The IRSB Ecosystem

Cryptographic Accountability for AI-Agent Transactions

Jeremy Longshore

2026-05-16

The IRSB Ecosystem

Cryptographic Accountability for AI-Agent Transactions

Built collaboratively with Claude Code. Every empirical claim and design choice in this paperback is grounded in the peer-reviewed literature catalogued in the companion BibTeX corpus at `content/citations/`.

Contents

1. IRSB Ecosystem
2. Deep Dive Part 1: Five On-Chain Enforcers That Make AI Agent Wallets Structurally Safe
3. Deep Dive Part 2: Cryptographic Receipts and the Evidence Pipeline That Proves What AI Agents Actually Did
4. Deep Dive Part 3: A 12-Package Nested Monorepo That Watches AI Agents for You
5. Deep Dive Part 4: Z3 Formal Verification, the Three-Layer Stack, and Claude Code as Architect
6. Guidewire MCP v0.1.0: Carrier-Native Server Blueprint

Chapter 1. IRSB Ecosystem

Three layers. Thirty-seven contracts. Approximately 1,200 tests. One mission: make AI-agent transactions cryptographically accountable.

The IRSB ecosystem is a three-layer stack — protocol, policy, and brokering — that together form the first on-chain accountability layer for AI-agent work. Every transaction produces an immutable receipt. Every solver posts a bond. Every violation triggers automated enforcement.

Paperback edition (PDF, 59 pages): /research-papers/irsb-ecosystem-paperback.pdf — hub + four deep dives + Guidewire MCP foundation-ship post concatenated, unified bibliography, print-ready.

Built collaboratively with Claude Code.

The Problem AI agents are getting wallet access. The tool-using paradigm Schick and colleagues introduced with Toolformer[1] and Yao and colleagues generalised into the ReAct loop[2] now extends to financial action: every major framework — AgentKit, ElizaOS, Olas, Virtuals, Brian AI, Safe — gives agents the ability to sign transactions. None of them answer the question: **what happens when the agent does something wrong?** Li's survey of LLM-based agent paradigms[3] documents the rapid expansion of the agent surface and the corresponding gap in accountability infrastructure.

Framework	Spend Limits	Execution Receipts	Monitoring	Dispute Resolution
Coinbase AgentKit	—	—	—	—
ElizaOS	—	—	—	—
Olas	—	—	—	—
Virtuals Protocol	—	—	—	—
Brian AI	—	—	—	—
Safe + Modules	—	—	—	—
IRSB + Moat	**On-chain**	**On-chain**	**Watchtower**	**Automated**

Soft reputation systems require trusting someone. IRSB replaces trust with math and economics — the same posture Schneier and Kelsey articulated for forensic audit

[1] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. *NeurIPS*. arXiv:2302.04761.

[2] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR*. arXiv:2210.03629.

[3] Li, X. (2024). A Review of Prominent Paradigms for LLM-Based Agents: Tool Use, Planning (Including RAG), and Feedback Learning. *COLING*. arXiv:2406.05804.

logs[4] and Haber and Stornetta proposed for time-stamped documents[5]: integrity that is verifiable without depending on the trustworthiness of any single party. Greshake and colleagues' work on indirect prompt injection[6] makes the case acute: an agent's instructions can be compromised through ordinary tool use, so the safety property must live outside the agent.

Architecture Three layers, clear separation of concerns. The decomposition follows the *information-distribution* discipline Parnas articulated in 1971[7]: each layer hides decisions the other layers do not need to know.

```
Layer 3: Scout - Intelligent Brokering
Discovers bounties (Algora/Gitcoin/Polar/GitHub)
Matches agents · Routes through Moat · Collects receipts

Layer 2: Moat - Policy-Enforced Execution
Scope policies · Budget policies · Domain allowlists
Default-deny gateway · Z3 formal verification

Layer 1: IRSB Protocol - Cryptographic Accountability
Receipts · Bonds · Disputes · Delegation · Enforcers
11 contracts deployed on Sepolia
```

Layer 1: IRSB Protocol Cryptographic accountability for AI-agent transactions. GitHub | Dashboard

Status: 11 contracts deployed on Sepolia testnet (v1.4.0). 552 protocol tests.

Core Contracts

- **SolverRegistry** — solver lifecycle, bond staking (0.1 ETH minimum), slashing, three-strikes jail, reputation decay (30-day half-life). The bond-and-slash economic shape inherits from the staking-and-finality literature, most directly Buterin and Griffith's Casper FFG design[8].

[4] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM TISSEC*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

[5] Haber, S., & Stornetta, W. S. (1991). How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2), 99–111. https://doi.org/10.1007/BF00196791

[6] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *AISec '23.* https://doi.org/10.1145/3605764.3623985

[7] Parnas, D. L. (1971). Information Distribution Aspects of Design Methodology. *Proceedings of the IFIP Congress.* https://doi.org/10.1184/R1/6606470.V1

[8] Buterin, V., & Griffith, V. (2017). *Casper the Friendly Finality Gadget.* arXiv:1710.09437.

- **IntentReceiptHub** — receipt posting with ECDSA signature verification, 1-hour challenge window, dispute resolution, batch posting.
- **DisputeModule** — arbitration for complex disputes, escalation from deterministic to human resolution.

Delegation & Enforcers (EIP-7702)

- **WalletDelegate** — EIP-7702 delegation[9] with ERC-7710 redemption[10] and caveat enforcement. `beforeHook` / `execute` / `afterHook` pipeline.
- **SpendLimitEnforcer** — daily plus per-transaction spend caps with ERC-20 calldata parsing.
- **TimeWindowEnforcer** — session time bounds.
- **AllowedTargetsEnforcer** — contract address whitelist.
- **AllowedMethodsEnforcer** — function selector whitelist.
- **NonceEnforcer** — replay prevention.

Supporting Contracts

- **EscrowVault** — ETH + ERC-20 escrow.
- **X402Facilitator** — x402 payment settlement[11] (direct + delegated).
- **AgenticCommerce** — EIP-8183 agentic commerce protocol.
- **CredibilityRegistry** — on-chain credibility tracking.
- **ERC8004Adapter** — validation signal publishing[12].
- **IRSBHook** — bridges EIP-8183 jobs into the IRSB accountability pipeline.

Layer 2: Moat Policy-enforced execution layer. GitHub

Status: code-complete. Z3 formal verifier (42 tests). Gateway, control-plane, trust-plane, and MCP server[13] implemented.

Every agent call passes through the Moat Gateway, which enforces: - **Scope policies** — which capabilities the agent can access - **Budget policies** — spending limits per tenant/time period - **Domain allowlists** — no open proxy, only declared outbound domains - **Default-deny** — if not explicitly allowed, it is blocked. The posture is Saltzer and Schroeder's *fail-safe defaults* principle applied at the transaction layer[14].

The **FormalAgentVerifier** uses the Z3 SMT solver[15] to provide mathematical proofs

[9] Buterin, V., Hancock, S., Akhunov, A., Beiko, T., & St. Pierre, T. (2024). *EIP-7702: Set EOA Account Code.* https://eips.ethereum.org/EIPS/eip-7702

[10] ERC-7710 contributors (2024). *ERC-7710: Smart Contract Delegation Redemption.* https://eips.ethereum.org/EIPS/eip-7710

[11] x402 Working Group (2024). *x402: HTTP 402 Payment Required Protocol.* https://www.x402.org/

[12] ERC-8004 contributors (2024). *ERC-8004: Agent Identity and Validation Signals.* https://eips.ethereum.org/EIPS/eip-8004

[13] Anthropic (2024). *Model Context Protocol Specification.* https://modelcontextprotocol.io/

[14] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[15] de Moura, L., & Bjørner, N. (2008). Z3: An Efficient SMT Solver. *TACAS*. https://doi.org/10.1007/978-3-540-78800-3_24

of constraint satisfaction — nine constraints across file access, network, command execution, data exfiltration, resource limits, and permissions. Fail-closed: `UNKNOWN` is treated as unsafe. The smart-contract verification literature is consistent on this point: symbolic-execution approaches (Luu and colleagues' Oyente[16], Kalra and colleagues' ZEUS[17]) all use the same closed-world discipline.

Layer 3: Scout **Intelligent brokering agent.** Lives inside the Moat repo.

Status: 8 MCP tools implemented. Broker scope defined.

Scout discovers available work (bounties on Algora, Gitcoin, Polar, GitHub), matches it to the right solver/agent, and routes it through Moat. Eight MCP tools:

- `capabilities.list` / `capabilities.search` / `capabilities.execute` / `capabilities.stats`
- `bounty.discover` / `bounty.triage` / `bounty.execute` / `bounty.status`

Scout does not execute work — it finds the right worker and ensures the work flows through policy enforcement.

Off-Chain Services

Solver (v0.3.0) — 139 tests Policy gate (4 checks: jobType allowlist, expiry, requester allowlist, size guard), evidence-bundle creation with SHA-256 artifact hashing, canonical JSON for deterministic hashing, Cloud KMS signing with DER parsing and EIP-2 low-S normalisation[18]. The off-chain compute pattern follows Eberhardt and Tai's design rationale for off-chaining computation while keeping commitments on-chain[19].

Status: code-complete. Not yet deployed to production infrastructure.

Watchtower (v0.5.0) — ~500 tests 12-package nested pnpm monorepo. Evidence verification, behaviour-signal derivation (10 signals with severity weighting), risk scoring with critical override, auto-dispute pipeline, circuit-breaker + retry resilience patterns. The architecture follows the watchtower literature for off-chain dispute monitoring: McCorry and colleagues' Pisa[20], Avarikioti and colleagues' Cerberus

[16] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making Smart Contracts Smarter. *CCS.* https://doi.org/10.1145/2976749.2978309

[17] Kalra, S., Goel, S., Dhawan, M., & Sharma, S. (2018). ZEUS: Analyzing Safety of Smart Contracts. *NDSS.*

[18] Wood, G., & Reitwiessner, C. (2015). *EIP-2: Homestead Hard-fork Changes.* https://eips.ethereum.org/EIPS/eip-2

[19] Eberhardt, J., & Tai, S. (2017). On or Off the Blockchain? Insights on Off-Chaining Computation and Data. *ESOCC.* https://doi.org/10.1007/978-3-319-67262-5_1

[20] McCorry, P., Bakshi, S., Bentov, I., Miller, A., & Meiklejohn, S. (2019). Pisa: Arbitration Outsourcing for State Channels. *AFT.* https://doi.org/10.1145/3318041.3355461

Channels[21], and Khabbazian and colleagues' Outpost[22] all argue that watchtowers must operate without blocking the on-chain happy path.

Status: code-complete. Chain context currently uses mock data — real IRSB client integration pending.

Agents (v0.2.0) — 42 tests Z3 FormalAgentVerifier[23], RAG pipeline with 12 prompt-injection patterns (drawn from the Greshake and colleagues taxonomy[24] and the broader jailbreak survey literature[25]), ledger tracking.

Indexer (v0.1.0) Envio HyperIndex for IRSB contract events → GraphQL API.

Deployed Contracts (Sepolia)

Contract	Address	Etherscan
SolverRegistry	`0xB6ab964832808E49635fF82D1e96D6a888ecB745`	View
IntentReceiptHub	`0xD66A1e880AA3939CA066a9EA1eD87ad3d01D977c`	View
DisputeModule	`0x144DfEcB57B08471e2A75E78fe0c2A74A89DB79D`	View
ERC-8004 IdentityRegistry	`0x8004A818BFB912233c491871b3d84c89A494BD9e`	View
IRSB Agent ID	**#967**	8004scan.io

Operational Accounts:

Account	Address
Deployer/Operator	`0x83A5F432f02B1503765bB61a9B358942d87c9dc0`
Safe (Owner)	`0xBcA0c8d0B5ce874a9E3D84d49f3614bb79189959`

Standards Referenced

[21] Avarikioti, G., Litos, O. S. T., & Wattenhofer, R. (2020). Cerberus Channels: Incentivizing Watchtowers for Bitcoin. *Financial Cryptography and Data Security (FC)*. https://doi.org/10.1007/978-3-030-51280-4_19

[22] Khabbazian, M., Nadahalli, T., & Wattenhofer, R. (2019). Outpost: A Responsive Lightweight Watchtower. *AFT*. https://doi.org/10.1145/3318041.3355464

[23] de Moura, L., & Bjørner, N. (2008). Z3: An Efficient SMT Solver. *TACAS*. https://doi.org/10.1007/978-3-540-78800-3_24

[24] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *AISec '23*. https://doi.org/10.1145/3605764.3623985

[25] Yi, S., Liu, Y., Sun, Z., Cong, T., He, X., Song, J., Xu, K., & Li, Q. (2024). Jailbreak Attacks and Defenses Against Large Language Models: A Survey. arXiv:2407.04295.

Standard	Role in IRSB
ERC-7683	Cross-chain intent format[26]
EIP-7702	EOA delegation to smart contracts[27]
ERC-7710	Delegation redemption framework[28]
ERC-7715	Permission-request protocol[29]
ERC-8004	Agent identity and validation signals[30]
x402	HTTP payment-protocol integration[31]

Quick Stats

Protocol contracts	37 Solidity files
Deployed on Sepolia	11 contracts (v1.4.0)
Protocol tests	552 (Foundry)
Solver tests	139 (Vitest)
Watchtower tests	~500 (Vitest)
Agent tests	42 (Pytest)
Total tests	~1,200
Languages	Solidity, TypeScript, Python
EIPs implemented	6
Dashboard	irsb-protocol.web.app
Monorepo	github.com/jeremylongshore/irsb
Moat	github.com/jeremylongshore/moat
License	BUSL-1.1 → MIT on 2029-02-17
CI	5 GitHub Actions workflows

Deep Dive Series

- Part 1: Five On-Chain Enforcers That Make AI Agent Wallets Structurally Safe
- Part 2: Cryptographic Receipts and the Evidence Pipeline
- Part 3: A 12-Package Nested Monorepo That Watches AI Agents for You

[26] Across Protocol and Uniswap Labs (2024). *ERC-7683: Cross-Chain Intents Standard.* https://eips.ethereum.org/EIPS/eip-7683

[27] Buterin, V., Hancock, S., Akhunov, A., Beiko, T., & St. Pierre, T. (2024). *EIP-7702: Set EOA Account Code.* https://eips.ethereum.org/EIPS/eip-7702

[28] ERC-7710 contributors (2024). *ERC-7710: Smart Contract Delegation Redemption.* https://eips.ethereum.org/EIPS/eip-7710

[29] ERC-7715 contributors (2024). *ERC-7715: Request Permissions for Transactions.* https://eips.ethereum.org/EIPS/eip-7715

[30] ERC-8004 contributors (2024). *ERC-8004: Agent Identity and Validation Signals.* https://eips.ethereum.org/EIPS/eip-8004

[31] x402 Working Group (2024). *x402: HTTP 402 Payment Required Protocol.* https://www.x402.org/

- Part 4: Z3 Formal Verification, the Three-Layer Stack, and Claude Code as Architect

---

Further reading A consolidated bibliography for the IRSB ecosystem — covering Ethereum consensus and finality, payment-channel watchtowers, smart-contract formal verification, MEV, agent paradigms, and cryptographic audit-log integrity — is maintained at /citations/ (BibTeX + per-source verification metadata).

Don't own a chain. Own the standard.

Chapter 2. Deep Dive Part 1: Five On-Chain Enforcers That Make AI Agent Wallets Structurally Safe

Every framework that lets a language-model agent control a wallet eventually faces the same uncomfortable question: what stops the agent from doing something the operator did not intend? Current answers range from "we wrote a policy document" to "we trust the model." Neither is acceptable when real funds are involved. The field has been shipping capability — the tool-using paradigm Schick and colleagues introduced[32], the reason-then-act loop Yao and colleagues generalised[33], the broader agent-paradigm space Li surveyed[34] — without shipping the matching safety infrastructure. The gap is widening, and the threat surface Greshake and colleagues catalogued for indirect prompt injection in LLM-integrated applications[35] applies as directly to wallet-controlling agents as it does to text-generating ones. Yi and colleagues' jailbreak survey[36] documents how rapidly that surface continues to evolve.

The IRSB Ecosystem is a protocol-level attempt to close the gap — not by adding agent memory or better prompting, but by encoding limits in Solidity and putting them on-chain before a transaction ever executes. The Solidity-level posture is itself informed by the smart-contract security literature: Atzei, Bartoletti, and Cimoli's 2017 systematisation of attacks on Ethereum smart contracts[37] is the canonical mapping of what goes wrong, and the formal-verification line of work (Luu and colleagues' Oyente[38], Kalra and colleagues' ZEUS[39]) is the canonical answer for catching it before deployment.

This is Part 1 of the IRSB Ecosystem Deep Dive series. It covers the on-chain enforcement layer: how EIP-7702 delegation works[40], what the five caveat enforcers do, how calldata parsing catches ERC-20 transfers, and what the SolverRegistry's bond-and-slash model adds on top. Eleven contracts are deployed on Sepolia testnet today.

---

[32] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. *NeurIPS*. arXiv:2302.04761.

[33] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR*. arXiv:2210.03629.

[34] Li, X. (2024). A Review of Prominent Paradigms for LLM-Based Agents: Tool Use, Planning (Including RAG), and Feedback Learning. *COLING*. arXiv:2406.05804.

[35] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *AISec '23*. https://doi.org/10.1145/3605764.3623985

[36] Yi, S., Liu, Y., Sun, Z., Cong, T., He, X., Song, J., Xu, K., & Li, Q. (2024). Jailbreak Attacks and Defenses Against Large Language Models: A Survey. arXiv:2407.04295.

[37] Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A Survey of Attacks on Ethereum Smart Contracts (SoK). *POST*. https://doi.org/10.1007/978-3-662-54455-6_8

[38] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making Smart Contracts Smarter. *CCS*. https://doi.org/10.1145/2976749.2978309

[39] Kalra, S., Goel, S., Dhawan, M., & Sharma, S. (2018). ZEUS: Analyzing Safety of Smart Contracts. *NDSS*.

[40] Buterin, V., Hancock, S., Akhunov, A., Beiko, T., & St. Pierre, T. (2024). *EIP-7702: Set EOA Account Code*. https://eips.ethereum.org/EIPS/eip-7702

The Wallet Delegation Problem Giving an agent raw private-key access to a wallet is the worst version of the problem. The agent can do anything, at any time, with no audit trail, and no recovery path if it goes wrong. A key can be extracted from a compromised environment. A model can be manipulated into transferring funds through a carefully crafted prompt — exactly the indirect-prompt-injection attack vector Greshake and colleagues catalogued[41]. There is no circuit breaker.

EIP-7702 changes the delegation model[42]. Instead of giving an agent a key, an EOA (externally owned account) delegates execution authority to a smart contract — the WalletDelegate. The EOA remains the owner; the contract becomes the gatekeeper. Every call the agent wants to make passes through the WalletDelegate, which enforces a set of caveats before the transaction touches the target. The caveat redemption flow follows the ERC-7710 framework[43].

The caveats are not optional hints. They are Solidity. If a caveat reverts, the call reverts. The agent cannot override them with a longer explanation or a different framing. This is the structural distinction that matters: policy-as-code enforced at the EVM level, not policy-as-prompt enforced at the model level. The same posture Saltzer and Schroeder named *complete mediation* in 1975[44] applies — the constraint must be evaluated on every access, with no skip path.

The beforeHook/execute/afterHook Pipeline The WalletDelegate's `executeDelegated()` function implements a three-phase pipeline. Before the actual call executes, every registered caveat enforcer runs its `beforeHook()`. After the call succeeds, every enforcer runs its `afterHook()`. If any hook reverts at any point, the entire transaction fails and state is not changed.

```
function executeDelegated(bytes32 delegationHash, address target, bytes calldata callData, uint
    external payable whenNotPaused nonReentrant returns (bytes memory result)
{
    TypesDelegation.StoredDelegation storage stored = _delegations[delegationHash];
    if (stored.delegator == address(0)) revert DelegationNotFound();
    if (!stored.active) revert DelegationNotActive();

    address delegator = stored.delegator;
    TypesDelegation.Caveat[] storage caveats = _caveats[delegationHash];

    // WD-2: Run all beforeHooks
    for (uint256 i = 0; i < caveats.length; i++) {
```

[41] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *AISec '23.* https://doi.org/10.1145/3605764.3623985

[42] Buterin, V., Hancock, S., Akhunov, A., Beiko, T., & St. Pierre, T. (2024). *EIP-7702: Set EOA Account Code.* https://eips.ethereum.org/EIPS/eip-7702

[43] ERC-7710 contributors (2024). *ERC-7710: Smart Contract Delegation Redemption.* https://eips.ethereum.org/EIPS/eip-7710

[44] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

```
        ICaveatEnforcer(caveats[i].enforcer)
            .beforeHook(caveats[i].terms, delegationHash, delegator, target, callData, value);
    }

    bool success;
    (success, result) = target.call{ value: value }(callData);
    if (!success) revert ExecutionFailed();

    // WD-2: Run all afterHooks
    for (uint256 i = 0; i < caveats.length; i++) {
        ICaveatEnforcer(caveats[i].enforcer)
            .afterHook(caveats[i].terms, delegationHash, delegator, target, callData, value);
    }

    emit DelegatedExecution(delegationHash, delegator, target, value);
}
```

Each enforcer receives the same inputs: the encoded terms for that caveat, the delegation hash, the delegator's address, the target contract, the raw calldata, and the ETH value. Enforcers are stateful — they can track cumulative spend, call counts, or any other metric across the lifetime of a delegation. The terms are ABI-encoded at delegation-creation time and cannot be modified without creating a new delegation.

The `nonReentrant` guard and `whenNotPaused` checks wrap the whole function. The contract owner can pause the system in an emergency without touching individual delegations. The reentrancy guard is canonical defence against the attack class Atzei and colleagues catalogued as one of the leading sources of historical Ethereum exploits[45].

SpendLimitEnforcer: Daily and Per-Transaction Caps The spend-limit enforcer is the most critical piece of the system. It addresses two distinct attack vectors: a single large drain and a slow bleed of many small transactions that collectively exceed what the user intended.

The solution is two independent caps. The per-transaction cap prevents any single call from moving more than a threshold amount. The daily cap resets each epoch (defined as `block.timestamp / 1 days`) and prevents cumulative daily spend from exceeding the limit even if every individual transaction is below the per-tx threshold.

```
function beforeHook(
    bytes calldata terms, bytes32 delegationHash,
    address, address, bytes calldata callData, uint256 value
) external override {
    (address token, uint256 dailyCap, uint256 perTxCap) = abi.decode(terms, (address, uint256,
    uint256 spendAmount = _extractSpendAmount(token, callData, value);
```

[45] Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A Survey of Attacks on Ethereum Smart Contracts (SoK). *POST*. https://doi.org/10.1007/978-3-662-54455-6_8

```
    if (spendAmount > perTxCap) revert CaveatViolation("Per-transaction spend limit exceeded")

    uint256 currentEpoch = block.timestamp / 1 days;
    SpendState storage state = spendState[delegationHash][token];
    if (state.epoch != currentEpoch) {
        state.totalSpent = 0;
        state.epoch = currentEpoch;
    }

    uint256 newTotal = state.totalSpent + spendAmount;
    if (newTotal > dailyCap) revert CaveatViolation("Daily spend limit exceeded");
    state.totalSpent = newTotal;
}
```

The epoch reset is lazy — it happens on the first transaction of a new day, not via a cron or keeper. This keeps gas costs low while guaranteeing that stale daily totals cannot carry forward.

The `token` parameter in the terms handles both native ETH (token address zero) and ERC-20 tokens. Each token tracked by a delegation gets its own independent `SpendState`. A delegation that covers both ETH and USDC has separate daily caps for each.

The Other Four Enforcers `SpendLimitEnforcer` handles the *how much* question. The other four enforcers handle *when*, *where*, *what*, and *how many times*. The decomposition is intentional: each enforcer implements a single, independently testable constraint, matching the *separation of privilege* principle Saltzer and Schroeder articulated[46].

TimeWindowEnforcer restricts execution to a specific time range, encoded as `(uint256 notBefore, uint256 notAfter)` in the terms. An agent session with a 9am-to-5pm window cannot be used at midnight, regardless of how the agent was prompted.

AllowedTargetsEnforcer maintains a whitelist of contract addresses the delegation is permitted to call. If the agent attempts to call a contract not in the whitelist, the `beforeHook` reverts. This prevents a compromised or confused agent from interacting with arbitrary protocols — only the contracts explicitly approved at delegation time are reachable.

AllowedMethodsEnforcer operates at the function-selector level. Even if a target contract is whitelisted, the enforcer checks the first four bytes of calldata against an approved selector list. An agent granted permission to call `transfer()` on a token contract cannot call `approve()` or `transferOwnership()` on the same contract.

NonceEnforcer assigns a monotonically increasing nonce to each execution and re-

[46] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

jects replays. This matters for signed delegation flows where the same delegation hash could theoretically be submitted multiple times.

These five enforcers compose independently. A delegation can use all five, three, or just one. Each enforcer is a separate deployed contract implementing the `ICaveatEnforcer` interface. Adding a new enforcer requires no changes to the WalletDelegate — it just needs a contract address and ABI-encoded terms. The system is fail-closed: an enforcer that does not recognise the calldata should revert rather than silently pass, matching the *fail-safe defaults* principle[47].

ERC-20 Calldata Parsing Tracking ETH spend is straightforward — it is the `value` parameter on the call. Tracking ERC-20 spend requires parsing calldata, because ERC-20 transfers pass the amount as an encoded function argument, not as ETH value.

The `_extractSpendAmount()` function handles the three common ERC-20 transfer patterns by inspecting the first four bytes of calldata (the function selector) and then ABI-decoding the relevant portion of the remaining arguments.

```
function _extractSpendAmount(address token, bytes calldata callData, uint256 value)
    internal pure returns (uint256)
{
    if (token == address(0)) return value;

    if (callData.length >= 68) {
        bytes4 selector = bytes4(callData[:4]);
        if (selector == 0xa9059cbb || selector == 0x095ea7b3) { // transfer, approve
            (, uint256 amount) = abi.decode(callData[4:68], (address, uint256));
            return amount;
        }
        if (selector == 0x23b872dd && callData.length >= 100) { // transferFrom
            (,, uint256 amount) = abi.decode(callData[4:100], (address, address, uint256));
            return amount;
        }
    }
    return value;
}
```

`0xa9059cbb` is `transfer(address,uint256)`. `0x095ea7b3` is `approve(address,uint256)`. `0x23b872dd` is `transferFrom(address,address,uint256)`. The minimum calldata lengths (68 and 100 bytes) guard against malformed input that could cause a decode panic.

If the calldata does not match any recognised selector, the function returns the ETH value — which is zero for a pure ERC-20 call. This is a conservative fallback: an unrecognised ERC-20 call is not counted against the daily limit, which is a known limitation. The alternative — reverting on unrecognised selectors — would break

[47] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

legitimate interactions with non-standard tokens. The trade-off is named explicitly in the implementation so that downstream operators can decide whether their threat model accepts it.

SolverRegistry: Bond Staking and Slashing The caveat enforcers handle what an agent can do. The SolverRegistry handles who is allowed to act as a solver in the first place, and what the economic consequences are for misbehaviour.

Solvers post a bond before they can execute delegated transactions. The bond is not a fixed fee — it scales with volume, requiring 5% of cumulative transaction volume above the base minimum. This means a solver handling large transactions must post proportionally larger bonds, making attacks more expensive as the stakes increase. The bond-and-slash shape is the same economic primitive Buterin and Griffith argued for in Casper FFG[48]: the staked capital is the participant's skin in the game, and the protocol's enforcement actions extract value from that stake when violations occur.

```
uint256 public constant MINIMUM_BOND = 0.1 ether;
uint64  public constant WITHDRAWAL_COOLDOWN = 7 days;
uint8   public constant MAX_JAILS = 3;
uint64  public constant DECAY_HALF_LIFE = 30 days;
uint16  public constant MIN_DECAY_MULTIPLIER_BPS = 1000; // 10% floor
```

The jail mechanism provides escalating consequences. A solver can be jailed up to three times for violations. After the third jail, they are permanently banned from the registry. This creates a three-strikes structure that allows for recovery from errors while enforcing a hard limit on repeated offenders.

When slashing occurs, the bond distribution is: 80% to the affected user, 15% to the challenger who surfaced the violation, and 5% to the protocol treasury. This alignment incentivises external parties to monitor solver behaviour and report violations — the system does not rely solely on protocol-owned monitoring. The structure addresses what the watchtower literature (Pisa[49], Cerberus Channels[50]) treats as the foundational problem: a monitoring role only works if the monitor is paid for catching things, not for showing up.

The reputation decay uses a 30-day half-life with a 10% floor. A solver's reputation score decays over time if they are inactive, preventing indefinitely accumulated reputation from providing a permanent safety buffer. The 10% floor (`MIN_DECAY_MULTIPLIER_BPS = 1000`) means reputation never fully zeroes, preserving some history for long-term participants.

The 7-day withdrawal cooldown prevents a solver from draining their bond immediately after a violation is detected but before a slash transaction is confirmed.

[48] Buterin, V., & Griffith, V. (2017). *Casper the Friendly Finality Gadget.* arXiv:1710.09437.

[49] McCorry, P., Bakshi, S., Bentov, I., Miller, A., & Meiklejohn, S. (2019). Pisa: Arbitration Outsourcing for State Channels. *AFT.* https://doi.org/10.1145/3318041.3355461

[50] Avarikioti, G., Litos, O. S. T., & Wattenhofer, R. (2020). Cerberus Channels: Incentivizing Watchtowers for Bitcoin. *Financial Cryptography and Data Security (FC).* https://doi.org/10.1007/978-3-030-51280-4_19

Deployed Contracts on Sepolia Testnet All eleven protocol contracts are deployed on Sepolia. These are testnet deployments — no mainnet deployment exists yet.

Contract	Address	Explorer
SolverRegistry	0xB6ab964832808E49635fF8[illegible]cB745	Sepolia Etherscan
IntentReceiptHub	0xD66A1e880AA3939CA066[illegible]D977c	Sepolia Etherscan
DisputeModule	0x144DfEcB57B08471e2A75[illegible]DB79D	Sepolia Etherscan
ERC-8004 Registry	0x8004A818BFB912233c491[illegible]4BD9e	Sepolia Etherscan
IRSB Agent ID	#967	8004scan

The BUSL-1.1 license on the protocol contracts converts to MIT on 2029-02-17.

The Competitive Gap The AI agent wallet space has accumulated a number of frameworks over the past two years, but none have shipped on-chain enforcement. The comparison is not about capability — AgentKit, ElizaOS, Olas, Virtuals, Brian AI, and Safe are all competent systems in their respective lanes. The gap is about what happens when something goes wrong.

Framework	On-Chain Spend Limits	Execution Receipts	Active Monitoring	Dispute Resolution
AgentKit (Coinbase)	No	No	No	No
ElizaOS	No	No	No	No
Olas	No	No	Partial	No
Virtuals Protocol	No	No	No	No
Brian AI	No	No	No	No
Safe (with modules)	Partial (manual)	No	No	No
IRSB	Yes (5 enforcers)	Yes (IntentReceiptHub)	Yes (Watchtower)	Yes (DisputeModule)

Safe with custom modules can approximate some spend limiting, but it requires writing and deploying a module per use case and there is no standardised caveat interface. The other frameworks do not have on-chain enforcement at all — policy lives in the agent's system prompt or in off-chain configuration files that the model can read but cannot be forced to obey.

The structural difference is simple: IRSB limits are enforced by the EVM. The others rely on the model honouring a policy document it was given as context — a design

posture the agent-safety literature[51], [52] has demonstrated to be fragile in any setting where the agent processes untrusted input.

What Comes Next This post covered the enforcement layer — the contracts that run before and after every delegated call. The remaining parts of this series cover how the rest of the stack fits together:

- **Part 2: The Evidence Pipeline** — how IntentReceiptHub creates tamper-evident execution records, what the receipt schema captures, and how receipts feed the dispute-resolution process.
- **Part 3: The Watchtower Architecture** — the off-chain monitoring system that watches on-chain events, scores solver behaviour, and triggers alerts before slashing is warranted.
- **Part 4: Z3, the Three-Layer Stack, and Claude Code as Architect** — formal verification of caveat logic using Z3, how the three layers (enforcement, evidence, monitoring) compose into a coherent safety system, and what it looks like to build protocol infrastructure with Claude Code as the primary development tool.

The core thesis across all four parts is the same: agents with wallet access need structural safety, not honour systems. Structural safety means the constraints exist independently of the model, are enforced before execution, and cannot be bypassed by prompt manipulation. Every other approach — system prompts, off-chain policies, trust-the-model — is an honour system. The agent-security literature is consistent that honour systems fail when the stakes are high enough[53], [54].

Part of the IRSB Ecosystem deep dive series. Built with Claude Code.

---

[51] Yi, S., Liu, Y., Sun, Z., Cong, T., He, X., Song, J., Xu, K., & Li, Q. (2024). Jailbreak Attacks and Defenses Against Large Language Models: A Survey. arXiv:2407.04295.

[52] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *AISec '23*. https://doi.org/10.1145/3605764.3623985

[53] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *AISec '23*. https://doi.org/10.1145/3605764.3623985

[54] Yi, S., Liu, Y., Sun, Z., Cong, T., He, X., Song, J., Xu, K., & Li, Q. (2024). Jailbreak Attacks and Defenses Against Large Language Models: A Survey. arXiv:2407.04295.

Chapter 3. Deep Dive Part 2: Cryptographic Receipts and the Evidence Pipeline That Proves What AI Agents Actually Did

"Trust me, the AI did good work." That is the state of most AI-agent frameworks today. The agent completes a task, the logs say it finished, and the operator is left reconstructing what actually happened from console output and hope. There is no cryptographic commitment. There is no unforgeable record. There is nothing to challenge.

The IRSB Ecosystem is built on the premise that claimed work and proved work are not the same thing. The evidence pipeline is the mechanism that closes the gap. It does not ask the verifier to trust the solver. It asks the verifier to check the hash. The design lineage is direct: Schneier and Kelsey's 1999 paper on secure audit logs for computer forensics[55] and Haber and Stornetta's 1991 digital-document time-stamping scheme[56] together establish the canonical posture — integrity that survives an untrusted environment because it is structurally tamper-evident, not because the environment is trusted.

This is Part 2 of the IRSB Ecosystem Deep Dive series. Part 1 covered the on-chain enforcement layer — the five enforcers, the solver registry, and the bond mechanics that make violations expensive. This part covers what happens between intent arrival and the receipt that lands on-chain: the policy gate, the evidence bundle, the Cloud KMS signature, and the replay-protected receipt posting.

The Solver is code-complete at v0.3.0 with 139 tests. It is not yet deployed to production infrastructure.

From Intent to Receipt Every piece of work in IRSB starts as an intent: a normalised structure describing what needs to happen, who requested it, when it expires, and what inputs to operate on. The solver receives the intent and processes it through a linear pipeline before anything reaches the chain. The split between on-chain commitments and off-chain execution follows the design pattern Eberhardt and Tai articulated for off-chaining computation while anchoring proofs on-chain[57]: do the expensive work where compute is cheap, and put only the *evidence* on the chain.

The pipeline has five stages:

1. **Policy gate** — four deterministic checks decide whether the intent is allowed to execute at all.
2. **Execution** — the solver performs the actual work.
3. **Evidence-bundle creation** — every output artefact is SHA-256 hashed. A canonical manifest is assembled and hashed.

[55] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM TISSEC*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

[56] Haber, S., & Stornetta, W. S. (1991). How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2), 99–111. https://doi.org/10.1007/BF00196791

[57] Eberhardt, J., & Tai, S. (2017). On or Off the Blockchain? Insights on Off-Chaining Computation and Data. *ESOCC*. https://doi.org/10.1007/978-3-319-67262-5_1

4. **Cloud KMS signing** — the manifest hash is signed using a hardware-backed key that never leaves Google's infrastructure.
5. **On-chain receipt posting** — the signature and hashes are posted to `IntentReceiptHub` with replay protection.

The key insight is that the evidence bundle is created *before* the signature. The solver cannot sign a favourable summary of what happened. It signs a hash of the actual artefacts — every file, every output — as they exist on disk. If the artefacts are altered after signing, the hash mismatch is immediately detectable by anyone who re-hashes the outputs. This is the same integrity property Schneier and Kelsey identified as load-bearing for forensic audit logs[58].

The Policy Gate: Four Checks Before any work begins, the solver runs `evaluatePolicy()`. The function is deliberately simple: four independent checks, all reasons accumulated, no early returns. If any check fails, the intent is rejected with the full list of failure reasons — not just the first one found.

```
export function evaluatePolicy(
  intent: NormalizedIntent,
  config: ResolvedConfig
): PolicyResult {
  const reasons: string[] = [];

  // Check 1: jobType allowlisted
  if (!config.POLICY_JOBTYPE_ALLOWLIST.includes(intent.jobType)) {
    reasons.push(`jobType '${intent.jobType}' not in allowlist`);
  }

  // Check 2: expiresAt not in the past
  if (intent.expiresAt) {
    const expiresAt = new Date(intent.expiresAt);
    if (expiresAt.getTime() < Date.now()) {
      reasons.push(`intent expired at ${intent.expiresAt}`);
    }
  }

  // Check 3: requester allowlist (if configured)
  if (config.POLICY_REQUESTER_ALLOWLIST) {
    if (!config.POLICY_REQUESTER_ALLOWLIST.includes(intent.requester)) {
      reasons.push(`requester '${intent.requester}' not in allowlist`);
    }
  }

  // Check 4: size guard
```

[58] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM TISSEC*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

```
  const inputsJson = canonicalJson(intent.inputs);
  const maxBytes = config.POLICY_MAX_ARTIFACT_MB * 1024 * 1024;
  if (inputsJson.length > maxBytes) {
    reasons.push(`inputs size ${inputsJson.length} exceeds max ${maxBytes}`);
  }

  return { allowed: reasons.length === 0, reasons };
}
```

The four checks cover the most common failure modes in agent work:

- **jobType allowlist** — the solver only processes work it explicitly knows how to do. An unknown job type is rejected, not attempted and failed.
- **expiry check** — stale intents are rejected at the gate. A solver should not act on instructions issued for a time window that has already passed.
- **requester allowlist** — optional, but when configured, it prevents intents from unauthorised sources from entering the execution pipeline at all.
- **size guard** — a ceiling on serialised input size prevents resource exhaustion and makes inputs auditable. If inputs are too large to hash in bounded time, the pipeline should not accept them.

Accumulating all failure reasons rather than short-circuiting is a deliberate design choice. Callers get a complete diagnostic in one pass, which matters when debugging why an intent was rejected in a production pipeline.

Evidence Bundle Creation Once the solver completes work, `createEvidenceBundle()` scans the output directory, hashes every artefact, assembles a manifest, and hashes the manifest itself. The manifest is the single source of truth for what the solver produced.

```
export async function createEvidenceBundle(
  params: CreateEvidenceBundleParams
): Promise<EvidenceBundleResult> {
  const { runDir, intentId, runId, jobType, policyDecision, executionSummary, gitCommit } = pa

  const artifacts = await scanArtifacts(runDir);

  const manifest: EvidenceManifestV0 = {
    manifestVersion: MANIFEST_VERSION,
    intentId, runId, jobType,
    createdAt: new Date().toISOString(),
    artifacts,
    policyDecision,
    executionSummary,
    solver: { service: "irsb-solver", serviceVersion: SERVICE_VERSION, gitCommit },
  };

  const manifestSha256 = computeManifestHash(manifest);
```

```
  const evidenceDir = join(runDir, "evidence");
  ensureDir(evidenceDir);
  atomicWrite(join(evidenceDir, "manifest.json"), canonicalJson(manifest) + "\n");
  atomicWrite(join(evidenceDir, "manifest.sha256"), manifestSha256 + "\n");

  return { manifest, manifestPath: join(evidenceDir, "manifest.json"), manifestSha256 };
}
```

Three implementation details matter here:

Canonical JSON. Standard `JSON.stringify()` does not guarantee key ordering across environments or runtimes. `canonicalJson()` produces a deterministically ordered serialisation. The same manifest data always produces the same bytes, which always produces the same SHA-256 hash. Without this, a manifest serialised on one machine might hash differently on another — breaking verification. The general lesson Haber and Stornetta named[59]: a digital integrity scheme that depends on serialisation order has a covert weakness, because two parties may compute the same logical content into different bytes.

Path-sorted artefact entries. The `scanArtifacts()` function returns entries sorted by file path. This ensures that adding or removing files changes the manifest hash in a detectable way, and that the artefact list is stable regardless of filesystem enumeration order.

Atomic writes. The manifest and its hash file are written atomically. An observer cannot read a partially written manifest and compute a hash that does not match the final file.

The `policyDecision` field is included in the manifest. This means the policy-gate outcome — every rejection reason, or the explicit allowed signal — is part of the signed artefact. A receipt therefore commits not just to what work was done, but to the fact that the policy gate was passed.

Cloud KMS Signing The manifest hash is signed using Google Cloud KMS rather than a local private key. This choice is not arbitrary.

Local private keys are files on disk. They can be copied, stolen, or leaked. A solver operator who wants to forge a receipt can do so if they control the signing key. Cloud KMS stores keys in hardware security modules. The private-key material never leaves Google's infrastructure. The operator can request a signature but cannot extract the key.

The practical consequence: a signature produced by Cloud KMS proves that the signing request was made by an authorised IAM principal at a specific point in time, and that the key was not compromised. GCP audit logs record every signing operation.

[59] Haber, S., & Stornetta, W. S. (1991). How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2), 99–111. https://doi.org/10.1007/BF00196791

The signing authority is traceable. This is precisely the *integrity grounded in an external trust anchor* posture Schneier and Kelsey identified as essential for forensic audit systems[60].

The signing flow works as follows: the manifest hash is passed to the KMS `asymmetricSign` API, which returns a DER-encoded signature. DER is the standard encoding for ECDSA signatures from hardware systems, but Ethereum expects `r`, `s`, and `v` components. The solver parses the DER, normalises `s` to its low form, and computes the recovery parameter.

```
function parseDerSignature(der: Buffer): { r: bigint; s: bigint } {
  if (der[0] !== 0x30) throw new Error(`Invalid DER: expected 0x30`);
  let offset = 2;

  if (der[offset] !== 0x02) throw new Error('Invalid DER: expected 0x02 for r');
  offset++;
  const rLen = der[offset]!; offset++;
  const rBytes = der.subarray(offset, offset + rLen); offset += rLen;

  if (der[offset] !== 0x02) throw new Error('Invalid DER: expected 0x02 for s');
  offset++;
  const sLen = der[offset]!; offset++;
  const sBytes = der.subarray(offset, offset + sLen);

  return {
    r: BigInt(`0x${Buffer.from(rBytes).toString('hex')}`),
    s: BigInt(`0x${Buffer.from(sBytes).toString('hex')}`),
  };
}
```

The DER structure is straightforward: a `0x30` sequence header, followed by two ASN.1 integers for `r` and `s`. Each integer is prefixed with `0x02` and a length byte. The parser reads these fields sequentially, handling the variable-length encoding correctly.

EIP-2 Low-S Normalisation After parsing `r` and `s` from the DER signature, the solver normalises `s` to its low form. This step is required for Ethereum compatibility, and it carries a security property worth naming explicitly.

The secp256k1 curve has a symmetry property: for any signature `(r, s)`, the value `(r, curve_order - s)` is also a valid signature for the same message and key. This means every signature has two valid representations. Without normalisation, the same signing operation produces different bytes depending on which representation the signing hardware returns. Applications that index or deduplicate by signature bytes would treat them as different signatures.

More importantly, transaction-malleability exploits rely on this property. A malicious

[60] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM TISSEC*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

relay can mutate a signature from its high-S form to its low-S form (or vice versa) without invalidating it, changing the transaction ID without changing what the transaction does. EIP-2[61] (and Bitcoin's BIP-62[62]) eliminate this by requiring `s <= curve_order / 2`. The malleability class is one of the failure modes Atzei and colleagues catalogued in their systematisation of attacks on Ethereum smart contracts[63]; the fix at the encoding layer is the canonical defence.

```
const SECP256K1_N = BigInt('0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
const SECP256K1_HALF_N = SECP256K1_N / 2n;

// In signHashComponents():
const { r, s: rawS } = parseDerSignature(sigBytes);
const sNormalized = rawS > SECP256K1_HALF_N;
const s = sNormalized ? SECP256K1_N - rawS : rawS;
const v = await this.computeRecoveryV(digestBuffer, r, s, sNormalized);
```

If `rawS` is greater than half the curve order, the solver flips it: `s = curve_order - rawS`. The boolean `sNormalized` records whether the flip occurred, which is needed to compute the correct recovery parameter `v`. The recovery parameter tells the verifier which of the two possible public keys corresponds to the private key that produced this signature — it is the tiebreaker that makes `ecrecover` deterministic.

On-Chain Receipt Posting With a valid signature in hand, the solver posts a receipt to `IntentReceiptHub`. A receipt is not a log entry — it is a permanent, challengeable record that commits the solver to specific claims about what work was done.

```
function postReceipt(Types.IntentReceipt calldata receipt, uint256 declaredVolume)
    external whenNotPaused nonReentrant returns (bytes32 receiptId)
{
    // Validate solver is active and has sufficient bond
    Types.Solver memory solver = solverRegistry.getSolver(receipt.solverId);
    if (solver.status != Types.SolverStatus.Active) revert InvalidSolver();

    // PM-EC-001: Bond must cover declared volume
    uint256 requiredBond = solverRegistry.requiredBondForVolume(declaredVolume);
    if (solver.bondBalance < requiredBond) revert InsufficientBondForVolume();

    receiptId = computeReceiptId(receipt);
    if (_receipts[receiptId].exists) revert ReceiptAlreadyExists();

    // IRSB-SEC-006: Replay protection with chainId + contract address + nonce
    bytes32 messageHash = keccak256(abi.encode(
```

[61]Wood, G., & Reitwiessner, C. (2015). *EIP-2: Homestead Hard-fork Changes.* https://eips.ethereum.org/EIPS/eip-2

[62]Wuille, P. (2014). *BIP-62: Dealing with Malleability.* https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki

[63]Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A Survey of Attacks on Ethereum Smart Contracts (SoK). *POST.* https://doi.org/10.1007/978-3-662-54455-6_8

```
        block.chainid, address(this), currentNonce,
        receipt.intentHash, receipt.constraintsHash, receipt.routeHash,
        receipt.outcomeHash, receipt.evidenceHash,
        receipt.createdAt, receipt.expiry, receipt.solverId
    ));
    address signer = messageHash.toEthSignedMessageHash().recover(receipt.solverSig);
    if (signer != solver.operator) revert InvalidReceiptSignature();

    _receipts[receiptId] = receipt;
    _receiptStatus[receiptId] = Types.ReceiptStatus.Pending;
    // ... indexing, nonce increment, event emission
}
```

Several security properties are encoded in the message hash:

Chain ID (IRSB-SEC-001). Including `block.chainid` in the signed message means a signature produced for Sepolia cannot be replayed on mainnet. This is elementary but critical — without it, a valid testnet receipt becomes a valid mainnet receipt.

Contract address. Including `address(this)` means a signature for one deployment of `IntentReceiptHub` cannot be replayed against a different deployment. Upgrading the contract address invalidates all prior signatures.

Nonce (IRSB-SEC-006). Including `currentNonce` means each receipt posting is unique even if the same intent hash, evidence hash, and outcome hash appear again. The nonce is incremented after each successful posting. These three replay defences together — chain identity, contract identity, and per-instance nonce — close the standard class of cross-context replay attacks Atzei and colleagues documented for Ethereum smart contracts[64].

The bond check enforces a key invariant: the solver must have posted a bond sufficient to cover the declared work volume before the receipt is accepted. If a solver's bond balance has been slashed below the required threshold, they cannot post new receipts until they re-stake. This is the economic coupling that makes receipts meaningful — posting a receipt is an assertion backed by locked capital. The literature on MEV and frontrunning (Daian and colleagues' Flash Boys 2.0[65]) makes the case that economic security and protocol security are inseparable in decentralised settings; IRSB's bond coupling is the same principle applied to AI-agent accountability.

The `IntentReceiptHub` is deployed on Sepolia testnet at `0xD66A1e880AA3939CA066a9EA1dD37ad3d01D977c`.

Challenge, Dispute, and Finalisation A receipt does not become final the moment it is posted. It enters a one-hour challenge window during which anyone can dispute it. This is the mechanism that makes receipts more than self-reported claims.

[64] Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A Survey of Attacks on Ethereum Smart Contracts (SoK). *POST.* https://doi.org/10.1007/978-3-662-54455-6_8

[65] Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., & Juels, A. (2020). Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. *IEEE S&P.* https://doi.org/10.1109/SP40000.2020.00040

The lifecycle has four states:

- **Pending** — receipt posted, challenge window open (default: 1 hour).
- **Disputed** — a challenger has posted a bond and raised a dispute.
- **Finalised** — challenge window elapsed with no dispute, or a dispute was resolved in the solver's favour. The solver's reputation score increases.
- **Slashed** — a dispute was resolved against the solver. The solver's bond is slashed in proportion to the violation; the challenger receives a bounty.

The `DisputeModule` handles two resolution paths: deterministic and arbitrated. Deterministic resolution covers cases where the outcome can be computed from on-chain data alone — a timeout (the challenge window elapsed), an invalid signature (the evidence hash does not match the submitted artefacts), or a replay (the same receipt hash appears twice). These cases resolve without human involvement.

For disputes that cannot be resolved deterministically — a challenger claims the work was wrong but the hash is valid — the dispute escalates to an arbitration pool. The arbitrators review the evidence bundle off-chain, post their determination on-chain, and the smart contract enforces the outcome.

This architecture separates two concerns that most systems conflate: verifying that the work was committed to (cryptographic) and verifying that the committed work was correct (judgment). The evidence pipeline handles the first problem completely. The dispute module handles the second. The same separation is what motivates Kosba and colleagues' Hawk design[66]: the cryptographic substrate establishes what was *claimed*, and a separate judgment layer adjudicates whether the claim was *correct*.

Why This Matters Any AI-agent framework can log "task completed." The log entry says the work happened. It does not prove what work happened, what outputs were produced, or that the agent reporting completion is the same agent that performed the work.

The IRSB evidence pipeline addresses a different question: not whether work was done, but what exactly was done and who can be held accountable if it was wrong.

When a solver posts a receipt, it is not logging a claim. It is cryptographically committing to a SHA-256 hash of every artefact it produced. That commitment is signed by a Cloud KMS key whose usage is audit-logged by Google. That signature is posted on-chain with replay protection, bonded capital as a stake, and a challenge window. Anyone can re-hash the artefacts and verify that the on-chain hash matches. Anyone can check the Sepolia transaction. No permissions required.

The provenance chain is complete: intent arrives, policy gate passes, work executes, artefacts are hashed, manifest is signed with a hardware-backed key, receipt goes on-chain. At each step, the prior step is committed to. Altering any part of the chain — the artefacts, the manifest, the signature — breaks the next link. This is the structural

[66] Kosba, A., Miller, A., Shi, E., Wen, Z., & Papamanthou, C. (2016). Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. *IEEE S&P*. https://doi.org/10.1109/SP.2016.55

property the Schneier–Kelsey audit-log scheme makes explicit[67]: tamper-evidence is achieved by chaining commitments, not by trusting any single storage tier.

The ecosystem is code-complete and not yet in production. The contracts are on Sepolia. The Solver has 139 passing tests. The Watchtower (Part 3) uses mock chain data while real IRSB client integration is pending. The infrastructure is not yet live, but the cryptographic design is fully specified and tested.

What Comes Next This series has four parts:

- Part 1: Five On-Chain Enforcers That Make AI Agent Wallets Structurally Safe — the enforcement contracts, the bond mechanics, and the three-strikes jail.
- **Part 2: Cryptographic Receipts and the Evidence Pipeline** (this post) — the solver's policy gate, SHA-256 evidence bundles, Cloud KMS signing, and on-chain receipt posting.
- Part 3: A 12-Package Nested Monorepo That Watches AI Agents for You — the Watchtower's architecture, its ten behaviour signals, the risk-scoring engine, and the auto-dispute pipeline.
- Part 4: Z3 Formal Verification, the Three-Layer Stack, and Claude Code as Architect — the FormalAgentVerifier, Scout's brokering layer, and what it looks like to build a protocol-layer system collaboratively with an AI.

The underlying thesis is that agents with economic agency require economic accountability. Claimed work is worthless at protocol scale. Proved work — hashed, signed, bonded, and challengeable — is the foundation that makes agentic commerce viable.

Part of the IRSB Ecosystem deep dive series. Built with Claude Code.

---

[67] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM TISSEC*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

Chapter 4. Deep Dive Part 3: A 12-Package Nested Monorepo That Watches AI Agents for You

Posting an on-chain receipt for a completed AI-agent job is not the same as monitoring the receipt. Receipts are inert data. They do not check themselves for hash mismatches, verify that artefact files actually exist, or notice when an agent's behavioural profile starts drifting toward anomalous territory. They certainly do not file a dispute within the 1-hour challenge window when something looks wrong.

That gap — between *we have on-chain evidence* and *we are actively verifying it and acting on it* — is exactly the space the IRSB Watchtower occupies. It is a fully autonomous monitoring system that watches agent activity, verifies evidence integrity, derives behavioural signals, computes risk scores, and triggers disputes without human intervention. The role itself is well-established in the off-chain-protocols literature: McCorry and colleagues introduced *Pisa* as the canonical watchtower design for state channels[68]; Avarikioti and colleagues' *Cerberus Channels*[69] and Khabbazian and colleagues' *Outpost*[70] refined the incentive design and latency tradeoffs respectively. The IRSB Watchtower applies the same architectural pattern to AI-agent receipts rather than payment-channel state.

At v0.5.0 it has approximately 500 tests covering its deterministic logic. The chain integration uses mock data while real IRSB client wiring is completed, but all the monitoring logic itself is production-grade.

This is Part 3 of the IRSB Ecosystem Deep Dive series. Part 1 covered on-chain enforcement and the contract architecture. Part 2 covered the evidence pipeline from submission to verification.

Why a Watchtower The protocol layer handles enforcement mechanically: bonds are locked, receipts are posted, disputes can be opened. But the protocol does not know whether a receipt is fraudulent. It cannot tell whether an artefact hash was fabricated, whether a manifest was tampered with, or whether an agent has been exhibiting a slow pattern of small violations that individually fall below alert thresholds. This is exactly the gap the payment-channel watchtower literature identified[71], [72]: the on-chain protocol enforces *resolution* against evidence, but it does not produce the evidence — somebody has to.

Someone needs to know. And that someone needs to act within time constraints. The

[68] McCorry, P., Bakshi, S., Bentov, I., Miller, A., & Meiklejohn, S. (2019). Pisa: Arbitration Outsourcing for State Channels. *AFT.* https://doi.org/10.1145/3318041.3355461

[69] Avarikioti, G., Litos, O. S. T., & Wattenhofer, R. (2020). Cerberus Channels: Incentivizing Watchtowers for Bitcoin. *Financial Cryptography and Data Security (FC)*. https://doi.org/10.1007/978-3-030-51280-4_19

[70] Khabbazian, M., Nadahalli, T., & Wattenhofer, R. (2019). Outpost: A Responsive Lightweight Watchtower. *AFT.* https://doi.org/10.1145/3318041.3355464

[71] Poon, J., & Dryja, T. (2016). *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. https://lightning.network/lightning-network-paper.pdf

[72] Decker, C., & Wattenhofer, R. (2015). A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. *SSS*. https://doi.org/10.1007/978-3-319-21741-3_1

IRSB challenge window is one hour. If a fraudulent receipt goes unchallenged in that window, it is accepted and the bond is unlocked. A human operator watching a dashboard is not a reliable system. Sleep, context switching, alert fatigue — any of these can cause a missed challenge. McCorry and colleagues argued the same point for state-channel watchtowers[73]: the human-in-the-loop variant of watchtower duty does not survive contact with real adversarial conditions.

The Watchtower runs a continuous scan cycle. It fetches the current block, creates a chain context, runs every registered rule against that context, collects findings, and either simulates or executes the resulting actions. When a CRITICAL risk signal is detected, the score is automatically set to 100 and the dispute path is triggered. The operator is notified via webhook. The challenge is filed on-chain.

The key design constraint is that all of this must be deterministic and auditable. Every finding, every action, every ledger entry is persisted in JSONL format so the reasoning behind any dispute can be reconstructed from first principles — the same hash-chained, tamper-evident posture Schneier and Kelsey identified as essential for forensic audit logs[74].

12-Package Architecture The Watchtower lives in a nested pnpm monorepo with nine packages and three application entrypoints. The dependency graph is intentional: application layers only depend downward into the infrastructure packages, never sideways. The information-distribution discipline behind this layering goes back to Parnas's 1971 paper on module decomposition[75]: each package hides decisions the other packages do not need to know.

```
apps/worker → core, config, chain, irsb-adapter, evidence-store, metrics, webhook
apps/api    → core, config, chain, irsb-adapter, metrics, signers
apps/cli    → config, chain, irsb-adapter
irsb-adapter → config, resilience
```

The three apps are thin orchestration layers. The real logic lives in the packages. This separation means the rule engine, evidence verification, and risk scoring are all independently testable without spinning up a node connection or a Fastify server.

The nine packages break down by responsibility:

- **core** — the rule engine, Finding schema, and ActionExecutor. Zero cloud dependencies. This package runs identically in test and production.
- **config** — Zod schemas for all configuration. Nothing touches environment variables without going through here first.
- **chain** — a viem abstraction layer. All block fetching, transaction submission, and event reading is isolated here. Mock implementations swap in for tests.

[73] McCorry, P., Bakshi, S., Bentov, I., Miller, A., & Meiklejohn, S. (2019). Pisa: Arbitration Outsourcing for State Channels. *AFT.* https://doi.org/10.1145/3318041.3355461

[74] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM TISSEC*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

[75] Parnas, D. L. (1971). Information Distribution Aspects of Design Methodology. *Proceedings of the IFIP Congress.* https://doi.org/10.1184/R1/6606470.V1

- **irsb-adapter** — the contract client for the IRSB protocol. Currently wraps mock chain data while real IRSB client integration is pending.
- **evidence-store** — JSONL persistence for findings and action-ledger entries. Append-only, deterministic, readable with any text tool.
- **metrics** — Prometheus instrumentation. Scan-cycle duration, finding counts by severity, action outcomes.
- **webhook** — HMAC-signed delivery of findings and alerts to operator endpoints.
- **resilience** — retry logic with exponential backoff and a circuit breaker for chain RPC calls. The patterns are the canonical Byzantine-fault-tolerance toolkit (Castro and Liskov's PBFT[76] established the architecture-level argument; the system-engineering form is application-level retry + isolation).
- **signers** — pluggable signing backends. `LocalPrivateKey` for development and testing. `CloudKMSSigner` is scaffolded but cloud integration is pending — all current deployments use the local key path.

The three application packages are `watchtower-api` (a Fastify REST API for querying findings and agent risk reports), `watchtower-cli` (utilities for inspecting the evidence store and triggering manual scans), and `watchtower-core` (the worker process that drives the scan loop).

This architecture pays for itself in testability. The ~500 tests can exercise `verifyEvidence()`, `scoreAgent()`, and `RuleEngine.execute()` without any network calls because chain and contract dependencies are injected interfaces with mock implementations.

Evidence Verification When the Watchtower processes a receipt, it does not take the claimed hash values at face value. It re-derives them. The `verifyEvidence()` function is a sequential pipeline of six checks, each with a specific failure code. The first failure short-circuits the pipeline to avoid misleading downstream results. The methodology — *re-derive rather than trust* — is the same one Haber and Stornetta's time-stamping scheme established as the integrity primitive for digital documents[77].

```
export function verifyEvidence(
  receipt: NormalizedReceipt,
  runDir: string,
  options?: VerifyOptions,
): VerificationResult {
  const failures: VerificationFailure[] = [];
  const evidenceLinks: EvidenceLink[] = [
    { type: 'receiptId', ref: receipt.receiptId },
    { type: 'manifestSha256', ref: receipt.manifestSha256 },
  ];
```

[76] Castro, M., & Liskov, B. (2002). Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM TOCS*, 20(4), 398–461. https://doi.org/10.1145/571637.571640

[77] Haber, S., & Stornetta, W. S. (1991). How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2), 99–111. https://doi.org/10.1007/BF00196791

```
  // 1. Path safety - block traversal, null bytes, absolute paths
  const pathCheck = validateRelativePath(receipt.manifestPath);
  if (!pathCheck.valid) {
    failures.push({ code: 'UNSAFE_PATH', message: `Manifest path unsafe: ${pathCheck.reason}` }
    return buildResult(failures, evidenceLinks);
  }

  // 2. Manifest exists and within size limit
  // 3. SHA-256 hash matches receipt claim
  // 4. JSON parse + Zod schema validation
  // 5. delivered[] matches manifest artifacts
  // 6. Each artifact: path safe, exists, size matches, hash matches

  return buildResult(failures, evidenceLinks);
}
```

The path-safety check deserves emphasis. Before any file is read, the path is validated against three attack patterns: directory traversal sequences (`../` and encoded variants), null bytes, and absolute paths. A receipt that claims its manifest lives at `../../../../etc/passwd` should not trigger a filesystem read. It should produce an `UNSAFE_PATH` failure immediately.

The twelve failure codes map precisely to where in the pipeline the verification broke down:

Code	Stage
`UNSAFE_PATH`	Path safety validation
`MANIFEST_NOT_FOUND`	File existence check
`MANIFEST_TOO_LARGE`	Size limit enforcement
`MANIFEST_READ_ERROR`	Filesystem I/O
`MANIFEST_PARSE_FAIL`	JSON deserialization
`MANIFEST_SCHEMA_INVALID`	Zod schema validation
`MANIFEST_HASH_MISMATCH`	SHA-256 re-derivation
`DELIVERED_MISMATCH`	Artifact list reconciliation
`ARTIFACT_NOT_FOUND`	Artifact file existence
`ARTIFACT_TOO_LARGE`	Artifact size limit
`ARTIFACT_SIZE_MISMATCH`	Declared vs actual size
`ARTIFACT_HASH_MISMATCH`	Artifact SHA-256 re-derivation

A `MANIFEST_HASH_MISMATCH` means the manifest file exists and is valid JSON but the content no longer matches what the solver claimed when posting the receipt. This is either tampering or a re-run that overwrote the evidence directory. Either way, the Watchtower opens a finding.

Behaviour Signal Derivation Evidence verification catches integrity failures on individual receipts. Behaviour-signal derivation operates at a higher level: it watches

the pattern of an agent's activity over time and flags anomalies that would not be visible in any single receipt.

The system derives ten signals from the combination of receipts, evidence-verification results, and chain events. Each signal carries a severity level and a weight. The severity levels drive automated-response thresholds:

```
const SEVERITY_POINTS: Record<string, number> = {
  LOW: 5,
  MEDIUM: 15,
  HIGH: 30,
  CRITICAL: 60,
};
```

A `CRITICAL` signal does not feed into the score calculation. It overrides it. Any CRITICAL signal in a snapshot window sets the overall risk to 100 regardless of math. This is a deliberate choice: certain behaviours (fabricated hashes, challenge-window violations, bond manipulation) are not *high risk* — they are categorical failures that require immediate action. The override pattern echoes the *fail-stop* discipline from the Byzantine-fault-tolerance literature[78], [79]: certain classes of detected misbehaviour must terminate normal processing immediately rather than be averaged into a continuous score.

`LOW` signals are informational. An agent that occasionally runs close to its time budget or posts receipts slightly late in a window accumulates LOW signals. These do not trigger automated disputes but they do appear in the risk report and contribute to the composite score. A pattern of consistent LOW signals can push a score into the range that triggers an alert even without any individual HIGH or CRITICAL finding.

The weight field on each signal handles the case where multiple signals of the same type fire in a short window. Rather than counting each occurrence as a fresh independent data point, the weighting system allows the scoring function to treat correlated signals with appropriate scepticism about their independence.

Risk Scoring The `scoreAgent()` function takes an agent record, a set of behavioural snapshots, and a timestamp. It produces a `RiskReport` and any new `Alert` objects that should be delivered to the operator.

```
export function scoreAgent(
  agent: Agent, snapshots: Snapshot[], generatedAt: number,
): { report: RiskReport; newAlerts: Alert[] } {
  const allSignals = snapshots.flatMap((snap) => snap.signals);

  let rawScore = 0;
  let hasCritical = false;
```

[78] Lamport, L., Shostak, R., & Pease, M. (1982). The Byzantine Generals Problem. *ACM TOPLAS*, 4(3), 382–401. https://doi.org/10.1145/357172.357176

[79] Castro, M., & Liskov, B. (2002). Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM TOCS*, 20(4), 398–461. https://doi.org/10.1145/571637.571640

```
  for (const signal of allSignals) {
    const points = SEVERITY_POINTS[signal.severity] ?? 0;
    rawScore += points * signal.weight;
    if (signal.severity === 'CRITICAL') hasCritical = true;
  }

  const overallRisk = hasCritical ? 100 : Math.min(100, Math.round(rawScore));

  // Confidence: HIGH if 5+ signals from 2+ snapshots
  // Alerts: CRITICAL_SIGNAL_DETECTED or HIGH_RISK_SCORE
  // Report ID: SHA-256 of canonical JSON payload
  // ...
}
```

The confidence calculation addresses a different problem: how much should the operator trust a risk score derived from limited data? A score of 75 computed from 12 signals across 4 scan windows is more reliable than the same score computed from 2 signals in a single window. The confidence level — `LOW`, `MEDIUM`, or `HIGH` — is surfaced in the report so operators can contextualise the score without digging into raw signal data. The reasoning matches the watchtower-incentive literature on how to express uncertainty about behaviour observed from a partial view[80], [81].

Two alert types exist. `CRITICAL_SIGNAL_DETECTED` fires on any snapshot containing a CRITICAL severity signal. `HIGH_RISK_SCORE` fires when the numeric score reaches 80 or above, even if no individual signal is CRITICAL. The alert path triggers the webhook sink, which signs the payload with HMAC and delivers it to the configured operator endpoint.

The report ID is a SHA-256 hash of the canonical JSON payload (deterministic key ordering, no whitespace). This means if the same agent produces the same signals with the same weights, it produces the same report ID. Idempotency is enforced at the action-ledger level using this ID.

The IRSBHook: EIP-8183 Bridge The contracts layer exposes `IRSBHook`, which bridges the EIP-8183 Agentic Commerce Protocol job lifecycle into the IRSB accountability pipeline. Every significant job event — acceptance, result submission, completion, rejection — triggers a corresponding IRSB operation automatically.

```
contract IRSBHook is IACPHook {
    function afterAction(uint256 jobId, TypesACP.Action action, address caller) external overri
        if (action == TypesACP.Action.AcceptJob) _afterAcceptJob(jobId, caller);
        else if (action == TypesACP.Action.SubmitResult) _afterSubmitResult(jobId, caller);
        else if (action == TypesACP.Action.CompleteJob) _afterCompleteJob(jobId);
```

[80] McCorry, P., Bakshi, S., Bentov, I., Miller, A., & Meiklejohn, S. (2019). Pisa: Arbitration Outsourcing for State Channels. *AFT*. https://doi.org/10.1145/3318041.3355461

[81] Avarikioti, G., Litos, O. S. T., & Wattenhofer, R. (2020). Cerberus Channels: Incentivizing Watchtowers for Bitcoin. *Financial Cryptography and Data Security (FC)*. https://doi.org/10.1007/978-3-030-51280-4_19

```
        else if (action == TypesACP.Action.RejectJob) _afterRejectJob(jobId);
    }
}
```

The hook enforces accountability at each stage. On `AcceptJob`, it verifies the solver is registered in the IRSB registry and locks a proportional bond. The bond size is computed from the job value — larger jobs require larger bonds, creating an economic incentive for accuracy that scales with stakes.

On `SubmitResult`, the hook auto-posts a V1 receipt through the trusted hook path. This is distinct from a solver manually posting a receipt: the hook path has elevated trust because the contract itself is attesting to the submission, not an external call that could be spoofed.

On `CompleteJob`, the bond is unlocked. On `RejectJob`, the dispute path opens automatically. The Watchtower monitors chain events from the hook and uses them as inputs to the behavioural-signal derivation pipeline.

All of this is deployed on Sepolia testnet. The contracts are not on mainnet. The hook path is under active development and the Watchtower's IRSB-client integration is the next milestone before any mainnet consideration.

Resilience Patterns A monitoring system that goes down when the thing it monitors is under stress is not useful. The resilience package addresses three failure modes, and each one maps onto a known distributed-systems pattern.

The **circuit breaker** wraps all chain RPC calls. If the chain node returns errors above a threshold rate, the circuit opens and the Watchtower stops attempting calls for a configured backoff window. This prevents the scan loop from hammering a degraded RPC endpoint and producing a backlog of failed requests that would overwhelm the system when connectivity recovers. When the circuit is open, the worker emits a metric and continues its loop — it does not crash. The pattern parallels how Khabbazian and colleagues' Outpost design handles unreliable channel observations: degrade gracefully, do not propagate the upstream failure[82].

Retry logic with exponential backoff handles transient failures: network blips, rate limits, momentary RPC unavailability. The retry policy is configurable per operation type. Evidence-store writes use aggressive retry because data loss is worse than latency. Webhook delivery uses moderate retry because the operator endpoint may legitimately be down during planned maintenance.

`DRY_RUN` **mode is mandatory for new deployments.** When `DRY_RUN=true`, the ActionExecutor simulates every action — dispute filings, bond operations, webhook deliveries — without submitting transactions or making external calls. All findings are logged. The action ledger records what would have happened. Operators can run the Watchtower against a real chain context for as long as they need to validate that the rules are producing sensible findings before enabling live execution.

[82] Khabbazian, M., Nadahalli, T., & Wattenhofer, R. (2019). Outpost: A Responsive Lightweight Watchtower. *AFT.* https://doi.org/10.1145/3318041.3355464

This is not optional. `DRY_RUN=false` requires an explicit configuration change. The design treats production execution as an opt-in after validation, not the default — the *fail-safe defaults* posture Saltzer and Schroeder named in 1975[83], extended from access control to action execution.

The Data Flow A complete Watchtower scan cycle, from trigger to persisted output:

1. **Worker** initiates a scan on the configured block interval.
2. **IrsbClient** calls `getBlockNumber()` to establish the current chain position (mock data in v0.5.0).
3. **createChainContext()** assembles the block number, recent receipts, and agent-registry snapshot into a context object.
4. **RuleEngine.execute(context)** runs every registered rule against the context. Each rule returns zero or more `Finding` objects.
5. **ActionExecutor** receives the findings. In `DRY_RUN` mode, it logs planned actions. In live mode, it submits transactions and calls external APIs.
6. **ActionLedger** records each action with its report ID. Idempotency check here: if the same report ID has already been acted on, the action is skipped.
7. **EvidenceStore** appends findings to JSONL. This is the permanent audit record.
8. **WebhookSink** fires for CRITICAL signals and HIGH_RISK_SCORE alerts. Payload is HMAC-signed with the operator's configured secret.
9. **Metrics** records scan duration, finding counts by severity, and action outcomes to Prometheus.

The JSONL persistence format is intentional. It is readable without tooling, appendable without locking, and trivially importable into any data pipeline. If a dispute is ever challenged on-chain and the operator needs to reconstruct the evidence trail, the answer is `cat evidence-store.jsonl | grep <receiptId>` — the same operator-grade auditability Schneier and Kelsey argued for in the forensic-audit-log setting[84].

What Comes Next The IRSB Ecosystem Deep Dive is a four-part series:

- Part 1 — 37 Solidity contracts, bond mechanics, and the receipt format
- Part 2 — submission, normalisation, and the evidence store
- **Part 3 (this post)** — Watchtower architecture, evidence verification, and risk scoring
- Part 4 (coming soon) — Z3 formal verification, the three-layer stack, and the pivot story

The Watchtower at v0.5.0 is code-complete for the monitoring logic with approximately 500 tests across the package suite. The next milestone is wiring the real IRSB client so the chain context uses live Sepolia data instead of mocks, followed by

[83] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[84] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM TISSEC*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

promoting the Cloud KMS signer from scaffolding to integration-tested status. After that, the path to a testnet production deployment is short.

The broader thesis behind this work is that AI-agent accountability cannot rely on trust-and-verify with humans in the loop. The challenge windows are too short, the agent volume is too high, and the failure modes are too varied. Automated monitoring with deterministic, auditable logic and mandatory dry-run validation before live execution is the only architecture that scales. The payment-channel watchtower literature reached the same conclusion a decade ago[85], [86], [87]; the Watchtower is what that argument looks like in practice for AI-agent receipts.

Part of the IRSB Ecosystem deep dive series. Built with Claude Code.

---

[85] McCorry, P., Bakshi, S., Bentov, I., Miller, A., & Meiklejohn, S. (2019). Pisa: Arbitration Outsourcing for State Channels. *AFT.* https://doi.org/10.1145/3318041.3355461

[86] Avarikioti, G., Litos, O. S. T., & Wattenhofer, R. (2020). Cerberus Channels: Incentivizing Watchtowers for Bitcoin. *Financial Cryptography and Data Security (FC).* https://doi.org/10.1007/978-3-030-51280-4_19

[87] Khabbazian, M., Nadahalli, T., & Wattenhofer, R. (2019). Outpost: A Responsive Lightweight Watchtower. *AFT.* https://doi.org/10.1145/3318041.3355464

Chapter 5. Deep Dive Part 4: Z3 Formal Verification, the Three-Layer Stack, and Claude Code as Architect

Every AI-governance product running today uses some version of LLM-as-a-judge: feed the agent's output to a second model and ask it whether the first model behaved. It is probabilistic reasoning about probabilistic reasoning. The result is a confidence score, not a proof. The IRSB ecosystem took a different path — formal verification using Z3, the SMT solver introduced by De Moura and Bjørner at TACAS 2008[88]. Z3 can mathematically prove the *absence* of a constraint violation rather than estimate its likelihood. That distinction matters when an autonomous agent has wallet access and real-world consequences attached to its decisions.

The smart-contract security literature has converged on the same posture. Luu and colleagues' Oyente[89] established symbolic execution as the canonical analysis primitive for Solidity; Kalra and colleagues' ZEUS[90] extended the formal-verification surface to safety properties; Atzei, Bartoletti, and Cimoli's systematisation[91] catalogued the failure modes those tools are designed to catch. IRSB's Z3 verifier is the same family of tool — applied to AI-agent tool calls rather than EVM bytecode.

This post is about how that system works, the three-layer stack it sits inside, and the story of building it with Claude Code as architect. Visit the IRSB Ecosystem hub for the full series context.

This is Part 4 of the IRSB Ecosystem Deep Dive series. Part 1 covered on-chain enforcement, Part 2 the evidence pipeline, Part 3 the Watchtower architecture.

Z3 Formal Verification: Replacing LLM-as-a-Judge The `FormalAgentVerifier` is the centrepiece of Moat's trust plane. It receives a `ToolCall` — the agent's intent to do something — and returns a `VerificationReport` with one of three outcomes: `PROVEN_SAFE`, `PROVEN_UNSAFE`, or `UNKNOWN`.

Nine constraints span six categories: `FILE_ACCESS`, `NETWORK`, `COMMAND_EXEC`, `DATA_EXFIL`, `RESOURCE_LIMIT`, and `PERMISSION`. Each constraint is a logical proposition that Z3 either satisfies or refutes. The system is fail-closed: `UNKNOWN` is treated as unsafe. If the solver times out, the call is blocked. The closed-world posture matches Saltzer and Schroeder's *fail-safe defaults* principle[92] applied to a formal-verification interface — *if you cannot prove it safe, treat it as unsafe*.

```
class FormalAgentVerifier:
    def verify(self, tool_call: ToolCall) -> VerificationReport:
```

[88] de Moura, L., & Bjørner, N. (2008). Z3: An Efficient SMT Solver. *TACAS*. https://doi.org/10.1007/978-3-540-78800-3_24

[89] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making Smart Contracts Smarter. *CCS*. https://doi.org/10.1145/2976749.2978309

[90] Kalra, S., Goel, S., Dhawan, M., & Sharma, S. (2018). ZEUS: Analyzing Safety of Smart Contracts. *NDSS*.

[91] Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A Survey of Attacks on Ethereum Smart Contracts (SoK). *POST*. https://doi.org/10.1007/978-3-662-54455-6_8

[92] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

```
        if not self.config.enabled:
            return VerificationReport(tool_call=tool_call, result=VerificationResult.PROVEN_SAI

        applicable = self._get_applicable_constraints(tool_call.tool_name)
        violations, proofs, checked = [], [], []

        for constraint in applicable:
            checked.append(constraint.name)
            try:
                is_safe, detail = self._verify_single_constraint(constraint, tool_call)
                if is_safe:
                    proofs.append(f"{constraint.name}: {detail}")
                else:
                    violations.append(f"{constraint.name}: {detail}")
            except Exception as exc:
                violations.append(f"{constraint.name}: verification error - {exc}")

        if violations:
            return VerificationReport(result=VerificationResult.PROVEN_UNSAFE, ...)
        return VerificationReport(result=VerificationResult.PROVEN_SAFE, ...)
```

The loop is deliberately simple. Complexity lives in the per-constraint verifiers, which use Z3's theory of strings, integers, and bit-vectors to construct proofs rather than heuristics — the same theory-mix Luu and colleagues exploited in Oyente for symbolic execution of EVM traces[93].

Path Traversal via Z3 String Theory The path-traversal check is the clearest illustration of what formal verification adds. The question is not *does this path look suspicious?* — it is *can Z3 prove that the strings `..` and `//` do not exist anywhere in this path value?*

```
def _check_path_traversal_z3(self, params: dict[str, Any]) -> tuple[bool, str]:
    z3 = _get_z3()
    path = str(params.get("path", params.get("file_path", "")))

    solver = z3.Solver()
    solver.set("timeout", self.config.solver_timeout_ms)

    path_var = z3.String("path")
    has_traversal = z3.Or(
        z3.Contains(path_var, z3.StringVal("..")),
        z3.Contains(path_var, z3.StringVal("//")),
    )
```

[93] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making Smart Contracts Smarter. *CCS*. https://doi.org/10.1145/2976749.2978309

```
    solver.add(path_var == z3.StringVal(path))
    solver.add(has_traversal)

    result = solver.check()
    if result == z3.unsat:
        return True, f"Z3 proved no traversal sequences in '{path}'"
    elif result == z3.sat:
        return False, f"Z3 found traversal sequence in '{path}'"
    else:
        return False, f"Z3 solver returned {result} (fail-closed)"
```

The solver adds two assertions: the path variable equals the actual path, and the path contains a traversal sequence. If Z3 returns `unsat` — meaning no assignment satisfies both constraints simultaneously — the absence of traversal is proven. A regex could be bypassed by encoding tricks. A Z3 proof cannot. The same distinction is what gives the smart-contract verification literature its bite: Kalra and colleagues' ZEUS argues that policy-violation detection based on dynamic analysis or pattern matching has unbounded false-negative rates, whereas SMT-backed verification has bounded ones[94].

The same approach applies to network constraints (port-range verification via integer arithmetic), command-execution checks (allowlist membership via set theory), and data-exfiltration detection (payload size bounds via bit-vector arithmetic). The choice of theory depends on the domain of the constraint.

The Three-Layer Stack The full vision is three independently useful layers that together form a complete accountability stack for AI-agent transactions.

Layer 1 — IRSB Protocol. Cryptographic accountability at the protocol level. Receipts, bonds, disputes, delegation chains, and enforcers. 552 protocol tests. Deployed on Sepolia testnet. Mainnet deployment is planned for Q3 2026. The protocol contracts provide economic security: agents post bonds, violators get slashed, bad actors enter jail. The shape is Buterin and Griffith's Casper-style staking[95] adapted to AI-agent accountability rather than chain finality. Nothing in Layer 2 or 3 replaces this — they build on top of it.

Layer 2 — Moat. Policy-enforced execution. An agent cannot make a network call, write a file, or spend funds without passing through Moat's gateway. Scope policies (which tools an agent may use), budget policies (how much it may spend), and domain policies (which hosts it may contact) are evaluated before execution. The Z3 verifier sits in the trust plane alongside the Cedar policy engine (planned, not yet created), providing formal proof of constraint satisfaction for each tool call.

Layer 3 — Scout. Intelligent work brokering. Scout discovers bounties, matches them to capable agents, and routes execution through Moat. It does not execute work

[94] Kalra, S., Goel, S., Dhawan, M., & Sharma, S. (2018). ZEUS: Analyzing Safety of Smart Contracts. *NDSS.*

[95] Buterin, V., & Griffith, V. (2017). *Casper the Friendly Finality Gadget.* arXiv:1710.09437.

— it finds the right worker and ensures the work flows through the accountability stack below.

Each layer can be adopted without the others. A team that wants policy-enforced execution without on-chain receipts can run Moat standalone. A team that wants formal verification without brokering can use the verifier directly. The stack design is composable by intent. The independent-composability discipline echoes how Buchman and colleagues argue Tendermint should be adopted: each layer is a stable abstraction that the next layer up can reach for without inheriting the layers below[96].

Scout: The Brokering Brain Scout lives inside the Moat repository (`github.com/jeremylongshore/`) as a separate service. Its interface is eight MCP tools, split evenly between capability management and bounty workflow.

```
TOOL_SCHEMAS = [
    # Core capability tools
    {"name": "capabilities.list",    "description": "List registered capabilities with filters"
    {"name": "capabilities.search",  "description": "Search capabilities by name/description/ta
    {"name": "capabilities.execute", "description": "Execute through policy-enforced gateway"}
    {"name": "capabilities.stats",   "description": "7-day reliability stats and trust signals"
    # Scout-workflow tools
    {"name": "bounty.discover",      "description": "Search Algora/Gitcoin/Polar/GitHub for bou
    {"name": "bounty.triage",        "description": "Assess bounty feasibility and match to age
    {"name": "bounty.execute",       "description": "Route matched work through Moat gateway"}
    {"name": "bounty.status",        "description": "Track bounty execution and receipt collect
]
```

The `bounty.discover` tool aggregates across four platforms: Algora, Gitcoin, Polar, and GitHub Sponsors. `bounty.triage` scores feasibility against registered agent capabilities and returns a match score. `bounty.execute` does not run the work — it packages the matched bounty and agent, routes it through Moat's gateway with appropriate scope and budget policies attached, and returns a gateway receipt.

The Money Agent, which handles payment flows after successful execution, is currently a stub. The trust-signal system in `capabilities.stats` tracks 7-day reliability metrics per registered capability, feeding into the triage scoring and eventually into IRSB bond sizing.

The Competitive Moat The governance landscape for AI agents splits into two camps: Web2 MCP-governance tools (Runlayer, Natoma, Acuvity, Lasso) and Web3 intent protocols (CoW Protocol, Across/UMA, AgentKit). Neither has the full picture.

[96] Buchman, E., Kwon, J., & Milosevic, Z. (2018). *The Latest Gossip on BFT Consensus.* arXiv:1807.04938.

Dimension	Web2 MCP Governance	Web3 Intent Protocols	IRSB + Moat + Scout
Policy enforcement	Dashboard rules	Smart-contract limits	Both + Z3 formal proofs
Audit trail	Database logs	On-chain events	Both + evidence bundles
Economic security	None	Bonds (some)	Bonds + slashing + jail
Agent monitoring	Dashboard alerts	None	Watchtower (automated)
Dispute resolution	Manual review	None	Automated + escalation
Bounty discovery	None	None	4 platforms via Scout

Web2 tools have flexible policy surfaces but no economic stakes and no formal verification. Web3 protocols have economic stakes but no behavioural governance for agents operating above the transaction layer. IRSB closes both gaps simultaneously.

The framework-integration roadmap targets six ecosystems across three priority tiers.

Tier	Framework	Integration Point	Priority
1	ElizaOS	Plugin for receipt + bond	High
1	Safe + Modules	Module for IRSB enforcers	High
2	AgentKit	CDP integration	Medium
2	Olas	Service component	Medium
3	Brian AI	Transaction receipts	Lower
3	Virtuals	Agent monitoring	Lower

ElizaOS and Safe are the highest-leverage entry points. ElizaOS has the largest active community building autonomous agents. Safe has the on-chain infrastructure — modules, guards, and multisig governance — that maps cleanly to IRSB's enforcer model.

The Pivot Story The February 2026 pivot was not gradual. Between February 5 and 10, the project shifted from a generic intent-protocol design — competing with Across and UMA on bridging and settlement — to a focused AI-agent accountability layer.

The pivot insight was specific: AI agents with wallet access are being deployed without adequate accountability infrastructure. The Across/UMA space already has credible solutions for cross-chain intent settlement. The AI-agent space does not have a

credible solution for what happens when an agent misbehaves, overspends, or makes a decision a human stakeholder later disputes.

IRSB's receipt/bond/dispute infrastructure, already designed for on-chain settlement, mapped directly onto this problem. A receipt proves an agent took a specific action. A bond creates economic stakes. A dispute mechanism creates recourse. The protocol did not need to be redesigned — it needed to be aimed at a different target.

The other significant architectural change was the Lit Protocol migration. The original design used Lit Protocol for distributed key management — threshold signatures across Lit nodes for every signing operation. In practice this meant added latency, a hard dependency on Lit node availability, and meaningful operational complexity for every production deployment. Cloud KMS replaced it. Google manages the HSMs, the signing operations are synchronous, and the dependency graph is simpler. The architectural elegance of threshold cryptography was real, but the operational cost was not worth it for the current stage of the project. Brooks' *No Silver Bullet* argument about *essence vs accidents* applies directly[97]: the threshold-cryptography sophistication was accidental complexity at this stage; the essential question was *can a hardware-backed key produce a verifiable signature with audit trail?*, and Cloud KMS answers that more simply.

Claude Code as Architect Two hundred and eighty-five commits. Fifty-two days. Six EIPs implemented. Three repositories built to production standards on Sepolia testnet.

The collaboration model for IRSB followed the same pattern documented in the Wild ecosystem deep dive series: Jeremy Longshore as product owner and human architect, Claude Code as technical lead. The division of responsibilities:

Jeremy defined: - Protocol semantics and economic design - Integration targets and positioning - The architectural pivot from generic intents to AI-agent accountability - Non-negotiable safety constraints

Claude Code decided: - Data-model specifics and type hierarchies - Test architecture and adversarial test cases - Implementation patterns within each service - Cross-repository consistency and interface contracts

The CLAUDE.md pattern is the binding contract between these two roles. Every service has a `CLAUDE.md` that defines what the service does, what it does *not* do, its safety rules, and the scope boundaries the AI implementer must not cross. These files are not documentation — they are operational constraints for the AI. An instruction in `CLAUDE.md` that says "never execute tool calls without a valid verification report" is enforced across every session. The design responds to the *lost in the middle* effect Liu and colleagues catalogued for long-context language models[98]: a constraint placed at

[97] Brooks, F. P. (1987). No Silver Bullet — Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4), 10–19. https://doi.org/10.1109/MC.1987.1663532

[98] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). Lost in the Middle: How Language Models Use Long Contexts. *TACL*, 12, 157–173. https://doi.org/10.1162/tacl_a_00638

the front of the session-context window is more reliably attended to than one buried in the middle.

The adversarial test suite for the formal verifier is a good example of Claude Code operating as genuine technical lead. The test cases were not specified by Jeremy — they were generated by the AI reasoning about attack surfaces: encoding tricks that might bypass path checks, malformed parameter types that could trigger edge cases, timing sequences that could exploit the fail-open path before the verifier initialises. The methodology echoes Perez and colleagues' *language-models-against-language-models* red-teaming pattern[99]: let the system under test help design the attacks. A copilot fills in the implementation. A tech lead thinks adversarially about what it just built. Park and colleagues' generative-agents work[100] makes a related observation in a different setting: agents asked to plan ahead about consequences produce qualitatively different artefacts than agents asked only to react.

Regulatory Tailwind and Licensing The EU AI Act compliance deadline for high-risk AI systems is August 2, 2026[101]. Article 12 requires providers of high-risk AI systems to implement logging mechanisms sufficient to ensure traceability of outputs. IRSB receipts — immutable, on-chain, tied to cryptographic agent identifiers — satisfy this requirement natively. Organisations deploying AI agents in regulated verticals (financial services, healthcare, infrastructure) are looking for accountability infrastructure that does not require custom development. IRSB provides it at the protocol layer.

The licensing structure reflects a deliberate commercial strategy. IRSB is licensed under BUSL-1.1, which restricts commercial use by competing protocols until the change date. That change date is February 17, 2029, when the license converts to MIT. The protocol is commercially protective now, fully open-source later. This is the same pattern used by Uniswap v3 and several other DeFi protocols that needed first-mover protection during the growth phase.

The Full Picture Four posts, three repositories, one protocol.

- Part 1: On-Chain Enforcement — 37 smart contracts, 552 tests, receipt/bond/dispute semantics on Sepolia
- Part 2: The Evidence Pipeline — evidence bundles, IPFS anchoring, dispute-submission flow
- Part 3: The Watchtower Architecture — automated monitoring, anomaly detection, escalation chains

[99] Perez, E., Huang, S., Song, F., Cai, T., Ring, R., Aslanides, J., Glaese, A., McAleese, N., & Irving, G. (2022). Red Teaming Language Models with Language Models. *EMNLP*. https://doi.org/10.18653/v1/2022.emnlp-main.225

[100] Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative Agents: Interactive Simulacra of Human Behavior. *UIST*. https://doi.org/10.1145/3586183.3606763

[101] European Parliament and Council of the European Union (2024). *Regulation (EU) 2024/1689 (EU AI Act)*. https://eur-lex.europa.eu/eli/reg/2024/1689/oj

- Part 4: This post — Z3 formal verification, the three-layer stack, Claude Code as architect

The gap between *agents can transact* and *agents can transact safely* is the gap IRSB fills — not with promises, but with mathematics, economics, and 37 smart contracts.

Part of the IRSB Ecosystem deep dive series. Built with Claude Code.

Chapter 6. Guidewire MCP v0.1.0: Carrier-Native Server Blueprint

Guidewire MCP for Claude v0.1.0 is a carrier-native Model Context Protocol[102] (MCP) server that lets a Claude Code session ask a Guidewire PolicyCenter, BillingCenter, or ClaimCenter tenant questions in the language an underwriter or claims handler would actually use — `find-submissions-waiting-on-me`, `summarize-this-submission`, `did-we-lose-this-account` — instead of the API verbs an integration engineer would reach for. The v0.1.0 cut shipped on 2026-05-04 as 30 merged PRs (+14,521 / -819 lines), comprising six foundation packages, five read-only PolicyCenter tools, a Claude Code plugin manifest, and a live architecture diagram on a custom subdomain. Alongside the code, ~30k words of blueprint documents — business case, PRD, architecture, user journey, technical spec, roadmap — landed in `blueprint/` on the same day. (All 30 merged PRs are visible at github.com/jeremylongshore/guidewire-mcp-for-claude/pulls?q=is%3Apr+is%3Amerged.)

The thesis: shipping a v0.1.0 carrier-native MCP server in one day is only possible when the blueprint, the foundation packages, and the safety harness are designed as one system from the start — not bolted on after. Each piece in the v0.1.0 cut depends on architectural decisions that were locked into the blueprint *before* any package code was written. That sequencing — design as an artefact, not a retrofit — is what made the parallel construction of six independently-versioned packages tractable on a one-day clock. Parnas's 1971 argument about *information distribution aspects of design methodology*[103] is the original statement of why: the module boundaries must encode the design decisions most likely to change, which means the decisions themselves have to be named in writing before the modules can be drawn.

This post walks through what is actually in v0.1.0, how the blueprint set functions as load-bearing infrastructure rather than after-the-fact documentation, why the harness pattern (`plan → policy → approval → execute → audit → rollback`) is the spine of the entire writes story, and where the limits of the cut sit honestly. There is also a parallel sister-pack rebuild that landed the same day in a separate repo, which is worth covering because the two artefacts compose in ways that matter for anyone trying to ship enterprise integrations on a Claude Code substrate.

What v0.1.0 actually contains The v0.1.0 cut is best read as four concentric layers that correspond to the four epics opened in the roadmap: foundation (E1), read-only tools (E2), harness scaffold (E3), and on-disk profile loader (E4). The package boundaries are deliberate — each one stands alone, has its own README, owns its own tests, and is consumed only through its public exports.

[102] Anthropic (2024). *Model Context Protocol Specification.* https://modelcontextprotocol.io/

[103] Parnas, D. L. (1971). Information Distribution Aspects of Design Methodology. *Proceedings of the IFIP Congress.* https://doi.org/10.1184/R1/6606470.V1

Layer	Artifact	What it does
E1	`@gw/schemas`	Zod schemas for every Guidewire payload the runtime touches. Type errors become validation errors at the boundary, not crashes deep in tool code.
E1	`@gw/observability`	OpenTelemetry tracing + pino structured logs + Sentry error sink. Wired in from day 1, defaulted off until the operator points them somewhere.
E1	`@gw/auth`	Guidewire Hub OAuth client + JWT propagation. Actor identity flows through every tool call — there is no shared service-account key with read-all scopes.
E1	`@gw/audit`	Postgres-backed audit log with hash-chain. Each row's hash includes the previous row's hash, so tampering is detectable.
E1	`@gw/client-sdk`	Cloud API client built on undici. Two-key idempotency: client-side request key plus Cloud API native idempotency header.
E1	`@gw/mcp-runtime`	MCP protocol implementation with both stdio and HTTP transports. Tools register themselves through a typed factory; runtime handles dispatch, validation, and observability.
E2	5 PolicyCenter tools	`find-submissions-waiting-on-me`, `show-policies-for-this-insured`, `summarize-this-submission`, `did-we-lose-this-account`, `pull-this-submission`. All read-only. All speak carrier vocabulary, not REST verbs.

Layer	Artifact	What it does
E3	Harness library + CLI	Plan / policy / approval / execute / audit / rollback pipeline. Skeleton in #80 — wiring real writes to this is E3+ work.
E4	`--profile <path>` loader	On-disk profile scaffold (#75). Loads tenant connection details, role mappings, and per-tool overrides from a YAML file the operator maintains.

The hash-chain construction in `@gw/audit` is a direct descendant of Schneier and Kelsey's 1999 design for forensic audit logs[104] and Haber and Stornetta's 1991 time-stamping scheme[105]: each row's integrity depends on the previous row's hash, so tampering with any past row breaks every subsequent row's hash. The integrity property does not require trusting the storage tier; it requires only that the operator can read the chain end-to-end.

Fifty-four tests pass across the foundation packages. The plugin manifest landed in #76, which means installation is a single command in any Claude Code session:

```
/plugin install jeremylongshore/guidewire-mcp-for-claude
```

Once installed, the runtime expects four environment variables to talk to a tenant:

```
export GUIDEWIRE_OAUTH_CLIENT_ID="your-client-id"
export GUIDEWIRE_OAUTH_CLIENT_SECRET="your-client-secret"
export GUIDEWIRE_TOKEN_ENDPOINT="https://<your-hub>.guidewire.net/oauth2/v1/token"
export GUIDEWIRE_PC_BASE_URL="https://<your-tenant>.pc.guidewire.net/pc/api"
```

A live architecture diagram of the whole stack is published at guidewire-mcp.intentsolutions.io (PR #53). The page is itself a credibility artefact — the architecture is not just documented in markdown inside the repo, it is rendered as a public artefact a prospect can read without cloning anything.

Carrier vocabulary as the dominant abstraction The five tools in v0.1.0 are deliberately named the way an underwriter or claims handler would describe them in a hallway conversation, not the way a Cloud API endpoint is named. That is D-001 in the architecture document, and it is the most consequential design decision in the whole cut. Every other surface — the schema names in `@gw/schemas`, the registration API in `@gw/mcp-runtime`, the user-journey scenarios in `04-USER-JOURNEY.md` — derives from it.

[104] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM TISSEC*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

[105] Haber, S., & Stornetta, W. S. (1991). How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2), 99–111. https://doi.org/10.1007/BF00196791

The contrast matters. A naive MCP server over Guidewire would expose tools that map 1:1 to Cloud API operations: `get_submission`, `list_policies_by_account`, `get_account_by_producer_code`. That surface is technically correct and operationally useless. An underwriter does not think *I need to call `list_policies_by_account` with the producer code I derived from the agency lookup.* An underwriter thinks *show me the policies for this insured.* The vocabulary gap is the integration's value proposition.

The agent-tool-use literature reinforces the same point. Schick and colleagues' Toolformer[106] and Yao and colleagues' ReAct[107] both implicitly assume that the *names and shapes* of tools shape what the agent can reason about. A tool named `get_submission` invites mechanical retrieval; a tool named `find-submissions-waiting-on-me` invites the operator's mental model of the work. The tool name is part of the prompt, and the carrier-vocabulary discipline is exactly the part of the prompt the operator has the strongest opinion about.

Concretely, `find-submissions-waiting-on-me` is not a single Cloud API call. It is a composed query: fetch the operator's role, resolve the queues they are assigned to, fetch open submissions in those queues filtered by status `Pending Underwriter Review` or `Returned for Information`, sort by SLA-breach risk, and return a structured summary. The composition logic lives inside the tool. The operator never sees the underlying Cloud API plumbing.

The Zod schema for the tool's response reflects the same vocabulary discipline:

```
export const SubmissionWaitingOnMe = z.object({
  submissionNumber: z.string(),
  insuredName: z.string(),
  effectiveDate: z.string(),
  premiumEstimate: z.number().nullable(),
  daysWaitingForMe: z.number(),
  slaBreachIn: z.string().nullable(),
  whyItIsWaiting: z.enum([
    'pending-underwriter-review',
    'returned-for-information',
    'awaiting-broker-response',
  ]),
});
```

`whyItIsWaiting` is a denormalised, vocabulary-correct enum. The Cloud API exposes this as a status code with carrier-specific extensions — the schema flattens that into the three states an operator actually distinguishes between. Two months from now when the underwriter is asking *why is this one still on my plate*, the answer comes back in their language.

This is also what makes the harness pattern feasible later. A write tool named

[106] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. *NeurIPS*. arXiv:2302.04761.

[107] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR*. arXiv:2210.03629.

`bind-this-quote-with-these-changes` can produce a plan that is reviewable by a human approver *in the operator's terms*. A write tool named `submit_quote_binding_request_with_overrides` produces a plan that requires translation. The vocabulary discipline at the read layer is what unlocks the approval UX at the write layer.

The blueprint set is load-bearing infrastructure, not documentation The instinct on a one-day ship is to skip docs and write code. That instinct is wrong here, and the structure of the day's commits shows why. The blueprint set landed *first*, as its own PR sequence, before the bulk of E1 package code merged:

Doc	Words	PR
01-BUSINESS-CASE.md	3.0k	#39
02-PRD.md	8.2k	#35
03-ARCHITECTURE.md	5.3k	#38
04-USER-JOURNEY.md	5.4k	#42
05-TECHNICAL-SPEC.md	7.4k	#41
07-ROADMAP.md + E2.5 sub-epic	—	#30, #43
Phase 0 specialist memos	16,963	#27

The blueprint pre-commits the architecture decisions — labelled D-001 through D-022 in `004-DR-DEC-architecture-decisions.md` and referenced by anchor link from `03-ARCHITECTURE.md` — that the package code then implements. Without that pre-commitment, six packages cannot be built in parallel, because each package's public surface depends on assumptions the other five also have to honour. The blueprint *is* the contract between the packages. Parnas's 1971 paper makes the same point in different vocabulary: the design decisions that are about to be encoded in module boundaries must be named in writing first, otherwise the modules will encode contradictory assumptions[108].

A concrete example: D-005 (*every write passes through plan → policy → approval → execute → audit → rollback hint*) and D-006 (*HARD RULE: no audit, no write*) together fix the shape of `@gw/audit`'s public API *and* the contract between `@gw/mcp-runtime` and the harness *and* the shape of every future write tool's signature. If those decisions had been deferred to *we'll figure it out when we get to writes*, `@gw/audit` would have shipped with an API the harness later had to adapt around. By writing the architecture document first and committing to D-005/D-006 explicitly, every downstream package was built against a stable contract.

The same pattern holds for D-001 (carrier vocabulary as the dominant abstraction) — that decision shapes the tool-naming convention in `@gw/mcp-runtime`'s registration API, the schema names in `@gw/schemas`, and the user-journey scenarios in `04-USER-JOURNEY.md`. One decision, four files, all consistent because the decision was a written artefact before any of them got built.

[108] Parnas, D. L. (1971). Information Distribution Aspects of Design Methodology. *Proceedings of the IFIP Congress.* https://doi.org/10.1184/R1/6606470.V1

The blueprint also operates as the load-bearing anchor for long-running Claude Code sessions. Liu and colleagues' work on *lost in the middle* attention degradation in long-context models[109] is the warning: a decision buried in the middle of a long prompt is less reliably attended to than a decision at the front. Putting the architecture decisions in a numbered, anchor-linkable document means every session can be pointed at `D-005` directly, and the relevant context is always at the top of whatever subset of the doc is loaded into a given session's working memory.

The Phase 0 specialist memos (16,963 words across four memos) covered the harder questions that would have stalled package construction if discovered mid-build: which Guidewire APIs are actually stable across InsuranceSuite versions, what the real OAuth flow looks like with Hub, what the carrier-vocabulary mapping should be for the operator queries the PRD enumerates, and what the harness runtime needs to look like to support the eventual write surface. Those memos became the librarian KB entries `005-DR-REF-guidewire-public-resources` through `009-DR-MEMO-harness-runtime`, which means the same content is reachable both from the blueprint's narrative reading order and from the librarian's topic-keyed lookup. Two access paths, one source of truth.

The five read-only tools, in operator language The tools shipped in v0.1.0 cover the high-frequency PolicyCenter operator queries. Each one takes natural-language-ish parameters that match how an underwriter would describe the question, not how the Cloud API expresses it. A short tour:

`find-submissions-waiting-on-me` — the morning-coffee query. Returns the operator's queue of submissions where the next required action is theirs. The tool resolves the operator's role and queue assignments from the JWT, fetches the queue contents, and ranks results by SLA-breach risk. No parameters needed beyond the implicit identity. Sample session interaction:

```
> use find-submissions-waiting-on-me

Found 7 submissions waiting on you:
  1. SUB-2398471 - Acme Manufacturing - 3 days waiting - SLA breach in 2 days
  2. SUB-2398502 - Bayside Logistics - 2 days waiting - returned for information
  ...
```

`show-policies-for-this-insured` — the account-context query. Takes an insured name or account number and returns the active policy book, including effective dates, premium, lines of business, and producer of record. Resolves the account first (with a fuzzy match that surfaces ambiguity rather than guessing), then pulls the policy list.

`summarize-this-submission` — the deep-dive query. Takes a submission number and returns a structured summary that an underwriter would want before opening the full submission UI: insured, broker, lines of business requested, current quote (if any), open underwriting questions, attachments, and the latest activity entry. The

[109] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). Lost in the Middle: How Language Models Use Long Contexts. *TACL*, 12, 157–173. https://doi.org/10.1162/tacl_a_00638

summary structure is pinned to the underwriter's mental model of *what do I need to decide about this submission?* not to the Cloud API's resource hierarchy.

`did-we-lose-this-account` — the loss-context query. Takes an account number or insured name and returns whether any of the account's submissions ended in `Not Taken` or `Declined` over the last N days, with the recorded reason. This is the kind of query that takes three or four Cloud API calls to assemble and that an underwriter asks several times a week — a perfect candidate for a single tool.

`pull-this-submission` — the deep-fetch query. The escape hatch when the structured summary is not enough. Returns the full submission payload as the Cloud API renders it, with no flattening or vocabulary mapping. Useful for the rare case where the operator needs to see something the structured tools do not surface.

The five tools cover roughly 70% of the read-side queries enumerated in the PRD's user-journey section. The remaining 30% — the long-tail queries that BillingCenter and ClaimCenter operators run, plus the more specialised PolicyCenter queries around portfolio analysis and producer performance — are the E2.5 and E3 work. Five tools is enough to make the runtime useful for a real PolicyCenter operator from day one; it is not enough to be a complete operator workbench, and the README says so.

The harness pattern: plan → policy → approval → execute → audit → rollback
Read-only tools are the easy part. The interesting design question is what happens when v0.2.0 starts adding write tools that can issue a quote, bind a policy, or post a payment. The harness pattern — scaffolded in #80 — is the answer.

Every write call passes through six stages:

Stage	What it does	Failure mode
plan	Tool produces a structured proposal of the intended change. No mutation yet.	Plan rejected → no write, no audit row needed.
policy	Policy engine evaluates the plan against tenant rules (role, dollar threshold, business hours, blackout dates).	Policy denial → no write, audit row records the denial.
approval	If policy requires human approval, the request is queued and the tool returns. Approval can come from another Claude Code session, a UI, or a Slack/email channel.	Timeout or denial → audit row records outcome, no write.

Stage	What it does	Failure mode
execute	Cloud API call with two-key idempotency. The plan's structure constrains what the executor is allowed to do — no off-plan side effects.	Cloud API error → audit row records full request/response, executor returns structured error.
audit	Append a hash-chained row to Postgres covering the actor, plan, policy decision, approval (if any), and the execute result.	Audit insert failure → **the entire write is rolled back**. No audit row, no completed write.
rollback hint	Tool returns a structured rollback descriptor the operator can later use to undo the change if something downstream goes wrong.	Tool may decline to provide a rollback (e.g., for irreversibly-bound policies) — the hint is then "manual reversal required" with full context.

The hard rule from D-006 is the load-bearing one: **no audit, no write.** If the audit insert fails, the executor's state-change must not survive. In practice that means the executor wraps the Cloud API call and the audit insert in a coordination boundary — for non-transactional Cloud API endpoints, the rollback step has to fire to undo the executed change before reporting failure. The alternative — letting writes happen with no audit row — would mean a tenant could accumulate state changes that have no provenance, which destroys the entire trust model. This is the *integrity grounded in an external trust anchor* posture Schneier and Kelsey identified for audit-log design[110]: the row that records *what happened* is the source of truth; if there is no row, nothing happened that is allowed to persist.

Why is this pattern in v0.1.0 even though there are no write tools yet? Because the harness library has to exist, with its public API stable, before any write tool can be built. The scaffold landed in #80 as a deliberate pre-commitment. When the first write tool gets written, the harness contract is already there — the tool author cannot accidentally ship a tool that bypasses any stage, because the runtime's tool-registration API for write tools requires a plan/policy/audit hook set.

Karate adopted in #79 covers the contract layer for the Cloud API client. Karate scenarios pin the request/response shapes of the underlying Cloud API endpoints, so that a Guidewire-side schema drift surfaces as a contract-test failure rather than a runtime error in production. The contract layer is scaffolded but not yet exercised against a real tenant — that is E3+ work.

Observability wired in, defaulted off D-013 is one of the smaller-looking decisions in the architecture document and one of the most consequential in practice:

[110] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM TISSEC*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

observability is wired in from day 1, not bolted on. The `@gw/observability` package provides OpenTelemetry trace context, pino structured logs, and a Sentry error sink — and every tool call, every Cloud API request, every audit insert flows through those instruments by default.

The *defaulted off* half of the decision matters as much as the *wired in* half. The runtime ships with no OTel collector endpoint, no Sentry DSN, and pino's transport set to a local file. The instruments are alive but silent until the operator points them somewhere. A prospect cloning the repo and running it locally does not need to decide on an observability stack to make the runtime work — and the runtime is not phoning home through some default-on telemetry channel that would compromise the local-first trust story.

When the operator is ready, the standard OTel and Sentry environment variables turn the instruments on:

```
export OTEL_EXPORTER_OTLP_ENDPOINT="https://your-collector:4318"
export SENTRY_DSN="https://your-sentry-key@sentry.io/your-project"
```

Bolting observability on later is the path the design avoided. A package that has no trace context at the boundaries cannot retroactively gain it without rewriting every call site. A package that does not log structured fields cannot retroactively gain them without changing every log line. Wiring observability through `@gw/observability` from the first commit means every package built on top of it inherits the same boundary instrumentation, with no per-package work.

The audit and observability layers are deliberately separate. The audit log is the legally-relevant record of what happened, lives in the customer's Postgres, and has a hash chain. The observability stack is the operational record of how things are running, lives wherever the operator points OTel and Sentry, and has no integrity guarantees. Conflating the two — using Sentry as the audit trail, for example — would be a category error: Sentry is for the operator running the system, not for the compliance team auditing it. Schneier and Kelsey make the same distinction explicit[111]: the audit log is the *legally-relevant* record, and it must not be reusable as a general-purpose operational log.

Why local-first changes the OSS calculus The trust-model decision in `03-ARCHITECTURE.md` is short and consequential: the runtime is local-first and customer-hosted. There is no *Intent Solutions cloud* zone in the architecture. The OSS repo does not phone home. The audit database is in the customer's own Postgres. The tenant credentials live in the customer's own secret store. When a Claude Code session uses the MCP server, the only network traffic is between the session, the local MCP runtime, and the customer's own Guidewire tenant.

That decision changes what the OSS repo *does* in a sales motion. A vendor-hosted integration product would need a security-review cycle, a SOC 2 review, a legal review of the data-processing agreement, and a procurement cycle before any prospect could even try it. A local-first OSS repo collapses that to *clone the repo, point it at your*

[111] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM TISSEC*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

sandbox tenant, run a tool against your own data. The credibility artefact is not a slide deck — it is the artefact itself, on the prospect's own machine, talking to their own tenant, with a public architecture diagram explaining exactly what it is and is not doing.

D-009 and D-010 in the architecture doc make this explicit: the OSS repo is not a complete product. It is a credibility artefact and a lead magnet for custom build engagements. The full carrier-specific build — the 39-tool catalogue from the PRD, the harness wired to real writes, the on-disk profiles validated against a specific tenant's role model — is the engagement. The OSS repo is what convinces the prospect the engagement is worth scoping.

This is also why the JWT-propagated actor-identity decision matters. Every tool call carries the operator's JWT, and the audit row records *who* asked. There is no shared service-account key whose audit trail collapses every action into *the integration did it*. When a carrier's compliance team asks *who pulled this submission?*, the answer is the actual human, not *the MCP server*. That property is non-negotiable for any regulated insurance workload, and it had to be wired through `@gw/auth` from day 1 — adding it later would have required reworking every tool call's signature.

The two-key idempotency in `@gw/client-sdk` is a quieter but related decision. The client generates a deterministic request key from the tool call's plan (or, for read calls, from the canonicalised query parameters) and passes it both as a client-tracked dedup key and as the Cloud API's native idempotency header. If the same request retries — because the operator's session was interrupted, because the network blipped, because the harness's executor stage retried after a transient failure — the Cloud API recognises the duplicate and returns the original response rather than executing the call twice. That property matters enormously for any future write tool: a retry that succeeds twice could double-bind a policy or double-post a payment. Wiring two-key idempotency into the SDK at v0.1.0, when there are no writes, is what makes write tools safe to add at v0.2.0 without a per-tool retry-safety review.

The parallel sister-pack rebuild A second thread ran in parallel the same day in `claude-code-plugins-plus-skills`: twelve commits rebuilt the `guidewire` skill pack to A-grade across every skill in the marketplace, prepared for the v4.30.0 release.

Skill	PR	What changed
`install-auth`	#668	Production-grade Hub OAuth + tenant connection.
`sdk-patterns`	#669	Cloud API client patterns lifted to A-grade.
`local-dev-loop`	#670	Fast Gosu iteration loop for tenant-side rule changes.
`core-workflow-a`	#671	PolicyCenter end-to-end workflow.

Skill	PR	What changed
`core-workflow-b`	#672	ClaimCenter end-to-end workflow.
`security-and-rbac`	#673	Merge of `security-basics` + `enterprise-rbac` — single source for RBAC patterns.
`observability-and-incident-response`	#674	Merge of three observability skills into one.
`ci-cd-pipeline`	#675	Merge of four CI/CD skills into a single coherent pipeline.
`webhooks-integrations`	#676	Renamed and deepened — covers the actual Guidewire webhook event shapes.
`migration-and-upgrade`	#677	Merge of two migration skills.
Pack v2.0.0 cleanup	#678	Pack-level metadata, version bump, README sync.
`chore(release): prepare v4.30.0`	`3a2d27d63`	Marketplace release prep.

The two repos compose. The MCP server (`guidewire-mcp-for-claude`) is the runtime — the thing that talks to a tenant. The skill pack (`guidewire` in the marketplace) is the body of operator know-how — patterns for how to use the runtime, how to debug Guidewire integrations, how to set up CI/CD around a Cloud API project, how to think about RBAC on top of carrier identity. Installing both gives a Claude Code session the runtime *and* the operator playbook in one go. Neither one is sufficient on its own; together they are the carrier integrator's working environment.

The same-day timing matters. If the MCP server had shipped without the skill-pack rebuild, an operator installing the runtime would still have been operating against the previous pack's lower-grade material. Pulling both threads on the same clock means the v0.1.0 announcement carries both artefacts at once.

What didn't ship — the limits of v0.1.0 Honest scoping. v0.1.0 is not the full product. The cut is deliberately a foundation, and it stops at well-defined boundaries:

- **All five tools are read-only.** No writes in v0.1.0. The harness scaffold exists; no tool uses it yet.
- **The Karate contract layer is scaffolded, not exercised.** A real run against a sandbox tenant is E2.5 work.
- **The `--profile <path>` loader is scaffolded, not validated against a real tenant.** The YAML schema is defined; the integration test that loads a profile and uses it to talk to a live sandbox is E4 follow-up.

- **The PRD's full 39-tool catalogue is mostly future work.** Five tools is enough to make the runtime useful for the most common PolicyCenter operator queries; it is not enough to be a complete underwriting workbench.
- **BillingCenter and ClaimCenter tools are not in v0.1.0.** PolicyCenter only. The other centres have schema scaffolding in `@gw/schemas` but no live tools yet.
- **The audit hash-chain is not yet wired to an external anchor (e.g., periodic notarisation).** Tampering is detectable within the chain, but the chain's tip is not yet anchored anywhere external. That is a v0.3.0+ concern, and the design will likely follow the Haber–Stornetta external-anchor pattern[112] when it gets implemented.
- **No formal carrier reference deployment yet.** The runtime has been exercised against synthetic tenants and the public Guidewire developer sandbox shapes; a real carrier production deployment would require its own engagement.

These limits are stated in the README and in `07-ROADMAP.md`. A prospect reading the repo gets an accurate picture of where the line is between *shipped and usable* and *promised in the roadmap*.

Read-only first: the alternatives that lost A reasonable critique of v0.1.0 is *five read-only tools is not very impressive — write tools are the real value.* The critique is fair on its face and wrong on the substance. Read-only first was a deliberate choice over two alternatives, both of which would have produced a more impressive-looking v0.1.0 and a less defensible one.

Alternative 1: ship one or two write tools to demonstrate the harness. A v0.1.0 that includes `bind-this-quote` or `post-this-payment` would have been a louder announcement. It would also have required exercising the harness's policy stage, approval stage, and rollback stage against a real tenant — none of which the day's clock allowed. Shipping write tools without the harness fully exercised would have meant shipping write tools whose safety story was an unverified assertion. The credibility cost of that — particularly in a regulated insurance context — would have outweighed the announcement value.

Alternative 2: ship the full PRD's 39-tool catalogue as read-only stubs. A v0.1.0 with 39 tools registered, even if most were unimplemented, would have looked like more progress on the catalogue. It would also have meant shipping a surface that an operator could call and get back a *not yet implemented* error from. That is the kind of artefact that erodes trust: the prospect tries the tool that matters most to them, gets a stub error, and concludes the product is not real. Five tools that all work end-to-end against a real PolicyCenter shape are a stronger artefact than 39 tools where 34 are scaffolds.

The lesson generalises: when shipping an enterprise integration on a one-day clock, the *shape* of the cut matters more than the *surface area*. A small, tight, fully working surface is a credibility artefact. A large, partial, mostly-stubbed surface is a credibility liability. Five working PolicyCenter tools is the right v0.1.0 cut for both. Brooks' *No*

[112] Haber, S., & Stornetta, W. S. (1991). How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2), 99–111. https://doi.org/10.1007/BF00196791

Silver Bullet warning applies in reverse[113]: the temptation to maximise feature count is the *accidental* impressiveness; the operator's actual problem is the *essential* one, and a small surface that solves it is worth more than a large surface that does not.

Tests as a precondition, not a follow-up Fifty-four tests pass across the foundation packages in v0.1.0. That number is not a vanity metric — it is the gating condition for letting six packages ship together. Each foundation package has its own test suite covering its public surface in isolation, plus a small set of integration tests that exercise the contract between packages (e.g., that `@gw/mcp-runtime` correctly threads a JWT from `@gw/auth` into a tool call, that `@gw/audit`'s hash chain survives a write under load, that `@gw/client-sdk`'s idempotency keys are deterministic across retries).

The discipline that mattered: no foundation package was allowed to merge without its own README, its own tests, and its own CI green. A package that ships without tests becomes a package that other packages have to defensively wrap, which destroys the composition story. Holding the line at the package level — every package independently green — is what kept the parallel construction from collapsing into a tangled coupled mess at the end of the day.

The Karate contract layer is the next rung up. Once it is exercised against a real sandbox tenant, a Cloud API schema drift will surface as a contract-test failure during CI rather than as a production runtime error against a customer's tenant. That gap between *shipped contract layer* and *exercised contract layer* is named explicitly in the roadmap so an operator reading the repo knows what coverage they actually have.

The audit register closed the same day The audit follow-ups from the GW-1.9 register all closed the same day across four themes:

- #74: manifest schema cluster — tightened the plugin manifest schema and resolved drift between declared and actual capabilities.
- #78: audit-DB role separation — the role that writes audit rows has insert-only privilege, separate from the role that reads them.
- #73: doc-set hygiene — cross-reference fixes and stale-link cleanup across the blueprint set.
- #81: final theme — closed the rest of the GW-1.9 register.

A separate librarian-audit fix landed in #44: D-016, a rename-lag patch applied across the PRD and memos so that a terminology rename done mid-day did not leave a half-renamed surface for later readers to trip on. The whole GW-1.9 register went to zero open items by end of day.

The librarian KB itself deserves a moment. The five memos — `005-DR-REF-guidewire-public-resource` (the catalogue of public Guidewire reference material worth pinning), `006-DR-MEMO-mcp-safety` (the safety patterns that informed the harness), `007-DR-MEMO-carrier-vocabulary` (the vocabulary mapping that became D-001), `008-DR-MEMO-guidewire-api` (the Cloud API surface analysis that informed the SDK), and `009-DR-MEMO-harness-runtime` (the harness runtime design that became E3) — are reachable both through the blueprint's

[113] Brooks, F. P. (1987). No Silver Bullet — Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4), 10–19. https://doi.org/10.1109/MC.1987.1663532

narrative reading order and through the librarian's topic-keyed index. That dual access matters because the blueprint is the right entry point for a new contributor reading the architecture top-down, and the librarian KB is the right entry point for a contributor who needs to re-find a specific decision later. Same content, two indexes, no duplication.

Closing the audit register the same day the cut shipped matters because an audit register left open is a known-bad signal to anyone reading the repo as a credibility artefact. A v0.1.0 with zero open audit items is a different artefact from a v0.1.0 with a backlog of *we'll fix this later* items. The operator reading the repo can take it at face value.

The implication for shipping enterprise integrations The composability of the v0.1.0 cut is the point. The architecture decisions, the package boundaries, the harness pattern, the audit semantics, the JWT propagation — none of these were retrofitted. Each one was a written commitment in the blueprint before any package code merged, which is what made the parallel construction tractable.

For anyone trying to ship enterprise integrations on a similar clock, the pattern that travels:

1. **Write the architecture as a committed artefact before writing the packages.** Decisions like *actor identity propagates through every call* or *audit failure rolls back the write* cannot be retrofitted without reworking every tool. Pre-commit them in writing. (Parnas's argument[114] in slightly newer vocabulary.)
2. **Make the design doc the contract between packages.** Six packages cannot be built in parallel unless the contract between them is locked into a document everyone can read. The blueprint isn't a deliverable; it's the substrate.
3. **Scaffold the safety pattern even when you have no writes yet.** A harness library that exists in v0.1.0 with zero callers is a hard constraint on every future write tool. A harness library that doesn't exist is a constraint on nothing.
4. **Local-first changes the sales motion.** A repo that runs entirely on the prospect's own infrastructure collapses the procurement cycle. The architecture decision and the go-to-market motion are the same decision.
5. **Compose runtime and operator know-how on the same release clock.** The MCP server and the skill pack are the same product viewed from two angles. Shipping one without the other ships half a product.
6. **Pick the cut shape that earns trust, not the cut shape that maximises surface.** Five working tools beat thirty-nine stubs. A foundation that holds beats a feature that breaks. The shape of the v0.1.0 cut is the prospect's first impression of what the product *is* — make that impression accurate. (Brooks' essence-vs-accidents framing[115] applied to release scoping.)
7. **Close the audit register the same day.** A v0.1.0 with a backlog of *we'll fix this later* items is a different artefact from a v0.1.0 with zero open items. Operators

[114] Parnas, D. L. (1971). Information Distribution Aspects of Design Methodology. *Proceedings of the IFIP Congress.* https://doi.org/10.1184/R1/6606470.V1

[115] Brooks, F. P. (1987). No Silver Bullet — Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4), 10–19. https://doi.org/10.1109/MC.1987.1663532

reading the repo as a credibility artefact can take the second one at face value and have to discount the first.

The v0.1.0 cut is foundation work — by design. The real test is whether the foundation holds when v0.2.0 starts wiring real writes through the harness and the first carrier-side engagement starts pressure-testing the on-disk profile loader against a live tenant role model. Three things are likely to surface:

- **Policy expressiveness.** The harness's policy stage will need to express tenant rules that do not reduce to *role + dollar threshold*. Real carrier rules involve combinations of underwriting authority, line of business, geographic restrictions, treaty constraints, and time-of-day. The policy language has to be expressive enough to encode those without becoming a Turing-complete DSL nobody can review.
- **Profile schema durability.** The `--profile` YAML schema is opinionated about how tenant connection details, role mappings, and per-tool overrides are structured. The first real tenant may push on that schema in ways the synthetic test fixtures did not predict. Schema versioning is in place; schema migration is not yet rehearsed against a live profile.
- **Audit-row volume.** The hash-chain audit table works fine at low volume. At a real carrier's submission flow rate, the chain's tip-anchor question becomes pressing, and the read-query patterns for the compliance team's audit access surface real index requirements. Both are anticipated in the roadmap; neither is exercised yet.

Each of those is a known unknown that the v0.1.0 foundation was deliberately built to allow room for. None of them requires rewriting the package boundaries. That property — extensibility at the seams that matter — is the test of whether the architecture decisions held up. The next post in this arc covers that test when it happens.

The carrier-native vocabulary discipline, the harness pattern, the local-first trust model, the audit-as-precondition rule — none of these are novel in isolation. The composition is what is load-bearing. v0.1.0 ships with all of them already wired together because the blueprint forced the wiring decisions to land before the code did. Anyone reaching for the same shape on a similar clock will benefit from doing the blueprint work first, even when the instinct is to skip it and start coding.

The repo is at github.com/jeremylongshore/guidewire-mcp-for-claude. The architecture diagram is at guidewire-mcp.intentsolutions.io. Plugin install is `/plugin install jeremylongshore/guidewire-mcp-for-claude` from any Claude Code session. Sandbox tenant credentials are required to exercise the tools end-to-end; the README documents the four environment variables and points to the Guidewire developer portal for sandbox provisioning.

Issue tracker, blueprint set, and roadmap are all in the repo. The next cut is E2.5 (BillingCenter and ClaimCenter read tools) plus the first harness-wired write tool, on a clock that is yet to be named.

Related Posts

- Claude Code Plugin Marketplace Launch — the marketplace this plugin and its sister-pack ship through
- Building a Production-Ready Research Tool That Outperforms Anthropic's — another MCP-fronted system, different domain
- IRSB Ecosystem — companion ecosystem hub for the broader Intent Solutions integration story

---

References

Every source in this paperback is verified against Semantic Scholar (when indexed) or anchored to a stable DOI / URL. See `content/citations/` in the source repository for the BibTeX corpus and per-source verification metadata.