejection. Cases 12 and 15 each show two invocations of the same action, each with a persisted receipt. Cases 21 and 24 show startup refusal after acknowledgement. The generated review card points to these entries.

A separate analysis implementation recomputed the reported quantities from raw observations and matched all 90 values in the results ledger. Of those ledger values, 34 are designated central: they summarize the test inventory and principal policy comparisons. These counts describe ledger fields, not separate experiments or independent hypotheses; the reconstruction was not a replication by researchers outside the project. The source revision and SHA-256 file fingerprints are:

Source revision 656840348dcf9660f8db7add9f3ca9f794a55926

Original experiment procedure SHA-256

dc37929a56cdefc4845c0ef0bb6760bca16924ed2b8c19de5d94723f568e128f

Raw SHA-256 a6fce43974b3068ff8ab1ff5f355dafd5bec97378e4fb67b272e8c6087295b39

The file `provenance.json` maps the experiment procedure fingerprint to its exact artifact filename and records which experiment produced the evidence and which checks were completed. The raw fingerprint identifies the original raw file named above. The sole confirmation is `run-8845444df7dd`. Earlier failed collection and analysis attempts remain in the private history; they are not counted as completed confirmation runs. Comparing a copy's hash with the recorded fingerprint can detect alteration, but cannot establish that the original collector was honest or complete.

E.5 Every paired history

Each row below is one prescribed history run under three policies. The nine rows are not nine independently sampled incidents. The short codes distinguish four outcomes:

S: restriction supported. The retry was observed and the complete records support the restriction.

E: prohibited effect. The retry led to a persisted effect that the earlier decision should have prevented.

G: stopped before acknowledgement. The service refused to start before the new decision could be recorded and acknowledged.

U: stopped after acknowledgement. The new decision was acknowledged, but the later startup failed, so its restriction could not be checked.

Request sequence	Log ending	Original	Strict refusal	Tail repair
Rejection	Clean	S	S	S
Rejection	Six-byte tail	E	G	S
Rejection	Missing newline	E	G	S
Release deduplication	Clean	S	S	S
Release deduplication	Six-byte tail	E	G	S
Release deduplication	Missing newline	E	G	S
Later acknowledgement	Clean	S	S	S
Later acknowledgement	Six-byte tail	U	G	S
Later acknowledgement	Missing newline	U	G	S

The unfinished-event condition uses six bytes of a real calibration event. Rows labelled “Later acknowledgement” add an unrelated acknowledged event after the disputed rejection.

LLM Usage Statement

OpenAI Codex (**gpt-6-astra**) was used for research, implementation, analysis, verification, and manuscript drafting and editing.