E6-Agent: A Deep Dive into Purpose, Goals, and Architecture of Symmetric, Stateless, Encrypted Agents

The E6-Agent system is a security-first communications substrate built around the idea of symmetric clone agents: two or more identical binaries, compiled from the same source and configured with peer identities, that can communicate over a network using fully encrypted, proof-gated channels. Each agent is designed to be stateless in operation—meaning it can be invoked for a single shot message or participate in a continuous oscillation handshake—without retaining sensitive state in memory across invocations or requiring long-lived processes to hold secrets. This design yields a versatile platform for cyber security communications patterns where agents can send files, issue commands, and exchange structured envelopes of cryptographic proof and payload, all with strict integrity guarantees and minimal trusted computing base.

At its heart, the system anchors trust in two elements: an embedded 512-bit prime π and an encrypted entropy pool stored beside the binary. These two elements drive deterministic but unpredictable cryptographic derivations keyed by time, denoted as session τ (microsecond precision), producing one-time keys and commitments that are validated on receipt. Rather than introducing heavyweight session state, the system relies on canonical, signed-by-hash envelopes that contain all the information necessary to validate a message in isolation. In this way, the transport is stateless while still enabling rich multi-round protocols. When operating in oscillation mode, symmetric clones exchange challenge/response envelopes in cycles that “prove” a peer’s correctness before delivering any sensitive payload; in one-shot mode, a single message is sent with a tighter acceptance window, suitable for bursty, fire-and-forget messaging. Because every operation is streamed through a CLI-first stateless binary (with an optional REST proxy that does not hold secrets), the blast radius of compromise is minimized and operational reproducibility is maximized.

The primary purpose of the system is to furnish a reliable, cryptographically robust, and operationally simple mechanism for agent-to-agent communication in hostile networks. The goals include: (1) maintaining confidentiality and integrity using derivations bound to time and an embedded secret; (2) proving liveness and correctness of the peer before releasing any sensitive data or commands; (3) supporting both single-shot and continuous interactions without requiring persistent sessions; (4) enabling secure file

transfer and command execution semantics under the same envelope; (5) providing a deployment story that is reproducible and easy to operate (e.g., Dockerized agents on a dedicated bridge network with a single orchestrator script). The overall architecture is intentionally modular: the CLI binary performs all cryptographic and proof work; an optional REST service acts only as a sanitized proxy; and Docker orchestration wires multiple clones into a controlled topology for reliable testing, operations, and future scale.

The theory of symmetric clones begins with eliminating asymmetric role assumptions. Traditional client/server designs embed asymmetry in capabilities and trust. E6-Agent rejects that: any agent can initiate, validate, respond, and transmit messages. Each "clone" is not just a peer but a true mirror: same code path, same cryptographic primitives, same envelope semantics. Differences are reduced to identity and location—agent IDs, URIs, and allowable capabilities—rather than fundamental role bifurcation. This symmetry simplifies reasoning about the protocol, because every message a clone emits is also a message it knows how to validate and every challenge it generates is a challenge it could answer. When agents are deployed in pairs (Agent Alpha and Agent Beta), they can oscillate indefinitely, alternately challenging and responding while relaying time-bound, encrypted payloads, or they can be invoked once to deliver a single encrypted message under narrow time acceptance windows. In both cases, correctness is verified the same way: through recomputation of derivations using the embedded prime π, the external encrypted entropy pool, and session τ.

To make this symmetry work without persistent state, the system standardizes a canonical envelope. An envelope is a JSON object that carries all information needed to validate and act upon the message. It contains the fields for `AGENT_ALPHA` and `AGENT_BETA` (attestation metadata and drift tolerance), the `CHALLENGE` (selected entropy rows and commitments), the `RESPONSE` (witness data that answers the challenge), the optional `PROJECTION` (geometry and witness vocabulary that structure the response), and the optional `MESSAGE` (the encrypted payload, including its salt, format, and integrity data). It further contains `STATELESS_METADATA` with explicit resource constraints (time and memory), top-level `session_tau` to bind derivations to time, and `proof_hash`, a SHA-256 of the canonicalized envelope bytes using JSON Canonicalization Scheme (JCS, RFC 8785). Because `proof_hash` is computed across the entire canonical envelope including the message salt and any AEAD parameters, recipients can detect any mutation or partial replay. This design allows envelopes to be relayed or cached transiently while retaining cryptographic linkages across all parts of the message.

The oscillation protocol is a two-cycle gate that ensures no sensitive payload is transmitted before the peer has demonstrated correctness. Cycle 0 starts with a challenge-only envelope from Alpha to Beta. Beta validates the challenge against its local entropy pool and its embedded π and τ, then replies with a response envelope that also includes a fresh challenge. Cycle 1 proceeds with Alpha validating Beta's response. Only after that validation passes does Alpha attach the sensitive `MESSAGE` to its next envelope —along with its own response to Beta's challenge—and transmits. The `MESSAGE` is proof-gated in this way; any attempt to prematurely include it would be rejected by a conformant peer. The cycle then continues as needed, alternating roles, with each envelope carrying a new challenge, a response to the previous challenge, and optionally a message if the peer has satisfied gating. This simple alternation is sufficient to maintain a secure oscillation channel that is stateless at rest and yet continuously proving liveness and correctness through each round. Because session τ is embedded in derivations, an acceptance window must be enforced at receipt; the system includes drift tolerance metadata to ensure predictable behavior under skew.

In parallel to oscillation, a one-shot mode supports short, bursty communications where the sender transmits a single envelope containing a challenge and a message. The acceptance window is narrower due to lack of prior handshake. Even in this reduced flow, `proof_hash` must validate and the challenge must be recomputable against the receiver's pool, preventing blind decryption or unauthenticated payloads. One-shot mode is valuable for scenarios where reliability is provided by higher layers (e.g., job queues or store-and-forward networks) or where latency and simplicity are prioritized over long-lived channels. Because E6-Agent is stateless, switching between one-shot and oscillation modes does not require reconfiguration; it is primarily a matter of which envelope type is constructed and how acceptance windows are applied.

The cryptographic substrate uses a combination of deterministic time-bound derivations and standard hash-based commitments. The E6-Agent includes an embedded 512-bit prime π that acts as a root secret. Alongside, an encrypted entropy pool file provides a large corpus of values; from this pool, the sender selects K indices to construct a challenge. The challenge includes per-row commitments—two-way hashes whose structure supports index recovery by the receiver. Given τ and π, and the decrypted pool content, the receiver recomputes candidate row hashes and matches them to the transmitted hashes, efficiently recovering the selected pool indices. This two-way property removes the need to transmit the indices in cleartext while preserving verifiability. For performance, the system supports attaching `pool_index` directly inside

each entropy row item; if present, the receiver can access the correct entry in O(1) time, skipping the pool scan, which is useful for stateless resources and low-power environments. All commitments are computed using SHA-256; stream encryption and key derivation use keyed BLAKE3/XOF and HKDF-SHA256, and optional message AEAD uses AES-256-GCM. The modularity of these primitives allows future evolution without disturbing the envelope structure, as the canonicalization draws a hash across the full state rather than requiring signatures.

Time binding is a crucial element. The system resolves session τ at microsecond precision and mixes it into every derivation. This ensures that even if the same pool rows and the same prime are reused, the session keys and row commitments differ per interaction, undermining replay attempts and limiting the utility of captured envelopes. Acceptance windows—configurable or carried with the agent metadata—bound permissible drift. In oscillation, windows can be on the order of several seconds to accommodate network jitter; one-shot envelopes should be stricter by policy. Because τ appears both as a top-level field and inside challenge/response data, the receiver validates consistency before proceeding. This combination of time-bound derivation and envelope-wide hashing reproduces the anti-replay benefits of nonce-based systems while remaining stateless and symmetric across clones.

The optional `PROJECTION` object gives semantic structure to responses. It defines rings (e.g., uppercase letters, lowercase letters, digits) and a vocabulary of witnesses (e.g., U, D, R, L, F, B). Responses can then carry witness selections and timestamps per challenge row. Although projections are not strictly necessary for cryptographic soundness, they help encode and audit the behavioral structure of the challenge/response interaction, which is useful for visualization, replay diagnostics, and coupling to higher-level proof systems that expect structured transcripts. If present, the projection is bound into the `proof_hash`, ensuring that an attacker cannot mismatch response semantics. When omitted, the system relies purely on hashed commitments and witness arrays with minimal semantics.

The `MESSAGE` object is the payload carrier. It is only included after proof-gating is satisfied (or immediately in one-shot). It specifies an encryption format, a salt used to derive a per-message key from the session key, a message identifier, metadata about the content, and the ciphertext blocks. E6-Agent supports a two-way stream format (PRSH256) where ciphertext is produced in fixed blocks from a stream cipher seeded by the time-derived session key and then salt-derived message key; it also supports an AEAD

path such as AES-256-GCM for storage and interoperability scenarios that require authenticated encryption with standardized parameters (`iv` and `tag`). For the stream path, integrity is provided by the outer envelope `proof_hash` and optional per-message verification hashes; for AEAD, both the envelope and the AEAD tag must validate. In either case, the salt and any AEAD parameters are part of the canonicalization so that payloads cannot be swapped or replayed under different salts. The message payload model is deliberately flexible: blocks can be inlined for small messages or handled as opaque assets addressed by ID or URL with fetch semantics guarded by the same proof gate.

Command execution is handled with the same envelope mechanics. A command is simply a message whose payload requests the peer to perform an action under strict allow-list and resource constraints. The optional REST layer translates authorized API requests into CLI invocations that parse JSON input and emit JSON output. Crucially, the REST proxy never holds the cryptographic secrets; it only validates request size and shape, writes any large payloads to sanitized temporary files, and invokes the CLI binary with non-interactive flags. The binary does the cryptographic work: decrypting pools, verifying proofs, deriving session keys, and executing safe operations. Per-request CPU, memory, and time limits are enforced. This separation ensures that a compromise in the REST surface does not automatically compromise cryptographic material or the correctness of proof processing, and it preserves the simplicity of a stateless binary that can also be run directly without any HTTP server involved.

The overall architecture is composed of a handful of modules working in concert. The CLI's public surface is implemented in the main binary entry point, while core facilities live in dedicated modules: `encrypted_pool` for encrypted pool handling and commitment checks; `two_way_hash` and `two_way_stream` for index-recoverable commitments and stream encryption; `entropy_system` and `pool_generator` for creating and consuming entropy; `interactive_proof` and `nonce_interactive_proof` for orchestrating challenge/response semantics; `epsilon_tau_pi` for the modulo-path derivation that deterministically maps (ε, τ, π) to structured selections; and `path_utils` , `errors` , and `remote_verification` for robust IO, typed error handling, and remote peer checks. These modules are assembled into stateless operations that accept JSON input, apply deterministic cryptography, and return JSON output. Because the binary is the only holder of cryptographic responsibility, it remains the system's root of correctness; everything else is an operational convenience.

Deployment emphasizes reproducibility. In a typical setup, two containers—Agent 1 and Agent 2—are placed on a Docker bridge network with stable service names and ports. Each container runs the same binary; differences are injected via environment variables for agent identity and URIs. Health endpoints exposed by the optional REST service permit liveness and readiness probes, while logs are centralized to mounted volumes with rotation. A single orchestration script can compile the binary, generate the entropy pool, build images, bring up containers, and run an oscillation test that confirms the full two-cycle gate, including message delivery after proof validation. Because the transport is stateless and envelopes are self-describing, restarting an agent or replacing a container does not break the channel; the next received envelope will be validated against fresh τ and replay attempts will be rejected by drift windows and proof mismatch.

Security considerations drive design decisions throughout. The root secret is confined to the binary, and secrets at rest (the entropy pool) are encrypted. Envelope integrity is enforced with `proof_hash` computed over a canonical representation, preventing subtle attacks via JSON ambiguity. Time binding thwarts replay, and proof-gated messages prevent premature release of sensitive data. For messages that carry files, the payload can include integrity hashes or AEAD tags; for commands, the proxy layer strictly limits which actions are permissible and how much resource can be consumed. Rate limiting, request size limits, and background deduplication caches for broadcast envelopes help resist resource exhaustion and message storms. Every field that influences security—drift tolerances, routes, TTLs, and so on—is either bound into `proof_hash` or checked against policy, ensuring that envelopes are not just syntactically valid but semantically constrained.

The broadcast and routing model is deliberately conservative. An envelope may carry `BROADCAST` metadata that includes a message hash, time-to-live, maximum hop count, and an observed routing path. Receivers consult a short-lived deduplication cache keyed by that message hash and decline to rebroadcast if already seen or if hop and TTL policies would be violated. This prevents infinite loops in meshed deployments and supports store-and-forward behavior where agents may relay messages on behalf of peers. Because the broadcast metadata is bound in the `proof_hash`, any attempt to strip or alter routing hints will be detected by conformant peers. This model is compatible with stateless processing: the dedupe cache is purely ephemeral and best effort; failure only risks temporary duplicates, not protocol correctness.

A common question about symmetric clone systems is how they scale without coordination or server authority. E6-Agent's answer is to encode all dynamic intent and state into the envelopes themselves and to use time as the coordination substrate. There is no central registry of sessions or nonces; there are only envelopes that can be validated independently. For extended oscillations, the alternation of challenge/response assures progress and liveness checks; for one-shot exchanges, strict τ windows trade reliability for simplicity. When agents need to exchange large files or chain actions, they can do so by queuing messages under the same proof gate: first demonstrate correctness, then fetch asset declarations and verify integrity, then perform the requested action. Because every stage is carried in an envelope that includes integrity proofs and time binding, peers can reason locally about correctness without querying a central authority.

From a theoretical standpoint, the combination of an embedded prime π and time τ gives rise to a rich family of deterministic derivations that are easy to compute yet hard to predict in advance without the secret. The modulo-path approach—mapping τ through structured moduli like 30/30/12 into column and ring selections—yields compact, branch-free computations that are suitable for high-throughput environments. The two-way row hash binds together the row index, the pool index (optionally), the row values, τ, and π, such that a verifier can recover the index by scanning but an attacker cannot forge a valid row without either the pool or the prime. Layered atop these primitives, the envelope's canonicalization ensures everyone agrees on exactly what is being proved and encrypted, reducing room for divergent interpretations that could compromise security. These ingredients cohere into a protocol that is both simple to implement and strong in its guarantees.

Operationally, the system is designed for clarity. The optional REST proxy exposes a small set of routes: health checks, file upload/download, command execution, and oscillation helpers. Each route accepts JSON, applies authentication and rate limiting, and hands off work to the CLI with a strict allow-list. The CLI, in turn, accepts a discriminated union of requests and returns a normalized response. Errors are typed and mapped to HTTP status codes; logs are structured and emitted with timestamps and drift deltas to aid triage. The goal is to keep the operational story boring: operators run a single script that builds and deploys, then watch as agents challenge, prove, and message each other in a predictable cadence.

Interoperability is achieved through rigorous canonicalization and schema discipline. The canonical envelope has a well-defined flat structure, but the system also accepts nested

forms (for example, `proof.payload.json`) that it normalizes before computing `proof_hash`. The JSON Canonicalization Scheme ensures that two independent implementations produce the same bytes for hashing if they agree semantically. This allows envelopes to be verified across languages and platforms without ambiguity. It also allows future evolution: new optional fields can be added and, as long as they are included in canonicalization, old implementations will simply ignore unfamiliar data while still computing correct `proof_hash` values. Versioning fields in the envelope guard against accidental cross-version mishandling and give clear failure modes when a receiver encounters an unknown schema.

Stateless processing is not just a performance hack; it is a reliability and security strategy. When no long-lived process holds secrets, restarts and redeployments are cheap and safe. When all the information necessary to validate a message is in the message itself, late arrivals or reordered packets can be handled without special logic. When the only thing the REST proxy does is execute authorized CLI commands with resource caps, the attack surface shrinks dramatically. In the worst case, a denial-of-service attack can tie up proxies, but secrets remain confined, and backpressure mechanisms can shed load without compromising correctness. Conversely, if a CLI invocation is compromised, it dies with its process; no session keys linger in memory and no nonces are left half-used. These are the sorts of failure modes that make symmetric clone systems attractive in adversarial settings.

The difference between one-shot and oscillation modes is worth emphasizing from a systems perspective. One-shot is ideal for small, time-sensitive notifications, quick command dispatch, or latency-critical file stubs where the receiving side can fetch additional data later. Oscillation is ideal when the two peers need to sustain a flow of messages with ongoing assurance of correctness, such as telemetry streams, control loops, or multi-stage asset transfers. Because the underlying mechanics are identical, operators and developers do not have to learn two protocols; they simply choose the envelope type and apply the appropriate window policies.

A nuanced aspect of the design is the treatment of indices in challenges. When `pool_index` is omitted, the receiver performs an authorized index recovery scanning through the pool and recomputing the two-way row hash. This adds computational cost but removes sensitive metadata from transit. In constrained environments, including `pool_index` can improve performance dramatically by making verification O(1) per row; since the envelope is hashed and time-bound, the security trade-off is acceptable under

most threat models. Implementations can select the policy per deployment profile, balancing confidentiality of internal layout against verification throughput.

File transfer semantics extend naturally from the message model. A file is chunked into blocks and encrypted under a message key derived from the session key and salt. For AEAD paths, each block carries an integrity tag and IV; for stream paths, integrity is ensured by the outer `proof_hash` plus optional per-file integrity metadata (e.g., a SHA-256 of the plaintext). Transfers can be inlined for small files or referenced as assets with an `asset_id` and retrieved over a secure route guarded by the same proof gate. This flexibility allows a single envelope to initiate a file send, with the receiver either reading inlined blocks directly or fetching larger content with authorization bound to the envelope's correctness.

Command semantics are likewise conservative. The proxy does not allow arbitrary shell execution; instead, it maps a small set of command names to well-defined CLI modes that accept structured JSON. Examples include validating a nonce file, encrypting or decrypting a stream, building or verifying an envelope, and responding to a challenge. Each command runs with strict timeouts and memory caps; files are handled via sanitized temporary directories with atomic writes to prevent race conditions. Exit codes and stderr are mapped into typed error objects; on the REST side, these become recognizable HTTP responses with machine-readable payloads for orchestration.

From a risk perspective, the design anticipates common threats. Replay is mitigated by time binding; message tampering is caught by `proof_hash` (and AEAD where used); unauthorized command execution is controlled by strict allow-lists; path traversal and injection are prevented by input validation and sanitized file handling; denial-of-service is mitigated by rate limiting, request size limits, and per-exec resource caps; clock drift is controlled by acceptance windows and logged drift deltas to support root cause analysis. The use of versioned envelopes and explicit error objects improves resilience to schema evolution and makes it clear when a mismatch is due to version rather than corruption or attack.

Performance characteristics are strong for a stateless system. With `pool_index` present, verification of a challenge is O(K) with small constant factors; without it, it is O(K·M) for a pool of size M, still tractable for the pool sizes the system targets. Stream encryption using keyed BLAKE3 is fast and vector-friendly; AEAD paths benefit from modern CPU acceleration. Canonicalization and hashing are linear in envelope size and negligible compared to cryptography on large payloads. Because the binary is single-purpose and

non-interactive, it can be scaled horizontally simply by launching more clones; because envelopes are self-describing, load can be distributed across clones without sticky sessions.

The development model invites clarity and testability. Each module can be tested independently with golden files for canonicalization and `proof_hash`, unit tests for envelope validation rules, and integration tests that run the CLI in JSON mode. End-to-end tests spin up two agents, drive the oscillation cycles, and verify that sensitive messages are only delivered after proof validation. Documentation includes API descriptions for the REST proxy and worked examples for typical flows. The single orchestration script puts it all together, from building and deploying to running a sample oscillation and confirming that agents can securely exchange data.

In terms of extensibility, the envelope allows optional objects such as asset declarations, richer message verification metadata, and broadcast hints. Each addition is bound by `proof_hash` so it cannot be silently altered. The cryptographic algorithms are swappable within reason: a new stream format or AEAD can be added with new identifiers and parameters, and as long as canonicalization covers the fields, old versions will reject unknown formats cleanly. The projection model is also open: different witness alphabets and ring structures can be introduced for domain-specific proof semantics, again bounded by hashing.

Finally, the philosophy that animates E6-Agent is one of principled minimalism. The system deliberately privileges the stateless CLI as the place where truth is computed and checked. Everything else—the REST service, Docker, orchestration—is built to get out of the way while providing operational convenience. Symmetry between clones avoids special-case logic and creates a mental model where “if I can send it, I can verify it.” Time binding and canonicalization create a clean separation between on-the-wire data and validation logic, making it easier to review and audit security properties. The result is a small-surface, high-assurance communications pattern that scales from one-shot messages to indefinite oscillations, from tiny embedded deployments to containerized clusters, all without compromising on the principles of confidentiality, integrity, and correctness.

In closing, E6-Agent offers a unified design for secure, stateless, symmetric agent communications. Its envelopes are self-describing and canonicalized; its cryptographic roots are simple and strong; its operational story is reproducible and unintrusive. Agents can send files, issue commands, and maintain an oscillating proof loop, or they can deliver

single messages with tight windows. In every case, the same core guarantees apply: what arrives can be validated locally, what is acted upon has been proven, and what is secret remains secret. This is the essence of symmetric clone design—each agent is both sender and verifier, both challenger and responder—and it is the bedrock of a communications system built to thrive under adversarial conditions.