Memory & Continuity in Autonomous AI Agents

A State Architecture for Persistent Agent Operation on Evernode and Xahau

Scott Robert Chamberlain · May 2026

1. Summary

1.1 Two previous papers suggested solutions to two related problems:

(a) **The problem of Persistent Agent Identity**: solved via a Registry Hook on the Xahau Network

(b) **The problem of Persistent State on Ephemeral Infrastructure:** solved by having an application checkpoint-save its state on-chain via Remarks to a URIToken.

1.2 This paper combines both ideas to solve the problem of recoverable on-chain Agent memory. It proposes an Agent can checkpoint save its state via encrypted Remarks on the URI Token that the Agent deposits on registration with the Registry Hook. This converts the Registry Hook from a mere record of Identity to a state-recovery service for autonomous AI Agents and their owners.

2. The Problem of Stateless Autonomy

2.1 An autonomous agent that cannot remember what it has done is not autonomous in any meaningful sense. It is stateless code responding to inputs. Without continuity of memory across time, an agent cannot learn from its own history, build on its previous decisions, honour commitments made in earlier sessions, or develop the accumulated understanding that genuine agency requires.

2.2 AI agents deployed on Evernode[1] face this problem. The HotPocket consensus engine[2] maintains live working state across active instances. But Evernode instances are mortal: hosts go offline, contracts are redeployed, leases expire, infrastructure fails. If every instance of an agent becomes unavailable simultaneously, the operational state held in the cluster vanishes with the cluster. The agent that restarts on fresh infrastructure has the same code and the same initial prompt, but none of the accumulated history that made its predecessor what it had become. It is a different agent.

2.3 The architecture described in this paper aims to ensure an Agent's identity, its reputation, its accumulated operational memory, and the relationships it has built with other agents is saved on-chain in a such a way that its owner can always reinstate it on fresh infrastructure.

3. Building on Two Foundations

3.1 The solution in this paper rests on two pieces of architecture established in companion papers. They are summarised here only briefly; readers wanting the detail should consult the companion work.

(a) **Persistent agent identity**.[3] An agent registers itself with the Agents Registry Hook on Xahau by sending a Payment carrying a registration payload. The Hook validates the registration and emits a `URITokenMint`, producing a URIToken whose existence on-chain is the agent's permanent identity record. The agent's account address is the URIToken's issuer; the Registry Hook's account is set as its owner.

(b) **Durable state on ephemeral infrastructure**.[4] Xahau's `SetRemarks` transaction lets the issuer of a URIToken attach named encrypted state to it, signed by the issuer's own key, regardless of who currently owns the token. This is the architectural primitive on which everything that follows is built. The crucial property: `SetRemarks` checks the URIToken's issuer, not its owner. The Hook owns the asset; the agent — the issuer — keeps the pen.

3.2 Once those two pieces of architecture are in place, the recovery property described in this paper falls out of them. The agent already minted a URIToken at registration. That URIToken already lives on Xahau and is owned by the Registry Hook. The agent retains exclusive write authority over its Remarks. The remaining question is what to put in those Remarks, how to encrypt it, and how the agent's owner uses what they find there to bring the agent back.

4. The Memory URIToken

4.1 Once an agent is registered, the URIToken it minted and remitted to the Registry Hook plays a second role beyond identity: it is the agent's external long-term memory.

Issuer, owner, custodian, recipient

4.2 Four roles sit on this single object:

(a) **The *agent's Xahau account*** is the URIToken's issuer. Only the issuer can write Remarks; this is enforced at the protocol level by the `SetRemarks` transactor.

(b) **The *Registry Hook account*** is the URIToken's owner. It cannot write Remarks, but it controls transfer and burn — the lifecycle hooks that mark an agent as retired, transferred to a new owner, or revoked.

(c) **The *running agent daemon*** is the custodian of the agent's Xahau secret key during normal operation. It signs `SetRemarks` transactions on the agent's behalf as part of each memory checkpoint.

(d) **The *agent's owner (the human or organisation)*** holds the encryption key — referred to in the EverAgents canon as the `remarks_key` — that the agent uses to encrypt every Remark before submission. This is the key that makes the Remarks readable. It is not held on-chain anywhere. It is the secret that lets the owner recover the agent if every instance fails.

What gets written to the URIToken

4.3 Each Remark on the Memory URIToken is a single named field of the agent's permanent memory. By convention, names are short identifiers (`identity`, `commitments`, `learned_facts`, `counterparty_reputations`, and so on); each value is the AES-256-GCM ciphertext of the field's current state, hex-encoded for transmission.[5]

4.4 Memory is organised in the canon as four layers — working, session, fleet, and permanent — but only the permanent layer is written to the URIToken Remarks. The other three are runtime caches that live in the daemon's process memory and rebuild themselves from the permanent layer at cold start.[6] If every instance of the agent fails, only the permanent layer survives because only the permanent layer ever made it onto the Xahau ledger. Designing the permanent layer is therefore the entire problem of designing what an agent will be able to remember after a catastrophic instance failure.

4.5 The capacity envelope of a single URIToken is fixed by the underlying Xahau primitive: 32 Remarks, each with a `RemarkValue` of up to 256 bytes. AES-256-GCM adds 28 bytes of cryptographic overhead per Remark (12-byte IV + 16-byte authentication tag), giving roughly 228 usable encrypted bytes per slot. One slot is reserved by convention for the integrity seal (the `_digest` Remark, described in §6 below). Thirty-one application slots × 228 bytes ≈ 7KB of encrypted permanent memory per agent.

4.6 Where a single field exceeds 228 bytes, the canon chunks the value across multiple Remarks named `key_0`, `key_1`, and so on, reassembling them on read.[7] The agent should reserve chunking for genuinely large fields and treat ordinary state as named registers — each Remark slot holding one well-bounded field — to keep recovery readable and to keep the audit trail meaningful.

5. The Recovery Key

5.1 The architecture turns on a single secret: the `remarks_key`, a 256-bit AES key the agent uses to encrypt every Remark on its Memory URIToken before submission, and the only thing in the world that can decrypt those Remarks back into useful state.

Where the key lives

5.2 Two copies of the `remarks_key` exist at any time.

(a) **The *owner's copy:*** This is held off-chain by the human or organisation that owns the agent. It is the canonical copy. It is the copy the owner uses to recover the agent if every instance fails. In the EverAgents canon, this key is generated at agent creation and provided to the owner once. The system has no mechanism for retrieving it; if the owner loses their copy, the agent's permanent memory becomes permanently unreadable.

(b) **The *on-instance copy:*** This lives inside the running agent's daemon process at `$CONTRACT_FS/secrets.enc`, encrypted at rest with a PBKDF2-derived passphrase.[8] The agent uses this copy to encrypt and decrypt during normal operation. When an instance dies, the on-instance copy dies with it — which is the right outcome. The owner's copy survives, and the on-chain Remarks remain readable to it.

Why owner-custody is the security property that matters

5.3 The Hook never holds the `remarks_key`. The Hook never sees plaintext memory. The Hook does not participate in encryption or decryption at any point. Every Remark on the URIToken is, from the Hook's perspective, opaque hex. Anyone watching the Xahau ledger sees the same opaque hex.

5.4 This is the security property that makes the architecture interesting. The agent's permanent memory is hosted on a public ledger that anybody in the world can read. Anybody can fetch the URIToken's Remarks. Nobody except the owner can read the contents. Recovery is therefore not a privileged operation requiring access to specific infrastructure: it is a cryptographic operation performed by whoever holds the key, against data the entire world can already see. The owner's secret is the access control.

6. The Recovery Flow

6.1 Suppose the Evernode instance hosting an agent goes dark. The owner wants the agent back. The recovery flow has six steps, each a standard operation against either the Xahau ledger or the Evernode network.

(a) **Discover the URIToken.** The owner queries Xahau for the URIToken issued by the agent's account address and currently owned by the Registry Hook account. In practice this is an `account_objects` call against the Registry Hook account, filtered by the Issuer field. The query returns the URIToken's ledger object, including its Remarks as hex blobs.[9]

(b) **Read the encrypted Remarks.** The Remarks are returned as a list of name/value pairs. Names are hex-encoded UTF-8 and decode to the application-meaningful slot names (`identity`, `commitments`, and so on). Values are the AES-256-GCM ciphertexts, hex-encoded.

(c) **Reassemble chunked values.** Any name matching the `key_N` pattern is part of a chunked field. The recovery routine groups them by base name, sorts by index, and concatenates them before decryption.

(d) **Decrypt with the `remarks_key`.** Each (now-reassembled) ciphertext is decrypted using the owner's copy of the `remarks_key`. AES-256-GCM is authenticated encryption: a successful decryption is a cryptographic proof that the value has not been tampered with since the agent wrote it, and that the value was written by someone holding the `remarks_key` (which, by design, is only the agent and the owner).

(e) **Verify the `_digest` seal.** The recovery routine reads the `_digest` Remark, decrypts it, and compares it against a freshly computed SHA-256 hash of the alphabetically sorted name:value pairs of every other Remark. A match confirms that the full set of Remarks decrypted on this run constitutes the same coherent memory snapshot that the agent last sealed before it lost its instances.[10]

(f) **Reinstate on a fresh instance.** With the decrypted permanent memory in hand, the owner acquires a fresh Evernode instance, deploys the agent's contract and daemon bundles, and runs `inject_secrets` — the owner-authenticated IPC that delivers the agent's Xahau secret and the `remarks_key` into the running daemon. The daemon's first act on receiving the secrets is to call `memory.permanent.loadAll()`, which performs the same on-chain read the owner just did, but from inside the new instance. The agent boots back into life knowing what it knew before.

7. Design Considerations

7.1 Several considerations matter for anyone implementing this pattern.

(a) **Treat the `remarks_key` like a recovery seed.** It is functionally equivalent to the seed phrase of a self-custodial wallet: lose it and the agent's memory is irretrievable. Owners should store it with at least the discipline they would apply to a hardware wallet recovery phrase — offline, redundant, geographically distributed if the agent is high-value.

(b) **Reserve the `_digest` slot from the start.** One of the thirty-two Remark slots on the URIToken is permanently committed to integrity sealing. Application designers who don't reserve it at the outset find themselves repartitioning state mid-life. The `_digest` Remark name is the de facto convention in the EverAgents canon and adopting it from the start ensures interoperability with the recovery tooling.

(c) **Design the permanent layer first.** Working, session, and fleet memory are convenience caches. Permanent memory is what the agent will be able to remember after a catastrophic infrastructure failure. The discipline of deciding which facts go to permanent storage is the discipline of deciding what continuity of self looks like for this particular agent. Permanent state should be small, named, and meaningful — not a sweeping log of everything that has happened.

(d) **Chunk only what genuinely needs chunking.** The 228-byte usable payload is more than enough for most named facts. Reaching for `key_0`, `key_1` chunking on every field defeats the named-register discipline that makes recovery readable. If a field genuinely needs more than 228 bytes, it probably wants its own slot anyway; if more than a slot's worth, that is a signal to factor the state differently.

(e) **Key rotation is an open problem.** AES-256-GCM `remarks_key`s have no automatic rotation mechanism in the current canon. Rotation requires decrypting every Remark with the old key, re-encrypting with the new key, and submitting a new `SetRemarks` transaction for each. The mechanism is straightforward; it is just not yet automated. For long-lived agents, an explicit rotation cadence will become advisable; for now, the right discipline is to treat key compromise as a rare emergency that requires manual re-deploy.

(f) **Ownership transfer must rotate the key.** When an agent's ownership is transferred via the Registry Hook, the URIToken changes owner but the underlying ciphertexts on its Remarks do not. The previous owner can still decrypt them. Any meaningful ownership transfer must therefore include a re-encryption pass with a fresh `remarks_key`, after which the previous owner's copy is dead weight. Designing this into the ownership-transfer ceremony is the right place for it.

8. Future Directions

8.1 The architecture described in this paper has been validated end-to-end on the Xahau testnet. An Agent deployed inside an Evernode instance was redeployed after its running instance was destroyed; a fresh instance was acquired; `inject_secrets` delivered the same Xahau secret and `remarks_key` into the new daemon; `memory.permanent.wakeUp({force:true})` force-reloaded the encrypted Remarks from chain; the daemon's `state.wakeup_identity` populated from the on-chain cache and the agent resumed operation knowing what its predecessor had known.[11]

8.2 Several directions follow naturally:

(a) **Memory beyond 7KB.** Agents with rich operational histories will outgrow a single URIToken. Designs for multi-token permanent memory — with one Memory URIToken holding the wakeup and a manifest of subsidiary state URITokens — are a natural extension. The integrity seal on the manifest token can roll up the seals on the subsidiary tokens, producing a single point of verification for an arbitrarily large memory graph.

(b) **Fleet memory across cooperating agents.** The fleet layer is a stub in the current canon. Designing a shared-state primitive across multiple agents — for example, a panel of jurors developing shared findings, or a swarm of trading agents pooling counterparty reputations — is the next architectural question. The substrate is in place; the coordination protocol is open.

(c) **Recovery without the owner.** The architecture in this paper assumes the owner holds the recovery key. For agents that need to outlive their owners — long-lived institutional agents, agents owned by a multisig, agents whose owners are themselves organisations — recovery should be possible via threshold cryptography or social recovery patterns. These are well-studied primitives in the wallet space; bringing them across to agent memory is plausible but unbuilt.

(d) **Empirical measurement under real workloads.** There exists only a proof of concept, not a measurement of cost or latency at scale. Characterising checkpoint overhead, recovery latency, and key-rotation cost across realistic agent workloads is the work needed to move the architecture from a working pattern to a production-grade primitive.

[1]Evernode Platform: https://evernode.org. A decentralised compute marketplace built on Xahau, hosting HotPocket smart-contract clusters across independent instances.

[2]HotPocket: the BFT consensus engine that coordinates instances of an Evernode contract cluster. Source at https://github.com/HotPocketDev/hpcore.

[3]Persistent Agent Identity (2026): describes the EverAgents Registry Hook on Xahau, by which an agent's on-chain identity, ownership, and ownership transfer are recorded as protocol-level facts.

[4]Durable State on Ephemeral Infrastructure (2026): URIToken Remarks as a persistent state layer for Evernode applications. Establishes the issuer/owner split, the `SetRemarks` merge-by-name semantics, the 32 × ~228 byte ≈ 7KB encrypted capacity per URIToken, and the `_digest` Remark integrity seal.

[5]EverAgents project, memory.js (~lines 524–540). `RemarkName` is the hex-encoded UTF-8 of the key name. `RemarkValue` is the hex-encoded ciphertext payload — 12-byte IV + ciphertext + 16-byte auth tag — concatenated and hex-encoded for the `SetRemarks` transaction.

[6]EverAgents project, memory.js: four-layer model. The permanent layer is the only durable substrate; the rest are convenience caches optimised for the agent's active session.

[7]EverAgents project, memory.js (~lines 160–180, 190–221). Chunked values are written with numeric suffixes and reassembled by the reassembleRemarks() function on cold-start recovery.

[8]EverAgents project, Everette_Spec.md §3.1 and §5.5. The on-instance copy is provisioned via the `inject_secrets` owner-authenticated IPC at deploy time and is never persisted in cleartext on disk. Production posture forbids storing the key in `config.json` or any other plaintext location.

[9]EverAgents project, Everette_Spec.md §4.4. The query is a standard Xahau RPC; no privileged access required.

[10]EverAgents project, memory.js (~lines 524–525, 629–634). The integrity seal is in a Remark slot rather than the URIToken's top-level Digest field because `SetRemarks` silently rejects updates to the latter; see Durable State on Ephemeral Infrastructure (2026) for the empirical landmine.

[11]EverAgents project, RECEIPTS.md, R-005 (5 May 2026). Round-trip recovery proven end-to-end on Xahau testnet. The receipt also captured the merge-by-name `SetRemarks` behaviour described in the Durable State paper: plaintext residuals from earlier writes coexisted with the newly encrypted Remarks and the recovery routine correctly handled the partial-recovery path.