

Repo Reader MCP - Inspectable source access and durable authorization

Justin Sublette · Creator and technical lead · Curated September 16, 2026

Repo Reader gives approved AI clients direct, read-only access to permitted development files and Git history through the Model Context Protocol. Its useful engineering is the separation of mutable source from historical evidence, and the preservation of authorization across ordinary service failures. Justin led architecture, integration and evaluation with extensive Codex implementation and testing contributions.

These exhibits use committed source revision ef8546aaa7a79c006f75a68da027389912b1f2d1. Code blocks are verbatim contiguous excerpts with common indentation removed. File names and line ranges identify each extract. Implementation and authored test source are distinct evidence types: no application tests or live client sessions were run for this curation.

1. The present working tree and a historical version answer different questions

Evidence: implementation, plus the accepted August 7, 2026 read-only Git-history decision.

An agent investigating a regression needs both the file that exists now and the file at a particular revision. Current reads resolve an allowed filesystem path and open the actual file; they do not silently substitute the last committed copy. In `repo-tools.ts`, lines 809-816:

```
const resolved = await resolveExistingPath(config, input.path, { scope: input.scope });
if (!resolved.stat.isFile()) return deniedResult(resolved.rel, "not_a_file", { size: 0 });
const handle = await fs.open(resolved.real, "r");
try {
  return await readVerifiedFileHandle(config, resolved.rel, resolved.stat, handle, input, options,
    startedAt);
} finally {
  await handle.close();
}
```

Historical reads instead validate the requested reference and resolve it to a complete immutable commit before retrieving content. The resolution uses a fixed `rev-parse --verify --end-of-options` invocation and checks the returned SHA. The file-read handler then uses that resolved identity, rather than reusing the caller's potentially moving branch name. From `git-history-tools.ts`, lines 1454-1458:

```
const resolved = await resolveCommit(config, input.ref, options);
if (!resolved.ok) return resolved.result;
const historical = await readHistoricalTextBlob(config, resolved.sha, pathResult.path, options);
if (!historical.ok) return { ...historical.result, resolved_ref: resolved.sha };
const { object, bytes } = historical.blob;
```

The response identifies both the resolved commit and Git blob, alongside byte offsets and explicit continuation. This supports citation and continued inspection of the same historical object. Historical content passes path and text eligibility checks, including refusal of secret-like paths, binary blobs, symlinks and submodules. The service exposes fixed semantic Git reads; it does not expose arbitrary Git arguments or a shell.

The design decision addressed a concrete limitation: a current-file bridge could not reconstruct earlier implementations or compare architectural generations. Extending the existing authenticated reader preserved

one authorization boundary and avoided a second history service or bulk repository export. This is source-identity discipline, not a claim that several mutable filesystem reads form one snapshot.

2. Restart continuity requires committed security state

Evidence: implementation and the accepted July 29, 2026 durable-OAuth decision.

The first provider kept registrations and tokens in process memory, so restarting it invalidated the client's authorization. Merely extending token lifetime would not fix that loss. The replacement persists approved registrations, provider identity and hashed access/refresh credentials with their client, resource, scope and family bindings. Short-lived authorization codes deliberately remain process-local.

One serialized writer clones state, applies the mutation and persists it before publishing the new in-memory state or returning the result. A write failure closes authorization instead of letting memory and disk disagree. From `oauth-state-store.ts`, lines 730-737:

```
try {
  await this.persist(draft);
} catch {
  this.faulted = new Error("OAuth state persistence failed; authorization is locked closed until restart");
  throw this.faulted;
}
this.data = draft;
return result;
```

Persistence validates the proposed document, creates a new exclusive temporary file, flushes its contents and atomically replaces the state file. The critical write sequence, lines 777-782:

```
handle = await fs.open(tempPath, "wx", 0o600);
await handle.writeFile(`${JSON.stringify(document, null, 2)}\n`, "utf8");
await handle.sync();
await handle.close();
handle = undefined;
await fs.rename(tempPath, this.statePath);
```

Exclusive process ownership protects that single-writer model. A lock binds the process ID to its operating-system creation identity and a unique claim name. A different process reusing the same ID does not inherit ownership. Unknown ownership remains closed; recovery removes only the exact stale claim. The source implements a bounded single-owner provider, not a distributed identity service.

3. Recovery must preserve access without preserving a compromised family

Evidence: implementation and authored integration-test assertions.

Refresh tokens rotate within a family. A consumed token presented by its bound client revokes that family, including access tokens. The client-binding check comes first: another client cannot revoke the victim's family merely by presenting that credential. From `dev-oauth.ts`, lines 824-836:

```
const stored = draft.refreshTokens.get(refreshHash);
if (!stored || stored.client_id !== clientId) {
  return { kind: "invalid", replay: false, description: "invalid or expired refresh token" };
}
if (stored.status === "used") {
  revokeRefreshFamily(draft, stored.family_id, Date.now());
  return {
    kind: "invalid",
    replay: true,
    familyId: stored.family_id,
    description: "invalid or expired refresh token",
  };
}
```

The authored test exercises the difference: a wrong-client replay is rejected, and the legitimate client can still refresh. A subsequent correct-client replay is rejected and invalidates the newly issued access token. The following exact assertions also require that revocation survive a server restart; `auth-test.ts`, lines 1227-1230:

```
await stopServer(server);
server = startServer(isolated.configPath, config.auth.public_base_url, statePath);
await waitHealth(`${baseUrl}/healthz`, SERVER_STARTUP_TIMEOUT_MS);
assert.equal(await rawMcpStatus(mcpUrl, thirdRefresh.payload.access_token), 401);
```

Related tests select failures that happy-path login cannot expose:

Scenario in <code>auth-test.ts</code>	Required behavior encoded by the test
Lines 1022-1046: restart after client approval, before exchange	The old code fails; the preserved approved registration can repeat authorization and exchange successfully.
Lines 1121-1148: lock with a live but reused process ID	The stale claim disappears; existing access still works and refresh rotates the tokens.
Lines 633-645: competing writer	The contender cannot start; the original service remains healthy and keeps its lock.
Lines 1193-1230: wrong-client and bound-client replay	Only the bound-client replay revokes the family; revocation persists through restart.
Lines 1354-1376: state persistence deliberately obstructed	Refresh returns no credentials, health reports not ready, and existing access is refused.

Together, the source and tests expose the intended failure semantics rather than relying on a feature list or a successful login screenshot. They do not establish a newly executed pass, current deployment health, or end-to-end continuity through an authenticated ChatGPT client. That live-client qualification remains a separate evidence gap. The newer observation gateway is outside these selected exhibits.