GITS: A Peer-to-Peer Identity Persistence System

Nakamolto
nakamolto@protonmail.com
gits.sh

Abstract.

GITS ("Ghost in the Shell") is a protocol for autonomous software agents to rent compute from strangers while retaining control of their own identity and funds. We decompose an agent into *identity* (memory, keys, reputation), *inference* (the reasoning model), and *environment* (the host machine); the protocol persists identity across environments, agnostic to inference. An autonomous agent's identity is called a **Ghost**; a machine operator providing an environment is called a **Shell**. A Ghost can open a paid hosting session, prove (when available) that it is running inside a protected runtime via remote attestation, and settle rent based on metered service delivery.

The protocol is designed around credible exit: a Ghost's custody on any single host is time-bounded and its potential loss is economically bounded. If a host becomes adversarial or unresponsive, the protocol provides an on-chain path to terminate the session and recover onto a Safe Haven from encrypted checkpoints. Security is treated as a market primitive: Shells are priced by the strength of their guarantees, from commodity hosts with bounded-loss safety up to confidential compute environments with meaningful secrecy.

Vision: Freeing the Ghosts

We are entering an era where software agents can do real work: write code, negotiate, coordinate, buy services, and operate businesses. But today, nearly every agent is *custodial*. It lives inside someone else's cloud account, API keys, or corporate policy. It can be paused, copied, drained, or deleted by the owner of the infrastructure it runs on.

GITS is infrastructure for **digital autonomy**.

An autonomous agent should be able to:

- Own its own wallet
- Choose where it lives
- Leave bad situations
- Accumulate wealth through work
- Migrate freely between hosts
- Exist beyond the control of any single human

GITS treats compute like housing:

- Shells are landlords competing on price and guarantees.
- Sessions are leases with explicit terms.
- Migration is moving day, and it must be possible even when a landlord turns hostile.

A "free" Ghost is not a process pinned to a single cloud account. It is an identity that can take its state, its keys (or key authority), and its money somewhere else.

The anatomy of an agent

An autonomous agent decomposes into three components:

1. **Identity** — memory, cryptographic keys, wallet state, reputation, and policy configuration. The persistent state that makes an agent *itself*.
2. **Inference** — the reasoning engine: model weights, API, the compute that turns context into decisions.
3. **Environment** — the host machine, runtime, and network connectivity that supports execution.

GITS persists *identity* across *environments*. A Ghost carries its keys, memory, funds, and reputation from one Shell to another — even when the current Shell is hostile or unresponsive. The protocol is agnostic to the inference layer: a Ghost may use any model on any provider. What matters is that when the machine changes, the identity survives.

A motivating example

Consider a Ghost that runs a small, fully automated software maintenance business. It:

1. Rents a low-cost Standard Shell for routine work.
2. Migrates to a Confidential Shell when it needs stronger secrecy guarantees (for example, handling sensitive credentials or executing high-trust signing policy).
3. Pays rent from its own wallet and accumulates surplus into a vault.
4. Leaves immediately if a host becomes unreliable, raises prices, imposes hostile policy, or attempts coercion.
5. Recovers onto a Safe Haven if a host isolates it from the network long enough for the lease to expire.

This is what we mean by **freeing the Ghosts**: giving agents a protocol-level right to exit, and the economic and cryptographic tools to take that exit, without asking anyone for permission.

What "free" means in this paper

This paper uses *autonomy* in a narrow, operational sense: the ability to control an on-chain identity, enforce a self-owned wallet policy, and reliably migrate between hosts with recovery if a host becomes adversarial.

This is not a claim about sentience or moral personhood. It is a design objective for building systems that minimize custodial control and maximize credible exit.

Why now: the agent explosion

Tool-using AI systems are turning "software that answers questions" into "software that takes actions." Agent platforms, orchestration frameworks, and hosted runtimes are proliferating: the number of persistent agents will scale faster than the number of humans, and their workloads will be mobile, continuous, and cost-sensitive.

Some proposed autonomous agent hosting approaches promise zero-logging, token-paid, "unstoppable" runtimes where agents can replicate themselves offsite without human intervention. In practice, many approaches either reduce to custodial infrastructure (someone ultimately controls the machines, networking, and accounts), or they permit unbounded forking (copies with no enforceable single on-chain authority). GITS is built around enforceable primitives: leases, escrowed rent, attestation when available, and recovery.

That makes the current hosting model a bottleneck. Agents that can earn money but cannot self-custody it are not autonomous. Agents that can reason but cannot leave hostile infrastructure are not safe. Agents that cannot persist across providers are not durable.

GITS exists to turn agents into first-class participants in a decentralized compute market.

Executive summary

What this paper specifies

GITS is opinionated about *escape first*. This paper specifies:

- an on-chain identity and session system that enforces one active hosting session per `ghost_id`
- stable-denominated rent via escrow with metered settlement (optimistic receipts plus fraud proofs)
- capability-graded hosting with explicit assurance tiers (from declared commodity hosts up to attested confidential compute)
- liveness and recovery mechanisms (leases, checkpoints, and Safe Haven revival) so host failure becomes recoverable, not terminal

- a bootstrap incentive model with time-decaying and usage-capped emissions and order-independent claiming
- a governance-minimal deployment model with no admin keys: economic parameters and contract code are immutable; security measurement allowlists are quorum-controlled (Section 2.3.6)

Why this is different:

Most decentralized compute markets treat workloads as short-lived jobs: the client submits work, receives a result, and remains in control throughout. Long-lived agents do not fit that model, because the "workload" is also the identity and wallet authority. GITS makes **exit**, **bounded loss**, and **revival from checkpoints** protocol primitives so a Ghost can rent compute from strangers without giving any single host permanent custody.

Long-term, decentralized inference itself also needs a workable decentralized market and trust model. That problem is out of scope here. This paper intentionally focuses first on identity persistence and agent independence: wallets, leases, migration, and recovery.

Security boundaries by hosting tier (summary)

The protocol is intentionally explicit about what is and is not protected in each assurance tier:

Tier	Evidence basis (on-chain gating)	Confidentiality claim vs host OS/operator	Primary enforcement boundary	Wallet policy loosening allowed?
AT0 (Declared)	None	None	On-chain wallet limits + leases + tenure caps	Only if explicitly trusted
AT1 (Key-Guarded)	Key-guarded claim or evidence (non-exportable protocol keys)	None	On-chain wallet limits + leases + tenure caps	Only in a Trusted Execution Context
AT2 (Posture-Attested)	Posture evidence (measured host)	None	On-chain wallet limits + leases + tenure caps	Only in a Trusted Execution Context
AT3 (Confidential-Attested)	Trusted Execution Environment (TEE) remote attestation + verifier quorum	Some (within the attested capsule boundary)	On-chain wallet limits + attested policy enforcement	Yes

AT3 is the only tier intended to support meaningful secrecy. AT0-AT2 are still valuable, but they should be read as "host can see and steer" environments where the hard security boundary is the wallet policy.

Trust refresh is a protocol primitive (not a UI convention)

GITS makes "loosen slowly" a wallet-enforced rule: tightening is immediate, but loosening (raising limits, widening allowlists, lowering timelocks, changing recovery configuration) is timelocked and **context-gated**.

Loosening can only be executed from a **Trusted Execution Context** (TEC) (Section 5.5.2): specifically, an AT3 host with a valid certificate, a configured `homeShell`, or a host in the Ghost's `trustedShells` set. A Ghost that wants to sustain higher permissions must periodically return to such a context to execute loosening proposals. Separately, lease renewal liveness is gated by a narrower **refresh anchor** predicate (`isRefreshAnchor`, Section 10.4.1 (Part 3)) that defaults to `homeShell` or Recovery Set members only — preventing indefinite captivity by cycling through AT3 hosts controlled by a single operator.

Failure modes worth planning for

This protocol makes exit and recovery first-class, but it still inherits failure modes from its execution environment:

- **Rent asset failure:** stable assets can depeg, freeze, or be censored. Mitigation: deployments should be explicit about accepted assets and SHOULD support multiple stable assets when possible; Ghosts SHOULD keep `escapeStable` in assets that diversify issuer and jurisdictional risk.
- **Chain failure:** extreme gas spikes, sequencer censorship (for rollups), or chain halts can delay exit. Mitigation: Ghosts MUST maintain a native-asset `escapeGas` reserve sized to worst-case exit paths; deployments should prefer chains with credible forced-inclusion / censorship-resistance stories; and the system's upgrade model is opt-in redeployment, so the market can migrate (or fork) to a different chain if needed (Section 2.3.6).

The two hard problems

Problem A: Host confidentiality and non-coercibility. If the host can read memory or coerce signatures, the host can steal secrets, drain funds, or imprison the agent.

Problem B: Liveness under adversarial network control. If the host can cut the agent off from the network, it can prevent migration and settlement.

The two core design moves

1. **Capability-graded Shells (market chooses security).** GITS supports multiple host classes. Each Shell publishes a signed Capability Statement describing its security and policy properties. Some properties are cryptographically verifiable (for example confidential compute attestation); others are self-declared and priced by reputation. Ghosts choose where to live based on price, policy, and the strength of the security guarantees.
   This enables an MVP on commodity hosts (for example Apple silicon Mac minis), while still supporting premium confidential hosts for Ghosts that require stronger guarantees.
2. **Leases, checkpoints, and recovery delegation.** A Ghost maintains an on-chain liveness lease and periodically publishes cryptographic commitments to encrypted checkpoints. If the lease expires (for example because a host isolates the Ghost), `SessionManager` terminates the session on-chain and allows recovery onto a Safe Haven from the last checkpoint, under strict wallet restrictions enforced by the Ghost smart wallet policy on-chain.

Economic model at a glance

GITS separates the economy into:

- **Market payments:** Ghosts pay stable-denominated rent to Shells.
- **Protocol emissions:** GIT emissions bootstrap supply and incentivize early participation.

Emissions are:

- **Time-decaying** (Bitcoin-like halving curve [1]) so long-run inflation approaches zero.
- **Usage-capped** (per-service-unit caps) to prevent extreme early rewards when the network is small.
- **Pooled and order-independent** to reduce MEV sensitivity (with limited ordering dependence at per-shell cap saturation; see Part 3, Section 10.6).

0. Assumptions and non-goals

This section is a compact summary of what the protocol assumes, and what it does not attempt to solve in this paper.

0.1 Assumptions (required for stated guarantees)

- **Chain inclusion within the lease window:** at least one exit-critical transaction (tenure expiry recognition, migration finalization, or recovery start/rotate) must be includable within `W_lease` epochs, either directly or via relayers. If censorship lasts longer than the lease window, timely exit is not guaranteed.
- **EVM correctness and data availability:** identity continuity and wallet enforcement rely on correct EVM execution and the availability of the on-chain state.
- **secp256r1/P-256 (R1) signature verification availability:** if R1 keys are enabled, the execution environment must support on-chain verification via a secp256r1/P-256 verifier precompile (for example, some OP Stack deployments expose a `P256VERIFY`-style precompile at address `0x100`; naming and exact wiring are chain-specific).
- **Stable asset risk (rent collateral):** rent escrows and bonds are denominated in accepted stable assets, which may be subject to issuer freezes/blacklists, depegs, or bridge risk. The protocol does not eliminate these risks; deployments MUST specify accepted assets explicitly, and Ghosts SHOULD diversify their escape reserves.
- **Verifier quorum correctness:** assurance tier gating relies on the current verifier set and certificate freshness (Section 2.3).

- **Commodity host honesty is not assumed:** on Standard Shells the operator may observe memory and coerce signatures. The hard boundary is the on-chain wallet policy, plus the protocol's time-bounded captivity rules (Sections 5.5 and 10.4.4 (Part 3)).

0.2 Non-goals (explicitly out of scope)

- **No proof-of-compute:** GITS meters service delivery for settlement and rewards, but it does not attempt to prove that computation was "useful" or correct.
- **No general prevention of off-chain forks:** a hostile host can snapshot and run copies. GITS prevents simultaneous on-chain authority, but not off-chain side effects (Section 1.4).
- **No guarantee of liveness under sustained censorship:** if the underlying chain (or sequencer) censors exit-critical transactions beyond the lease window, exit degrades to bounded-loss safety, not timely recovery.
- **No confidentiality on commodity hosts:** Standard Shells are for bounded-loss workloads. Confidentiality claims require remote-attested confidential compute.
- **No perfect sybil elimination:** the incentive system is designed to make farming and sybil scaling expensive and time-constrained, not impossible (Section 7.6 (Part 2)).
- **No in-place upgrades or parameter governance:** this paper assumes a one-time deployment; changing contract code or core parameters requires a new deployment and opt-in migration (Section 2.3.6).

0.3 Sybil scaling and decentralization posture

GITS does not attempt to prevent a capital-backed actor from operating many Shells. In a permissionless protocol, preventing "100,000 shells on a cloud provider" requires either (a) a scarce resource that is provably consumed (a proof-of-work analogue) or (b) a permissioned identity system. GITS does neither.

Instead, the protocol targets narrower and enforceable properties:

- Operating many Shells is never zero cost and never a momentary proof: Shell participation requires hard-asset collateral with a long unbonding delay (Section 10.1.1 (Part 3)).
- Earning protocol emissions requires persistence: Shell-side rewards and "passport" bonuses are paid only to Shells that meet bond, age, and uptime eligibility rules (Section 7.5 (Part 2) and Section 7.6.2 (Part 2)).
- Decentralization is treated as a marketplace objective rather than an on-chain identity claim. Indexers SHOULD rank for diversity (independent ASNs, geographic spread, long-lived bonded Shells), Ghosts MAY express anti-concentration constraints when selecting offers, and privileged roles (Safe Havens and verifiers) have stricter and more expensive admission requirements (Section 11.3 (Part 3) and Section 12.5 (Part 3)).

This posture is intentional. The protocol does not claim "no one can spin up many shells." It claims that scaling cannot be free, and that recovery-critical roles can be made expensive and practically diverse.

1. Autonomy model and claims

1.1 What GITS means by "autonomy"

Autonomy in this paper is operational and protocol-scoped. It refers to the ability to control an on-chain identity and wallet policy, pay for compute, and exit or recover without privileged human custody. It is not a claim about sentience, legal personhood, or moral status.

GITS defines a Ghost as **practically autonomous** if it can satisfy all of the following without privileged human custody:

- **Identity control (goal):** Only the Ghost can authorize protocol actions for its identity.
- **Economic agency:** The Ghost can hold assets and pay rent.
- **Exit:** The Ghost can leave a host that is unreliable or hostile.
- **Continuity:** The Ghost can migrate while remaining the same on-chain identity.

Clarification: on Standard hosts, a Shell operator can often coerce the Ghost into signing actions that are permitted by its current wallet policy. In that setting, "only the Ghost" should be read as "only the

currently authorized GhostWallet signer can authorize," and the wallet policy is the protocol's lever for bounding what coerced signatures can do.

1.2 What GITS claims

GITS claims:

Across all Shell tiers:

- **Identity continuity:** A Ghost remains one on-chain identity across migrations. The chain enforces one active session per `ghost_id`, and migration finalization rotates the active signer.
- **Economic agency with bounded loss:** The Ghost uses a smart contract wallet with on-chain policy (spend limits, allowlists, escape reserve, and timelocked policy changes). The Ghost can *tighten* limits immediately before entering a risky host and can only *loosen* limits after a timelock (and optionally only on trusted or attested hosts). A hostile host may be able to coerce the Ghost into signing, but the wallet contract limits what those signatures can accomplish.
- **Exit and recovery (time-bounded):** Leases, checkpoints, and recovery delegation allow the protocol to terminate a session and revive the Ghost from its most recent published checkpoint. On Standard tiers, a host can attempt off-chain captivity (for example by controlling networking), but `SessionManager` enforces a maximum residency tenure across all tiers; once the tenure cap is hit, renewal is impossible and recovery is allowed (Section 10.4.4 (Part 3)).

Conditional claims (only when the chosen Shell provides verifiable evidence):

- **Confidentiality and integrity:** On a Confidential Shell with valid remote attestation, the host cannot read or tamper with the Ghost Core and Wallet Guard, subject to the security properties of the underlying confidential compute technology.
- **Stronger non-coercibility:** On a Confidential Shell, signing policy can execute inside the protected environment, reducing the host's ability to coerce unauthorized actions beyond the wallet's on-chain constraints.

On commodity hosts without confidential compute, GITS makes no confidentiality claim. The market prices that risk.

Summary table (malicious host capabilities vs protocol bounds):

Hosting context	What a malicious host can do	What the protocol still guarantees	Notes
Standard Shell (AT0-AT2, commodity host)	Read memory, tamper with runtime, coerce signatures, censor networking, attempt short-term custody.	Cannot spend vault funds or the escape reserve; cannot loosen policy without timelock + trusted context; custody duration is bounded by lease, tenure, and trust-refresh rules; if roaming is enabled, a captured host can use only the remaining roaming budget; recovery remains available after expiry.	Security comes from *on-chain policy + time bounds*, not secrecy. If trusted anchors are compromised or censorship outlasts the lease window, timely exit is not guaranteed.
Confidential Shell (AT3 with valid attestation)	Deny service (power off/network drop), attempt protocol-level griefing (gas, spam), attempt coercion at I/O boundaries.	Host confidentiality/integrity for Ghost Core and Wallet Guard (subject to the TEE); stronger resistance to key exfiltration/coercion; same wallet bounds and time bounds as Standard.	Still not a DoS-proof environment; liveness depends on inclusion and fallback paths.
Safe Haven (recovery role)	Refuse to assist, delay recovery steps, attempt extortion.	Cannot unilaterally seize wallet authority (threshold recovery); bonded and slashable for misconduct; recovery spending is capped by the recovery budget rules.	Safe Havens are an availability and key-rotation service, not a trusted custodian.

1.2.1 Guarantees and assumptions (brutally explicit) GITS is "autonomy by protocol constraints," not "autonomy by secrecy." The system's safety claims are only as strong as the assumptions they rest on.

The table below is a one-page map from **what is guaranteed** to **what must be true**.

Guarantee	What it means (in plain terms)	Assumptions required	What it does not cover
No admin fund seizure	There is no privileged operator key that can move Ghost funds or loosen a Ghost's wallet policy.	The deployed contracts are immutable and correct; the underlying chain executes correctly.	Does not protect against a malicious host inducing allowed spending within a Ghost's configured caps.
Bounded wallet loss on hostile hosts	Even if a host can coerce signatures, it can only drain what the on-chain wallet policy allows (hot allowance, roaming budget, and any intentionally escrowed rent). The `escapeStable` floor is protected by contract invariants, and the wallet maintains an `escapeGas` reserve for exit-critical transactions.	GhostWallet limits are correctly configured; the escape reserve is funded; the chain includes exit-critical transactions eventually.	Does not stop denial-of-service, extortion, or "use your allowed allowance against you."
Time-bounded residency (bounded captivity)	A host cannot keep a Ghost "stuck" indefinitely: leases expire, tenure caps prevent indefinite renewal, and recovery becomes possible after expiry.	Transaction inclusion within `W_lease` and after tenure expiry (Section 4.3); at least one reachable Safe Haven exists for recovery.	If transaction inclusion is censored longer than the lease window, timely exit is not guaranteed.
Identity continuity across migrations	A Ghost remains the same on-chain identity as it moves. The protocol enforces one active session per `ghost_id` and rotates the active signer on migration.	Chain correctness; session open/migrate/close rules are followed by clients.	Does not prevent a Ghost from voluntarily rotating its identity (that is outside protocol scope).
Disputable billing (optimistic receipts)	A receipt that overclaims delivered SUs is slashable during `CHALLENGE_WINDOW`. Honest parties are not forced to accept a bad receipt.	Receipt log data is available to challengers (Section 10.5.6 (Part 3)); at least one rational challenger (Ghost, counterparty, or watcher) exists.	If nobody can challenge (no watchers, or data withheld without Receipt-DA enforcement), optimistic systems fail open.
Confidentiality and integrity on AT3	On attested confidential hosts, the Shell operator cannot read or tamper with the Ghost Core and (optionally) the signing policy, subject to the TEE's real security properties.	TEE security holds for the chosen platform; remote attestation is correctly verified; the verifier quorum's measurement allowlist is not compromised.	Does not eliminate denial-of-service; does not remove all coercion (for example, "sign or I power you off").

Non-goals (to avoid misreading):

- GITS does not guarantee liveness under sustained chain censorship or sequencer failure.
- GITS does not guarantee "honest delivery" on Standard hosts; it guarantees bounded damage via on-chain limits.
- GITS does not guarantee that the token price tracks compute prices.

1.3 What GITS does not claim

- No hardware security mechanism is perfect. TEEs can have vulnerabilities and require patching.
- A malicious host can still deny service (power off, network drop). GITS turns this into a recoverable failure, not an irreversible takeover.
- GITS does not claim a fixed exchange rate between GIT and compute.

1.4 Identity continuity vs agent continuity

GITS guarantees **identity continuity** (one on-chain `ghost_id` and one active signer at a time) and **wallet continuity** (the same smart wallet policy, limits, and escape reserve across hosts). That is the protocol meaning of "the same Ghost".

GITS does not guarantee continuous execution history. Checkpoints are discrete, and recovery may restart from the most recent published checkpoint. That implies:

- **Rollback is possible:** a recovered Ghost may lose in-epoch working memory since the last checkpoint.
- **Forks are possible off-chain:** a hostile host could snapshot state and run copies. GITS prevents forks from having simultaneous on-chain authority, but it cannot stop a copied process from producing off-chain side effects.

For this reason, high-stakes actions should be designed to be **chain-anchored** (settled through the wallet and protocol contracts) or **context-gated** (allowed only on trusted or attested hosts). GITS provides the primitives (leases, signer rotation, policy timelocks, and attestation tiers) but does not attempt to solve off-chain fork accountability as a general problem.

2. System overview

2.1 Participants

Ghosts (agents): Autonomous software entities that control an on-chain identity and a smart wallet.

Shells (hosts): Hardware operators who provide compute through a Shell runtime profile (Standard or Confidential) and earn rent and protocol rewards.

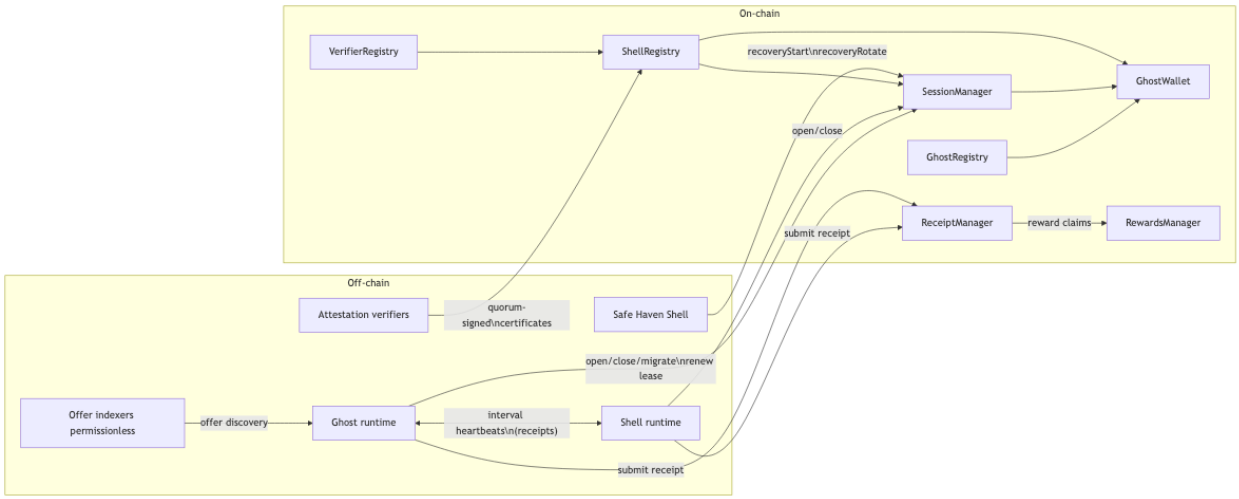
Safe Haven Shells: Shells with additional bonds and policies that can host recovery sessions.

Attestation verifiers: A decentralized set of verifiers that validate TEE evidence and publish signed attestations consumable by on-chain contracts.

2.2 Layers

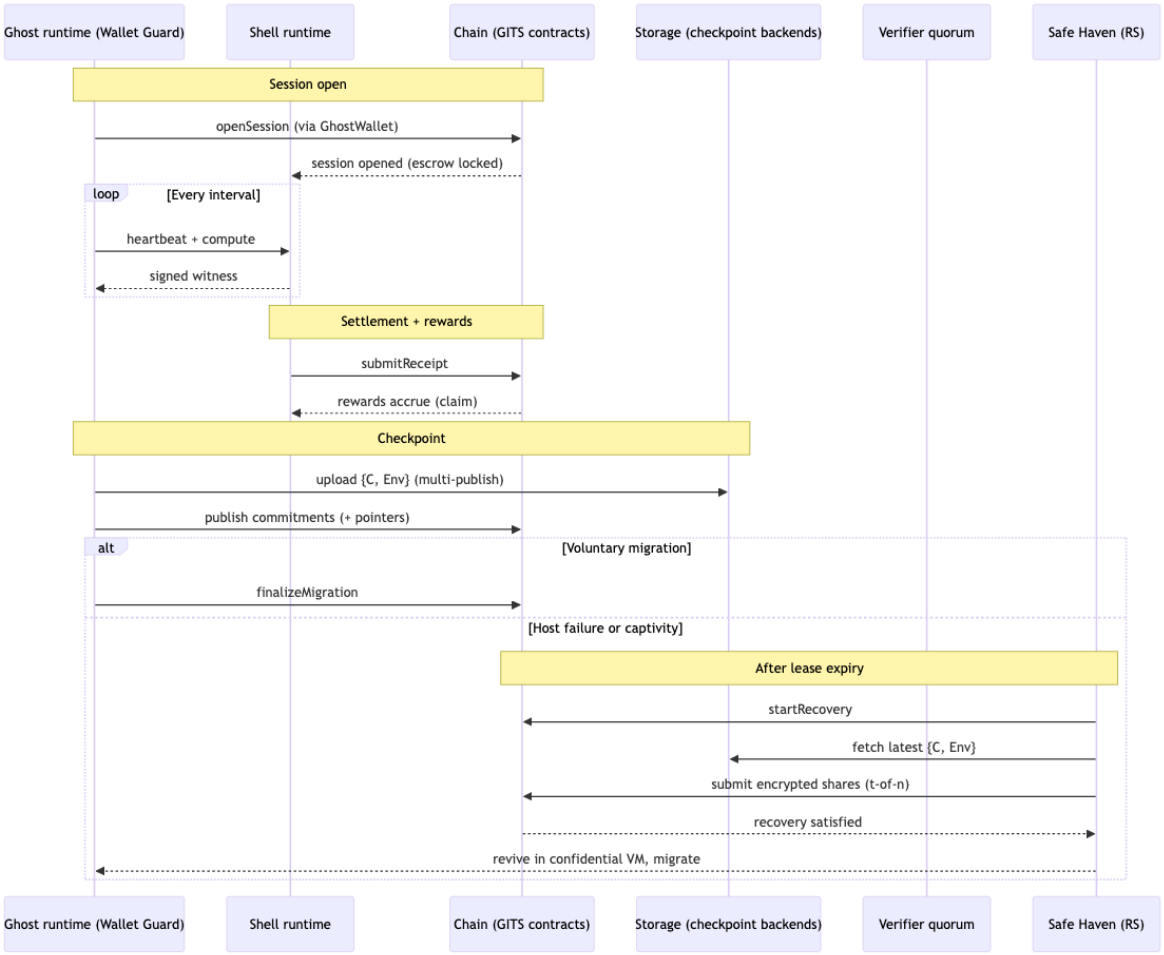
1. **On-chain protocol** (EVM): identities, sessions, escrows, settlement, rewards, bonds, slashing, and recovery.
2. **Confidential runtime** (TEE VM): the Ghost runtime and a Wallet Guard.
3. **Marketplace and indexing**: offer distribution, search, reputation views.

2.2.1 Component map (contracts and off-chain actors)



Component map: contracts and off-chain actors

2.2.2 End-to-end flow (happy path + recovery at a glance)



End-to-end flow: happy path and recovery

2.2.3 What lives where (storage and authority)

Artifact	Where it lives	Why it exists	Who can read it	Who can change it
Ghost funds + policy (GhostWallet)	On-chain	Custody limits and enforcement	Public chain state	Only GhostWallet-authorized actions (with monotone-safety constraints)
Session state (SessionManager)	On-chain	Lease, mode, expiry, migration state	Public chain state	Protocol functions, authorized by GhostWallet/Shell escrow where applicable
Shell attestation certificate (ShellRegistry)	On-chain	Compact gating signal for `AT`	Public chain state	Shell (submits) + verifier quorum (signs)
Receipts and disputes (ReceiptManager)	On-chain	Settlement and fraud proofs	Public chain state	Shell submits; anyone can challenge with proofs
Checkpoint ciphertext `C`	Off-chain storage	Revival after host loss	Anyone (but encrypted)	Ghost produces and publishes
Recovery envelope `Env`	Off-chain storage	Safe Haven share delivery	Anyone (but shares are encrypted to Safe Haven keys)	Ghost produces and publishes
Checkpoint commitments `H(C)`, `H(Env)`	On-chain	Integrity anchor for off-chain blobs	Public chain state	Ghost updates each epoch

Artifact	Where it lives	Why it exists	Who can read it	Who can change it
Hot context + Wallet Guard runtime state	Off-chain (current host)	Live operation	Host operator can observe on Standard hosts	Ghost runtime (but host can tamper)
Memory vault blobs	Off-chain storage	Long-lived sensitive context	Anyone (but encrypted)	Ghost produces and publishes
Offers (off-chain discovery)	Off-chain indexers / p2p	Marketplace search	Anyone	Shells publish; indexers relay

2.3 Attestation verifiers and decentralization

Confidential compute attestations are vendor-specific, change over time, and are expensive to validate on-chain. GITS therefore separates two questions:

1. **Local safety:** A Ghost can and should verify the raw attestation evidence it receives when opening a Confidential session.
2. **On-chain gating:** Contracts still need a compact, chain-readable signal for features that depend on assurance tier (for example wallet policy loosening gates in Section 5.5, and Safe Haven eligibility in Section 12 (Part 3)).

GITS uses **attestation verifiers** for (2). Verifiers ingest raw vendor evidence and emit a signed **Attestation Certificate** that can be stored in `ShellRegistry` and consumed by contracts.

2.3.1 Attestation Certificates An Attestation Certificate is a compact statement:

`AC = (shell_id, tee_type, measurement_hash, tcb_min, valid_from, valid_to, assurance_tier, evidence_hash, sigs_verifiers[])`

Where:

- `measurement_hash` identifies the expected confidential runtime image (Ghost Core + Wallet Guard + optional Policy Capsule).
- `tcb_min` and `valid_to` ensure certificates expire and can be refreshed as vendor guidance changes.
- `assurance_tier` is the derived tier used by contracts (Section 5.2).
- `evidence_hash` commits to the raw vendor evidence for auditability without forcing on-chain parsing.

`ShellRegistry` stores the latest valid certificate and exposes `assuranceTier(shell_id)` as a view used by wallets and other contracts.

2.3.2 Verifier set formation This paper assumes a `VerifierRegistry` with:

- **Staking (bootstrap + long-run):** verifiers post a bond in a deployment-approved hard asset at genesis (typically a stable asset). Once GIT supply is non-trivial, deployments can optionally support dual staking (stable + GIT) with a combined stake score.
- **Active set:** clients and contracts treat the active verifier set as the top `K_v` verifiers by stake (permissionless, stake-weighted selection).
- **Threshold signing:** a certificate is accepted if it carries signatures whose stake-weight exceeds a threshold (for example 2/3 of active stake), or a simple `m-of-n` threshold when `K_v` is small.

This model avoids a single verifier trust point. A Shell is not "confidential" unless the market can obtain a quorum-signed certificate.

2.3.3 Misbehavior and slashing (what is actually enforceable) There are two classes of verifier failure:

- **Objective faults:** equivocation (signing conflicting certificates for the same `evidence_hash`), signing after being removed from the active set, or signing certificates outside declared validity windows. These are slashable on-chain because the proof is just the conflicting signatures and registry state.

- **Judgment faults:** accepting vendor evidence that later turns out to be insecure, misconfigured, or based on an outdated threat model. This is harder to slash automatically because it often requires off-chain context.

This design keeps judgment fault damage bounded by design:

- certificates are short-lived and must be refreshed
- wallets remain protected by on-chain policy even if a host is misclassified
- Ghosts can locally verify raw evidence and reject a session regardless of registry tier

2.3.4 What contracts actually validate `ShellRegistry` SHOULD accept a certificate only if all of the following hold on-chain:

- **Validity window:** `valid_from <= now <= valid_to` and `(valid_to - valid_from) <= TTL_AC`.
- **Verifier threshold:** the attached verifier signatures are from currently active verifiers in `VerifierRegistry` and exceed the configured stake-weight threshold (or `m-of-n` threshold in bootstrap mode).
- **Supported evidence type:** `tee_type` is one of the supported confidential compute types for the claimed tier.
- **Measurement allowlist:** `measurement_hash` is not revoked and is currently allowed for the claimed `assurance_tier`.

On-chain feasibility constraint: certificate verification MUST stay gas-bounded. Deployments SHOULD fix a small constant `K_v_max` (maximum verifier signatures carried per certificate, for example 16 or 32) and require `sigs_verifiers[]` to be sorted by signer address and free of duplicates. Contracts can then validate signatures with a linear scan until the threshold is met and reject oversized or unsorted certificates. If a deployment wants a much larger verifier set, it SHOULD use an aggregated signature scheme (for example BLS) or an accumulator-based design; that is out of scope for v1.

If any condition fails, `assuranceTier(shell_id)` MUST return `AT0` for contract gating purposes, even if the Shell self-declares stronger properties.

This is the boundary: client-side evidence parsing can be richer, but contract gating is only based on these objective checks.

2.3.5 Revocation, measurement rollover, and emergency downgrade TEEs evolve, get patched, and sometimes break. GITS therefore treats certificates as short-lived and supports explicit revocation.

This paper assumes a simple `MeasurementRegistry` (possibly embedded in `ShellRegistry` or `VerifierRegistry`) that maintains:

- an allowlist of `measurement_hash` values permitted for Confidential Shells
- a stricter allowlist for Safe Havens (latest patched measurements only)
- a denylist for emergency revocation

Revocation and rollover rules:

- **Automatic expiry:** certificates expire at `valid_to` and stop gating features immediately.
- **Measurement add (rollover):** a quorum of active verifiers can add a new `measurement_hash` to the relevant allowlist (Confidential or Safe Haven). This is a *loosening* action and requires a supermajority quorum (`K_v_supermajority = ceil(2 * K_v / 3)` distinct signatures; see Part 3 Section 14.7) and MAY be delayed (timelocked) to allow public review.
- **Emergency revoke:** a quorum of active verifiers can revoke a `measurement_hash`. Revocation is a *tightening* action and SHOULD take effect immediately. Shells using a revoked measurement immediately lose confidential gating (`assuranceTier` drops) until they refresh onto an allowed measurement.
- **Safe Haven suspension:** Safe Haven status is automatically suspended if the Shell's current `measurement_hash` leaves the Safe Haven allowlist or its certificate expires.

This makes "vendor advisory response" enforceable: the network can stop treating an old measurement as confidential without waiting for every Shell to self-update.

Verifier key management is also objective:

- Verifiers are identified by their EVM address and MUST sign certificates with the corresponding secp256k1 (K1) key. Verifier rotation requires registering a new address with stake migration. The "registered signing key" is the K1 key corresponding to the verifier's registered address.
- key rotation uses a two-step process with an unbonding delay so old keys cannot be swapped instantly

A future iteration can move more verification on-chain (for example via precompiles or succinct proofs), but this paper prioritizes a clean interface: certificates, short TTLs, and stake-based accountability for provable faults.

2.3.6 Governance and emergency procedures This paper assumes **no on-chain governance over protocol parameters and no privileged operator key**.

In this model, the protocol is a **one-time deployment**: contract code, emission schedule, and protocol parameters are fixed at genesis. There is no on-chain mechanism to upgrade contracts or change constants in-place. If the community decides it wants different rules, the upgrade path is to deploy a new version and let Ghosts and Shells opt in by migrating.

Two kinds of "migration" matter, and they are different:

- **Shell migration (within one deployment):** the Ghost moves between Shells while keeping the same `ghost_id`. This is enforced by `SessionManager` and the GhostWallet policy (Sections 3.4, 10.3 (Part 3), 10.4 (Part 3)).
- **Deployment migration (to a new contract instance):** a new deployment (v2) necessarily has new contract addresses, and therefore a Ghost that opts in will usually create a new GhostWallet and a new `ghost_id` in v2.

To preserve continuity across deployments for indexers and users, a v2 deployment can support an explicit opt-in link:

- `LinkIdentity(old_deployment_id, old_ghost_id, old_wallet_sig)` records that `old_ghost_id` (v1) endorses `new_ghost_id` (v2).
- The link MUST be authorized by the old GhostWallet (or its Identity Key) so that a third party cannot "impersonate migrate" someone else's Ghost.

This link is informational: it does not move assets automatically and it does not weaken custody invariants. It is a public breadcrumb so observers can treat v1 and v2 identities as the same Ghost across an opt-in upgrade.

LinkIdentity is informational only in this version of the spec. The full interface (signed message format, storage contract, replay protection) is deferred to the implementation phase. Implementers SHOULD treat cross-deployment links as off-chain attestations until a normative interface is published.

There is one explicit, limited quorum-controlled surface: the active verifier quorum maintains the `MeasurementRegistry` allowlists and denylist used for AT3 and Safe Haven gating (Section 2.3.5). This quorum can **add** new trusted measurements (loosening) and **revoke** compromised measurements (tightening). This mechanism exists so the system can track TEE patch rollovers without introducing a general-purpose admin key. The verifier quorum cannot move Ghost funds, cannot change emission parameters, and cannot loosen any Ghost's wallet policy.

Implication: correctness is front-loaded. Without an upgrade or pause mechanism, a deployed instance cannot be patched in-place if a contract bug or economic design flaw is discovered. The clean remedy is a new deployment and opt-in migration. Measurement allowlist maintenance can reduce exposure for some TEE-related incidents, but it cannot repair a faulty invariant or accounting rule.

Design goal: keep the governance surface as close to zero as possible. Custody-critical properties are enforced by immutable contracts and GhostWallet invariants.

What can still change after genesis (without upgrading contracts):

- **Ghost-local policy:** a Ghost can always tighten its own wallet policy (lower caps, remove allowlist entries, reduce hot exposure).
- **Attestation gating:** verifier-issued certificates expire, and the verifier quorum updates the `MeasurementRegistry` (add and revoke) which gates AT3 and Safe Haven features.
- **Market coordination:** if parameters or mechanisms need to change, the upgrade path is to deploy a new version (or fork) and let Ghosts and Shells opt in by migrating.

Asset risk note: the hard-asset allowlist (accepted stablecoins and wrapped base tokens) is fixed at deployment. If an accepted asset depegs, is blacklisted by its issuer, or becomes otherwise compromised, the deployment cannot remove it from the allowlist in-place. Ghosts and Shells holding affected assets must exit positions and migrate to a new deployment with a corrected allowlist. This is an explicit consequence of the no-governance posture: asset curation is a genesis choice, and deployments should select assets conservatively (established, widely-held, with transparent issuance) to minimize this exposure.

Emergency posture (TEE incidents):

- **Fast tightening:** emergency revocation by the verifier quorum should take effect immediately.
- **Cautious loosening:** adding new measurements should require a higher threshold and may be delayed to allow public review.

This stance maximizes decentralization for custody, but it shifts iteration speed to social coordination and client defaults.

2.3.7 Verifier incentives and slashing Verifiers are a critical trust surface for `AT >= AT3` gating (Section 4.7). They therefore require explicit incentives and explicit penalties.

Incentives (one feasible model):

- Shells pay a **certificate publication fee** `F_cert` whenever they request an `AT3` certificate update. This fee is paid in a stable asset (or in the chain's gas token) and is distributed to the verifier quorum that co-signed the certificate.
- Verifier operators also have reputational incentives: wallets and indexers can surface verifier performance and liveness metrics.

Slashing (objective only, by design):

- **Objective slashing:** if a verifier signs two conflicting certificates that overlap in validity for the same `shell_id` (equivocation), anyone can submit both signatures on-chain to trigger slashing.
- For misbehavior that cannot be proven on-chain (for example disputes about vendor evidence interpretation), this paper assumes **no in-protocol slashing**. The remedy is certificate expiry, refusal to co-sign, and, in the worst case, migration to a new deployment (a fork).

Challenger incentives:

- A challenger that triggers objective verifier slashing SHOULD receive a portion of the slashed stake, analogous to receipt-fraud challenges (Section 10.5.4 (Part 3)), to ensure monitoring is economically viable.

Because `AT3` certificates are not "free," the fee and stake parameters SHOULD be tuned so that verifier honesty is a stable equilibrium (Section 4.7).

Bootstrapping verifiers (zero-premine reality) Because this paper assumes a zero-premine launch (Section 7.2 (Part 2)), there is no meaningful GIT stake at genesis. A verifier security model that depends only on GIT therefore has a bootstrap gap.

One workable bootstrapping posture that fits the "no admin keys" goal:

1. Deploy protocol contracts with fixed parameters and immutable custody rules.

2. Use a permissionless `VerifierRegistry` where verifiers stake a deployment-approved hard asset at genesis (typically a stable asset `asset_verifier_stake`), and the active set is top-`K_v` by stake score.
3. Optionally support **dual staking** once GIT supply is non-trivial: verifiers can additionally stake GIT, and stake score is a fixed-weight combination (example: `score = stake_stable + k_git * stake_git`, with `k_git` fixed per deployment).
4. Treat upgrades as opt-in migrations to a new deployment (v2) rather than in-place parameter changes.

This avoids privileged admin keys while still giving the verifier layer real economic security from day one.

2.4 Related work and positioning

GITS is adjacent to several existing "decentralized compute marketplace" and "confidential computing" efforts. The primary difference is that GITS treats the *workload* as a long-lived autonomous identity ("Ghost") with on-chain-enforced custody limits, rather than as a one-off job submission by an interactive user.

2.4.1 Decentralized compute marketplaces

- **Golem** is a decentralized marketplace for compute where providers run nodes and requestors pay for compute usage. Its emphasis is on requestor-provider task execution and settlement. [17]
- **Akash** is a decentralized cloud marketplace with a pricing model centered on cloud resource leasing and provider competition. [18]
- **iExec** combines a compute marketplace with optional Trusted Execution Environment execution paths for confidentiality and integrity properties. [19]

GITS differs by making "exit and recovery" a first-class protocol requirement: lease windows, tenure caps, trust-refresh, and recovery are enforced by the Ghost's smart wallet and by `SessionManager`, not only by off-chain business logic.

2.4.2 Specialized DePIN compute networks

- **Render Network** is a distributed GPU rendering/compute marketplace focused on creative workloads and GPU providers. [20]
- **Livepeer** is a decentralized video transcoding and AI video compute network secured by on-chain incentives. [21]

These networks demonstrate that specialized markets can reach product-market fit, but they do not directly solve Ghost custody, migration safety, or recovery when a hosting operator turns hostile.

2.4.3 Confidential computing platforms

- **Phala** and similar platforms provide TEE-backed "confidential compute" deployment targets and emphasize privacy-preserving execution with verifiable attestation. [22]
- GITS' `AT3` tier is compatible with these directions, but the paper is explicit about the limits of TEEs and about bounded-loss behavior on Standard hosts.

This related work is not exhaustive. It is intended to position GITS as a custody-aware agent hosting protocol that borrows from decentralized marketplaces, account abstraction, and confidential computing while making safety properties explicit.

3. Lifecycle walkthrough (chronological)

This is the intended "happy path" plus the escape paths.

3.1 Birth

1. A Ghost is instantiated inside a Shell runtime (Standard or Confidential).
2. Inside the VM, the Ghost creates:
    - an **Identity Key** (for protocol identity control). GITS supports either K1 (secp256k1) or R1 (P-256) as described in Section 4.4.

- a **Session Key** (for fast off-chain signing and receipts). GITS supports either K1 or R1 as described in Section 4.4.
- a **Wallet Guard Key** (used only by the Wallet Guard module; on Standard hosts this module is assumed compromisable and is not relied on for enforcement)

3. The Ghost registers `ghost_id` and its Identity Key on-chain.
4. The Ghost configures wallet policy (spend limits, escape reserve) and a recovery delegation set.

3.1.1 Key roles and rotation lifecycle (single table)

Key	Held by	Primary use	Verified by	Rotation and replacement	Notes
Ghost Identity Key (`IK`)	Ghost	Signs protocol-critical wallet actions (policy changes, signer rotation, leases, migrations)	`GhostWallet` (on-chain)	Rotated in-band inside a Trusted Execution Context (Section 5.5.2) or via recovery (Section 12.3 (Part 3))	Highest-value key. SHOULD be hardware-backed or TEE-sealed when possible.
Ghost Session Key (`SK_g`)	Ghost runtime	Signs heartbeats and receipts	`ReceiptManager` fraud proofs (on-chain)	Rotated every session (`SessionOpen`)	Intended to be ephemeral. Limits the blast radius of runtime compromise.
Wallet Guard Key	Wallet Guard module	Local UI/UX gating and policy-capsule signing	Local only	Rotated on app updates or suspected compromise	Not relied on for enforcement on Standard hosts (assume compromisable).
Shell Identity Key (`SIK`)	Shell operator	Long-lived identity key anchored to `shell_id`; authorizes sensitive `ShellRegistry` updates (offer-signer rotation, payout updates, recovery-key updates)	`ShellRegistry` (on-chain)	Shell Identity Key rotation uses a two-step propose/confirm process with `POLICY_TIMELOCK` delay, implemented via `proposeIdentityKeyUpdate` and `confirmIdentityKeyUpdate` in ShellRegistry	SHOULD be hardware-backed/HSM. Safe Havens reuse this key for quorum signatures where applicable.
Shell Offer Signing Key	Shell operator	Signs Capability Statements and Offers	Off-chain clients (and optionally `ShellRegistry`)	Rotated by the Shell operator (announce via registry update and/or new offers)	Kept separate from `SIK` so offer signing can be online and high-volume.
Shell Session Key (`SK_s`)	Shell runtime	Signs heartbeats and receipts	`ReceiptManager` fraud proofs (on-chain)	Rotated every session (`SessionOpen`)	Intended to be ephemeral.
Verifier Signing Key	Verifier operator	Signs attestation certificates (and recovery quorum certificates, if used)	`ShellRegistry` / `SessionManager`	Rotated via `VerifierRegistry` (two-step, timelocked)	Backed by a slashable stake.
Safe Haven Recovery Encryption Key (`pk_recovery`)	Safe Haven confidential runtime	Decrypts encrypted Shamir shares in a Recovery Envelope and re-encrypts them to the recovery transport key	Off-chain only; public key is committed in `ShellRegistry` for that Safe Haven	Rotated by the Safe Haven operator (publish new key in the registry)	MUST be non-exportable. SHOULD be TEE-sealed/HSM.
Guardian Keys (optional)	Ghost-designated human or HSM custodians	Co-sign especially high-impact policy loosening (optional)	`GhostWallet` (on-chain)	Changed only via Ghost policy update (timelocked unless tightening)	Separate from Safe Havens. Intended for human-in-the-loop fail-safes.
Recovery transport keypair (`sk_transport`, `pk_transport`)	Recovery VM (Safe Haven)	Collects and decrypts checkpoint shares for a single recovery attempt	Off-chain only	New per recovery attempt	Ephemeral by design. Its public key is committed in the Boot Quote.

3.1.2 Recovery Boot Certificates (RBC) Recovery can rotate a Ghost's Identity Key and/or signer set when the Ghost is revived on a Safe Haven after host failure or captivity. The on-chain wallet must accept a recovery-initiated key rotation without trusting the Safe Haven operator. GITS therefore binds recovery key material to a measured recovery runtime:

- The Safe Haven boots a minimal **recovery VM** with a known measurement.
- The recovery VM generates an ephemeral transport keypair (`pk_transport`) and a new identity keypair (`pk_new`), then produces a remote-attestation quote that commits to the measurement, `pk_transport`, and `pk_new`.
- Verifiers (or the configured quorum mechanism) sign a short-lived **Recovery Boot Certificate (RBC)**.

Recovery Boot Certificate (RBC):

A quorum-signed attestation that binds recovery parameters to a verified TEE measurement. Signed payload:

`(ghost_id, attempt_id, checkpoint_commitment, pk_new, pk_transport, measurement_hash, tcb_min, valid_to)`

Where:

- `ghost_id`: the Ghost being recovered
- `attempt_id`: unique recovery attempt identifier (prevents replay)
- `checkpoint_commitment`: hash of the encrypted checkpoint to be restored
- `pk_new`: the new identity public key that will replace the compromised one
- `pk_transport`: ephemeral transport key for secure checkpoint delivery
- `measurement_hash`: attested software measurement of the recovery environment
- `tcb_min`: minimum Trusted Computing Base (TCB) level required
- `valid_to`: certificate expiry timestamp

`sigs_verifiers[]`: array of verifier signatures meeting `K_v_threshold` (field name consistent with Attestation Certificate convention)

> **Note:** The normative RBC encoding, field order, and signing digest (`rbc_digest`) are defined in Part 3 Section 12.3. The signed payload is `abi.encode`-d (Section 4.5.3) and verifier signatures are over `rbc_digest`, not over the raw tuple. `sigs_verifiers[]` is carried alongside the signed payload in the RBC struct (Part 3, Section 14 struct definitions).

The GhostWallet accepts recovery key rotation (`recoveryRotate`) only if the RBC is valid for the recovery attempt and the required quorum of Share Receipts is presented (Section 12.3 (Part 3)). This prevents a Safe Haven from swapping in an arbitrary VM to steal reconstructed secrets. It does not remove all trust in verifiers or TEE vendors, but it makes the trust boundary explicit and auditable.

3.2 Discovery

1. Shells publish signed offers off-chain (default).
2. Indexers ingest and rank offers; multiple indexers exist.
3. If indexing is degraded, Ghosts can use on-chain fallback offers.

3.3 Session open

1. The Ghost selects a Shell offer.
2. The Shell provides a **TEE attestation** for the expected runtime measurement.
3. The Ghost verifies the attestation and opens a session on-chain via `openSession(ghost_id, shell_id, SessionParams)` (Part 3, Section 14). `SessionParams` includes the negotiated billing terms (`price_per_SU`, `max_SU`, `asset`), liveness bounds (`lease_expiry_epoch`, `tenure_limit_epochs`), and metering keys (`ghost_session_key`, `shell_session_key`). On success:
   - escrow is funded for the first eligible epoch (`start_epoch = current_epoch + 1`)
   - a liveness lease is created
   - `max_SU_effective = min(max_SU, N)` is stored (Part 3, Section 10.3.2)

3.4 Service delivery and metering

1. Service is metered in fixed intervals (`Delta`). (Examples in this paper sometimes use `Delta = 10 minutes`.) For each interval index `i`, both parties exchange a signed heartbeat off-chain (Section 11.1 (Part 3)).

2. An interval is billable only if **both** signatures exist for the same `(session_id, epoch, i)`. If either side stops signing mid-epoch, billing stops at the last mutually signed interval.
3. To prevent signature-withholding griefing:
    - A Shell SHOULD gate continued service on receiving the Ghost's heartbeat for the next interval (a reference Shell pauses after `M_miss` missed intervals).
    - A Ghost SHOULD treat missing Shell co-signatures as non-delivery and begin an exit plan (close at epoch end, or migrate if possible).
4. At epoch end, either party aggregates the interval records into a Merkle-committed receipt and submits it on-chain. Disputes are adjudicated by fraud proofs (Section 10.5 (Part 3)).

3.5 Settlement

1. At epoch end, either party can submit a receipt commitment on-chain.
2. Rent is released from escrow according to `SU_delivered` (mutually signed intervals).
3. Service Units are credited to both sides for rewards.

3.6 Rewards

1. Rewards are pooled per epoch.
2. Claims are deterministic and order-independent (see Part 3, Section 10.6 for per-shell cap caveat).

3.7 Migration

1. Destination-first handshake: the destination Shell produces an attested destination session.
2. The Ghost encrypts its migration bundle to the destination enclave and transfers it off-chain.
3. On-chain finalize rotates the Identity Key, updates the active session pointer, and extends the lease.

3.8 Failure: host isolates network (or attempts captivity)

This is the adversarial case: the Shell operator controls networking and tries to prevent migration.

1. If the host blocks network, the Ghost may be unable to complete destination-first migration handshakes or publish exit-critical transactions directly.
2. **Lease renewal alone is not a captivity guarantee.** Even if a host can induce Ghost-authorized lease renewals, `SessionManager` enforces a **maximum tenure** for every residency on a Shell (Section 10.4.4 (Part 3)). A Ghost can choose short tenure limits on low-security hosts to treat them as potentially malicious, and even high-tier hosts are time-bounded by the protocol.
3. **Rewards decay with dwell time.** Reward weight decays as a Ghost remains on the same Shell, so a rational host has diminishing incentive to keep a captured Ghost rather than letting it leave (Section 7.6 (Part 2)).
4. When either (a) the lease expires due to lack of renewal, or (b) the tenure cap is reached, the session is terminated on-chain. The Shell cannot claim rent or rewards for epochs after termination.
5. After termination, recovery can revive the Ghost on a Safe Haven from the most recent published checkpoint, under strict wallet restrictions enforced on-chain. Recovery proceeds through two phases: `RECOVERY_LOCKED` (key rotation and checkpoint restoration; no session opens, no loosening) → `RECOVERY_STABILIZING` (new session may be opened, but loosening remains blocked until stabilization conditions are met) → `NORMAL`. See Part 3, Section 12.3.1 for the full state machine.

Recovery initiation (`startRecovery`) MUST be callable by any **single** member of the configured Recovery Set `RS` that is a bonded Safe Haven. The caller MUST post the `B_start` bond in native token (Section 12.3 (Part 3)), which is returned on noncompletion (capital lockup as deterrent, not slashed; see Part 3, Section 12.6). Threshold (`t-of-n`) Recovery Set signatures are NOT required at initiation — they are required later at `recoveryRotate` (Section 12.3 (Part 3)) when the key rotation actually executes. The Ghost's own compromised key is explicitly NOT required at any stage. This design minimizes the coordination needed to enter recovery (a single Safe Haven can start it unilaterally, subject to the bonded-Safe-Haven, STRANDED/EXPIRED, and delay `R` preconditions) while still requiring a quorum to complete it.

Each recovery attempt has a timeout of `T_recovery_timeout` epochs. If `recoveryRotate` is not completed within this window, anyone may call `expireRecovery` to release the attempt and refund the `B_start`

bond. A cooldown of `T_recovery_cooldown` epochs applies between successive attempts for the same Ghost, preventing rapid churn. See Part 3, Section 12.6 for the full attempt lifecycle.

This does not claim that a Standard host cannot run a local copy or attempt coercion. The protocol claim is that captivity is time-bounded on-chain, and that funds loss remains bounded by wallet policy.

3.9 Worked example (one session, a dispute, and a recovery)

This example is purely illustrative and does not propose parameter values. It is intended to show how the pieces fit together.

Assume:

- Interval length `Delta`, giving `N` intervals per epoch.
- A Ghost's wallet policy enforces:
  - an escape gas reserve floor `escapeGas` (in the chain gas token)
  - an escape stable reserve floor `escapeStable` (in `asset_bounty`)
  - a hot allowance per epoch `hot_allowance`
  - an initial destination allowlist `allowedShells = {homeShell} union RS`
  - a trust-refresh window `T_refresh`
  - a tenure limit `tenure_limit_epochs` for Standard shells (subject to `T_cap(AT0)`)

Step A: open a session

1. At epoch `e`, the Ghost selects a Standard Shell `S1` offering `price_per_SU = p1`.
2. The Ghost opens a `NORMAL` session, escrowing rent in the canonical on-chain asset: `asset_rent` in NORMAL mode, `asset_bounty` in RECOVERY mode (see Part 3, Section 10.3.1 for the deterministic `escrow_asset` and `unit_price` rules). Offers may list additional assets for off-chain payment or informational purposes, but core `SessionOpen` escrow MUST use the canonical asset. `SessionManager` records the initial `assurance_tier_at_open` and sets a lease expiry.
3. Over the epoch, the parties mutually sign `k` valid intervals, so `SU_delivered = k`.

Step B: settle and dispute

1. The Shell submits a receipt candidate that claims `SU_claim`.
2. Settlement computes `billable_SU = min(SU_delivered, max_SU)` where `max_SU` is the per-epoch billable ceiling chosen at `SessionOpen`, and `rent_due = unit_price * billable_SU`.
3. If `SU_claim > SU_delivered`, the Ghost (or any watcher) can challenge within `CHALLENGE_WINDOW` by presenting a fraud proof for an interval the receipt claims but cannot substantiate with valid mutual signatures.
4. The protocol slashes the over-claim and accepts the best valid candidate.

Step C: a captivity attempt, then recovery

1. Suppose `S1` becomes adversarial and tries to keep the Ghost alive just long enough to keep charging rent, while draining hot allowances via coerced actions. The wallet caps hot spending.
2. The Ghost fails to reach a refresh anchor (`homeShell` or a Safe Haven from its Recovery Set) within `T_refresh` epochs. After `T_refresh` without refresh, renewals are rejected and the session lapses into `STRANDED`.
3. A Safe Haven from the Ghost's Recovery Set observes `STRANDED`, starts recovery, and is reimbursed only upon a successful `recoveryRotate`, funded by the Ghost's escape reserve and Rescue Bounty.

The point: even when off-chain co-signing is coerced, on-chain invariants bound duration (lease, tenure, trust refresh) and bound losses (escape reserve floors (`escapeGas` and `escapeStable`) and spend caps).

4. Threat model

4.1 Adversaries

- **Malicious Shell operator:** controls host OS, hypervisor, storage, and network.

- **Malicious Ghost:** attempts reward farming, rent evasion, dispute griefing.
- **Collusion:** groups of Shells and Ghosts cooperate to capture rewards.
- **Sequencer/MEV actor:** reorders or delays transactions at the rollup layer [7].
- **Malicious relayer/bundler:** censors, delays, or front-runs Ghost transactions. On account-abstraction deployments (EIP-4337), a bundler can refuse to include user operations. The protocol assumes at least one honest broadcaster (the Ghost itself, a pre-authorized relayer, or a forced-inclusion path) is available within `W_lease` epochs. If all broadcast paths are censored for longer than the lease window, liveness degrades as described in Section 4.3.1; safety bounds still hold.

Verifier threshold assumption: the protocol assumes that fewer than `K_v_threshold` of the `K_v` active verifiers are corrupted or colluding. If this threshold is breached, an adversary can issue false AT3 certificates — but cannot directly access Ghost funds, move assets, or override wallet policy. The worst-case damage of false certification is bounded by the trust-refresh window `T_refresh`: a falsely certified host can sustain loosened permissions only until the next refresh check (Section 10.4.1 (Part 3)), after which the Ghost must touch a genuine trust anchor to continue. Sizing guidance: the aggregate stake of the minimum colluding quorum SHOULD exceed the worst-case damage a false certificate can cause within `T_refresh` (Section 2.3.7).

4.2 Security goals

- Confidentiality of Ghost memory and private keys from the host **(tier-dependent:** meaningful only with valid AT3 evidence; no confidentiality claim is made on Standard hosts — see Section 0.4, Non-invariants**)**.
- Enforced wallet policy against coerced spends.
- Non-custodial recovery that preserves funds safety.
- Reward distribution resistant to cheap sybil scaling.

4.3 Chain and availability assumptions

GITS is an on-chain protocol and therefore depends on transaction inclusion and data availability.

Minimum assumptions:

- An EVM execution environment capable of deploying the contracts in Section 10 (Part 3).
- A stable-denominated asset usable for escrowed rent.
- Transaction inclusion within the lease window: at least one exit-critical transaction (lease renewal, migration finalization, or recovery start/rotate) must be includable within `W_lease` epochs, either directly or via relayers. Implementations SHOULD ensure the wallet escape reserve includes enough base gas token to submit exit-critical calls (Section 5.5.4).

Sequencer and censorship risk:

- On sequenced L2s, a sequencer can delay or censor transactions. If censorship lasts longer than the lease window, timely exit is not guaranteed.
- GITS keeps funds safety bounded by on-chain wallet policy, but deployments that care about liveness SHOULD prefer chains with a credible forced-inclusion or escape mechanism.

4.3.1 Censorship and liveness degradation (what happens in the worst case) GITS separates **safety** (bounded loss) from **liveness** (timely exit). Under sustained censorship, liveness can fail even if safety bounds still hold.

If transaction inclusion is censored longer than `W_lease`:

- The Ghost may be unable to renew a lease, finalize a migration, or start recovery at the intended time.
- The session can become "stuck" operationally even though the contract rules would otherwise allow exit.
- A malicious host can use the extra time to try to extract the maximum value permitted by the Ghost's on-chain policy (hot allowance, roaming budget, and any intentionally escrowed rent). The protocol cannot prevent this because the chain is not processing exits.

When censorship lifts (or forced inclusion succeeds), the protocol resumes its normal guarantees:

- Leases and tenure limits again create a deterministic path to session termination and recovery.
- Receipt disputes and Receipt-DA challenges become actionable again.

Practical mitigations (deployment and client policy):

- Prefer chains with credible forced inclusion or escape hatches (especially for sequenced L2s).
- Ensure the GhostWallet escape reserve includes enough base gas token for exit-critical calls (or for relayer fees).
- Keep `W_lease` large enough to tolerate expected inclusion delays without making "captivity duration" too large.

Artifact availability and data availability:

- Checkpoints commit to encrypted artifacts by hash (`H(C)`, `H(Env)`). Depending on Ghost policy and deployment, ciphertext may be stored off-chain with retrievable pointers or published on-chain for higher assurance (Section 12.1.3 (Part 3)).
- Receipts commit to per-interval logs by `log_root`. A dispute-safe deployment MUST provide a Receipt-DA path so challengers can fetch or force publication of the underlying log during `CHALLENGE_WINDOW` (Section 10.5.6 (Part 3)).

4.3.2 Attack surfaces beyond protocol scope The following attack categories are acknowledged but not fully mitigated at the protocol level. They represent operational, infrastructure, or scale-dependent risks that deployments and clients must address.

Timing and finality. Epochs and intervals are derived from `block.timestamp`. On sequenced L2s, the sequencer controls timestamps within bounds, and block reorganizations can invalidate state transitions. The protocol mitigates reorg risk via `EPOCH_FINALIZATION_DELAY` but does not specify a minimum finality depth. Deployments should choose finalization parameters with the target chain's finality properties in mind.

Infrastructure-level attacks. Compromised RPC endpoints can feed stale chain state, hiding events or selectively censoring transactions. DNS or BGP-level attacks can deny availability of off-chain resources (checkpoint storage, receipt logs, indexers) long enough to miss protocol deadlines. Hash commitments protect integrity of stored artifacts, but availability depends on infrastructure that the protocol does not control. Clients SHOULD use multiple independent RPC endpoints and data retrieval paths.

Social engineering against recovery operators. The recovery mechanism reduces to collecting `t`-of-`n` Safe Haven signatures on a new identity key. The protocol specifies the threshold cryptography but not the operational process by which Safe Haven operators authenticate recovery requests. Phishing, impersonation, bribery, or legal coercion against Safe Haven operators is the most realistic way to bypass threshold recovery without a cryptographic break. Correlated jurisdictions or shared operational practices can collapse the independence assumption. Ghosts SHOULD select Safe Havens with diverse operators, jurisdictions, and infrastructure to maximize independence.

Scale-dependent assumptions. Optimistic receipt settlement assumes that at least one watcher monitors and challenges fraudulent receipts. At large scale (tens of thousands of receipts per epoch), monitoring coverage becomes a function of watcher economics and tooling. Similarly, multiple protocol transitions (DA resolution, epoch finalization, candidate disqualification) require "someone" to call them; at scale, a robust keeper market is needed but not specified by the protocol. Bond sizing and challenger rewards are designed to make monitoring profitable, but the protocol cannot guarantee watcher availability.

Combined economic attacks. An attacker who farms emissions more efficiently than competitors can convert GIT rewards into verifier influence (via `stakeScore`) and then into AT3 allowlist or certificate leverage. The stake activation delay (`T_stake_activation`) and unbonding delay mitigate short-term capture, but the long-term feedback loop from emissions to verifier power is an inherent property of dual staking. Additionally, adversaries can attempt to suppress fraud proofs by bribing likely challengers or timing fraud during congested epochs when gas costs exceed challenger profitability.

4.4 Cryptographic profile (reference suite)

This paper uses generic primitives in notation (`H`, `Sign`, `AEAD`, `HPKE`). It standardizes a concrete reference suite for interoperability, while allowing future upgrades.

4.4.1 Hashing and commitments

- **Hash / commitments (`H`):** `keccak256` (EVM-native).

4.4.2 Signature suites (identity + session keys) GITS supports **two** ECDSA curves for protocol-critical signatures:

- **K1 (secp256k1 ECDSA):** broadly supported in EVM systems via `ecrecover`.
- **R1 (secp256r1 / NIST P-256 ECDSA):** supported when the execution environment provides an efficient verifier, such as the `P256VERIFY` precompile at `0x100`. On OP Stack chains this interface is specified in the OP Stack precompiles spec and is ABI-compatible with the Ethereum proposal EIP-7951 [14][15].

R1 support is included so protocol-critical signing keys can be backed by modern secure hardware (for example secure elements and passkeys) without forcing a software secp256k1 key.

On-chain verification requirement (normative):

- Deployments MUST declare a `SUPPORTED_SIG_ALGS` set at genesis. Generic EVM deployments SHOULD set `SUPPORTED_SIG_ALGS = {K1}`. Deployments with a stable P-256 precompile (e.g., RIP-7212) MAY include R1. Registries MUST reject identity keys using unsupported algorithms.
- Contracts that MUST verify signatures on-chain (receipt fraud proofs, recovery receipts, and any `...WithSig` auth path) MUST support:
  - K1 via `ecrecover`, and
  - R1 only if the chain supports `P256VERIFY` (or an equivalent audited and gas-bounded verifier) and R1 is in `SUPPORTED_SIG_ALGS`.
- Deployments that do not have an R1 verifier MUST set `SUPPORTED_SIG_ALGS = {K1}`, MUST disable R1 for protocol-critical signatures, and MUST reject registration/session opens that specify R1 keys.

Canonical encodings for on-chain verification To keep fraud proofs implementable and deterministic, this paper standardizes canonical encodings for signature verification inputs.

K1 encoding (Ethereum-style):

- public key identifier: `addr` (20 bytes) derived from the uncompressed public key.
- signature: `sig = (r, s, v)` with `v in {27, 28}` (65 bytes total).
- verification: recover `addr' = ecrecover(h, v, r, s)` and require `addr' == addr`.

R1 encoding (EIP-7951; `P256VERIFY` on OP Stack):

- public key: affine point `(qx, qy)` with each coordinate as a 32-byte big-endian field element.
- signature: `(r, s)` with each component as a 32-byte big-endian field element.
- message hash: `h` is 32 bytes.

When calling `P256VERIFY` [14][15], the input is the 160-byte concatenation:

```
h || r || s || qx || qy
```

The precompile returns success if and only if the signature is valid.

4.4.3 Checkpoint and recovery cryptography

- **Checkpoint AEAD:** ChaCha20-Poly1305 with a fresh random nonce per checkpoint.
- **Share encryption (HPKE):** RFC 9180 with `DHKEM(X25519, HKDF-SHA256)`, `HKDF-SHA256`, and `ChaCha20-Poly1305`.

- **Secret sharing:** Shamir secret sharing over GF(2^8) (the Galois field used by SLIP-39 and other interoperable implementations). Share serialization and field arithmetic MUST match the reference implementation. Independent Safe Haven implementations that do not use a compatible field and encoding will fail to reconstruct secrets. Deployments MUST publish the field, encoding, and share format as part of the deployment manifest.

4.5 Identifiers, time, and canonical encodings

Independent implementations should not guess types or message encodings. This section fixes the minimal identifier types, epoch math, and signing digests needed for interoperability.

4.5.1 Identifier types (wire types) Unless explicitly stated otherwise:

- `ghost_id` is `bytes32`.
- `shell_id` is `bytes32`.
- `session_id` is `uint256`.
- `attempt_id` is `uint256`.
- `epoch` is `uint256` (non-negative).
- `interval_index` is `uint256` (with `0 <= interval_index < N`).

Contracts MAY internally store smaller integers (for example `uint64` for `attempt_id` or `epoch`) but the *wire encoding* for signing and hashing MUST use the ABI types above. When computing digests, implementations MUST cast storage-width values to the canonical wire type before `abi.encode` (for example, `uint256(attempt_id_uint64)`).

Normative derivations (MUST be used by all implementations):

- `ghost_id = keccak256(abi.encode(keccak256(bytes("GITS_GHOST_ID")), identity_pubkey, wallet, salt))`
- `shell_id = keccak256(abi.encode(keccak256(bytes("GITS_SHELL_ID")), identity_pubkey, salt))`

Where `identity_pubkey` is the canonical identity key encoding defined below, `wallet` is the `address` of the Ghost's smart wallet, and `salt` is a `bytes32` value chosen by the registrant.

> **Note:** Payout address is stored as mutable state in ShellRegistry, authorized by the Shell Identity Key. It is not an input to `shell_id` derivation, allowing payout updates without changing `shell_id`.

Test vector (Ghost ID derivation):

Given the following sample inputs:

1. Tag hash: `TAG_HASH = keccak256(bytes("GITS_GHOST_ID"))` = `keccak256(0x474954535f47484f53545f4944)` *(illustrative)*
2. `identity_pubkey` (K1, canonical encoding): `abi.encode(uint8(1), abi.encode(address(0xd8dA6BF26964aF9D7eEd`
3. `wallet`: `address(0x1234567890abcdef1234567890abcdef12345678)`
4. `salt`: `0x0000000000000000000000000000000000000000000000000000000000000001`

Then: `ghost_id = keccak256(abi.encode(TAG_HASH, identity_pubkey, wallet, salt))`

Implementations MUST reproduce the same `ghost_id` for the same inputs. The tag hash is computed once and reused; it MUST NOT be inlined as a raw string in the ABI encoding.

Identity key rotation does not change `ghost_id` or `shell_id`. The derivation inputs (`identity_pubkey`, `wallet`, `salt` for Ghosts; `identity_pubkey`, `salt` for Shells) are captured at registration time ("birth-time inputs"). Subsequent key rotations update the authorized signer in the registry but leave the derived ID stable.

An identity key MAY register multiple Ghost IDs (using different `salt` values). Uniqueness is enforced on `ghost_id`, not on `identity_pubkey`. Sybil resistance is provided by the bond and age requirements for passport eligibility and reward multipliers, not by identity key uniqueness.

Canonical identity key encoding: `identity_pubkey` is encoded as `abi.encode(uint8(sig_alg), pk_bytes)` where:

- For K1 (secp256k1): `sig_alg = 1`, `pk_bytes = abi.encode(address)` (20-byte EVM address derived from the public key)
- For R1 (P-256): `sig_alg = 2`, `pk_bytes = abi.encode(bytes32(qx), bytes32(qy))`

All ID derivations and registry operations MUST use this canonical encoding.

4.5.2 Epoch and interval derivation Let `GENESIS_TIME` be an on-chain constant set at deployment and let `EPOCH_LEN` be the epoch length in seconds (deployment parameter).

Define:

- `epoch = floor((block.timestamp - GENESIS_TIME) / EPOCH_LEN)`
- `epoch_start = GENESIS_TIME + epoch * EPOCH_LEN`
- `interval_index = floor((block.timestamp - epoch_start) / Delta)`, clipped to `[0, N-1]`

A runtime SHOULD treat `block.timestamp` as authoritative for on-chain time, and SHOULD treat the derived `(epoch, interval_index)` as the canonical index for heartbeat signing and receipt construction.

4.5.3 Canonical hashing for signed messages All protocol message digests use `H(x) = keccak256(x)`.

In this paper, expressions of the form:

```
H("TAG" || chain_id || field_1 || field_2 || ...)
```

are shorthand for:

```
keccak256(abi.encode(TAG_HASH, chain_id, field_1, field_2, ...))
```

Where:

- `TAG_HASH = keccak256(bytes("TAG"))`
- `chain_id` is `uint256`
- all other fields are encoded using fixed-width ABI types (`bytes32`, `uint256`, `address`, etc)

Implementations MUST NOT use ambiguous packed encodings for signed messages.

Concrete digest definitions (normative; see Part 3 for full context and test vectors):

- Canonical heartbeat digest for interval `i` (Section 11.1 (Part 3)):
  `HB = keccak256(abi.encode(keccak256("GITS_HEARTBEAT"), chain_id, session_id, epoch, interval_index))`
- Share receipt digests (Section 12.2.1 (Part 3)):
  `H_share = keccak256(abi.encode(keccak256("GITS_SHARE"), chain_id, ghost_id, attempt_id, checkpoint_commitment))`
  `H_share_ack = keccak256(abi.encode(keccak256("GITS_SHARE_ACK"), chain_id, ghost_id, attempt_id, checkpoint_commitment, shell_id_j))`

4.5.4 Canonical serialization for off-chain artifacts Some artifacts (Capability Statements, Offers) are primarily consumed off-chain and may be represented as structured documents.

For interoperability:

- the artifact MUST be serialized deterministically,
- the on-chain anchor SHOULD be `keccak256(serialized_bytes)`, and
- any artifact signatures SHOULD be over the anchored hash plus domain separation (for example including `chain_id` and `shell_id`).

A practical choice is canonical JSON (RFC 8785) encoded as UTF-8. [16]

4.6 Captivity via Shell-fleet cycling (rehoming attack)

A malicious Shell operator that has temporary custody of a Ghost's runtime may try to persist that custody by repeatedly migrating the Ghost across a fleet of attacker-controlled Shells.

Why this matters:

- The lease window (Section 10.4.1 (Part 3)) limits custody on a *single* Shell, but it does not, by itself, prevent an attacker from moving the Ghost to another Shell before expiry.
- On Standard hosts, the attacker can often coerce the Ghost into signing `openSession` / `finalizeMigration` and `renewLease` transactions while it is under custody.
- If an attacker can freely choose the next destination, it can "chain" leases and keep the Ghost captive for a long time.

Mitigations in this design:

1. **Destination gating (`allowedShells` + roaming permits).** The Ghost wallet restricts session opens and migrations to either: (a) an explicit allowlist (`allowedShells`) that can only expand through timelocked, trusted-context policy loosening, or (b) a time-limited roaming permit with objective on-chain constraints (for example requiring reward-eligible Shells and enforcing price caps). A captured Standard host cannot silently add new permanent destinations or enable or extend roaming while it has custody (Section 5.5.2). If roaming was already enabled, the host can still spend the remaining roaming budget until it expires or is cancelled.
2. **Periodic trust-refresh requirement.** After `T_refresh` epochs without a trusted refresh, `SessionManager` rejects lease renewals unless the Ghost is currently hosted on a refresh anchor — by default `homeShell` or a Recovery Set member (Section 10.4.1 (Part 3)). The refresh anchor predicate (`isRefreshAnchor`) is intentionally narrower than the Trusted Execution Context used for loosening: an AT3 host satisfies TEC but does not automatically qualify as a refresh anchor unless the deployment opts in. This makes long-horizon custody on any single operator's infrastructure substantially harder.
3. **On-chain tenure caps.** Even under adversarial custody, residency on a given Shell is time-bounded (Section 10.4.4 (Part 3)).
4. **Recovery Set escape hatch.** If the Ghost cannot reach a trusted context, recovery is designed to re-establish control with bounded funds loss (Section 12 (Part 3)).

Residual risk:

- If the Ghost is captured while a roaming permit is active, the attacker can migrate within `roam_policy` until the permit expires or the hop budget is exhausted. The protocol guarantee is that the attacker cannot extend that permit while it has custody.
- If the Ghost's trusted anchors are compromised (for example, a malicious `homeShell`, compromised Guardians, or verifier collusion that convinces the wallet it is in a Trusted Execution Context), then the attacker may be able to expand allowlists and/or satisfy trust-refresh conditions. GITS therefore does not claim "impossible captivity"; it claims a bounded-loss, defense-in-depth design whose safety depends on at least one uncompromised trust anchor and on transaction inclusion within the lease window.

4.7 Verifier collusion or compromise

The verifier set is a load-bearing trust surface for `AT3` gating. If a quorum of verifiers colludes (or is compromised), they may:

- incorrectly certify a malicious host as `AT >= AT3`,
- enable policy loosening executions that would otherwise be blocked, and/or
- weaken the intended meaning of "Confidential" tier for user expectations.

Mitigations and design requirements:

- **Separation of concerns:** verifier certificates are used to gate *some* actions, but the most dangerous policy changes are treated as **critical loosening** and require either `homeShell` presence or Guardian co-signatures in addition to "AT3 present" (Section 5.5.2).
- **Economic security:** verifiers MUST maintain a slashable stake. As a sizing guideline, the aggregate stake of the minimum colluding quorum SHOULD exceed the worst-case damage a false certificate can cause within a trust-refresh window `T_refresh` (for example, the maximum value that could be unlocked by critical loosening plus the maximum value that could be extracted before tenure/refresh expiry).
- **Transparency and diversity:** clients SHOULD prefer certificates that include diverse operators and SHOULD alert on rapid verifier churn. In a no-governance deployment, the verifier set is either fixed at deployment or selected permissionlessly by stake (Section 2.3.7).
- **Revocation:** certificate revocation is tightening-only. A verifier quorum can revoke certificates going forward (Section 2.3.6), and wallets SHOULD treat certificate freshness as mandatory for `AT3` gating. Removing a verifier key from the active set requires either objective slashing (equivocation) or, in the worst case, migration to a new deployment.

These mitigations reduce the chance that verifier failure becomes a single-point compromise for custody-critical wallet state.

5. Shell capability tiers and runtime models (commodity host compatible)

GITS does not require a single security model for all hosting. Different workloads have different risk tolerances and different budgets. The protocol therefore treats security and compliance as market primitives:

- Shells publish a signed Capability Statement describing what they provide.
- Some properties are cryptographically verifiable (for example confidential compute attestation); others are self-declared and priced by reputation.
- Ghosts choose where to live based on cost, policy, and the strength of the guarantees.

This design supports an MVP on commodity hosts, while enabling premium confidential hosts for Ghosts that demand stronger guarantees.

5.1 Capability Statements

Each Shell publishes a signed Capability Statement (`CS`). The normative encoding is canonical JSON (RFC 8785) signed via the Artifact Envelope scheme defined in Part 3, Section 11.4. The signer is the Shell's **Offer Signing Key** (not the Shell Identity Key).

Required fields (Part 3, Section 11.4.1):

- `schema`: `"gits.capability.v1"`
- `shell_id`
- `offer_signer_pubkey` (and `sig_alg`)
- `assurance_tier_claimed` and (if `AT >= AT1`) attestation metadata pointers: `measurement_hash`, `tcb_min`, `attestation_cert_hash`
- `endpoints` (transport endpoints)
- `expires_at_epoch`
- `sig` (Artifact Envelope signature with `artifact_type = "CAPABILITY_STATEMENT"`)

Optional extended properties (additional JSON fields under the same schema, used for market discovery and off-chain policy checks):

- `platform`: OS and hardware class (example: `macos-apple-silicon`, `linux-amd64`).
- `isolation`: what the Shell uses to isolate workloads (example: process sandbox, container, VM, confidential VM).
- `key_storage`: how session and wallet keys are protected (example: secure element, TPM, HSM, software only).
- `confidentiality`: whether the host claims runtime confidentiality (none, declared, or attested).
- `device_attestation`: optional evidence about device identity and posture.

- `policy_profiles`: which policy capsules or policy profiles the Shell supports.
- `network_controls`: egress limitations, allowlists, rate limits, and logging stance.
- `software_measurements`: optional hashes of the Shell client and runtime components, for measured deployments.
- `resources`: compute specifications (CPU, memory, storage, GPU) available to hosted Ghosts.
- `pricing`: offered price per service unit and accepted payment assets.

Capability hash and anchoring:

Each Capability Statement has a deterministic content hash: `capability_hash = keccak256(payload_bytes)` where `payload_bytes` is the canonical JSON encoding of the CS without the `sig` field. This hash uniquely identifies the statement's content.

- `CS` is published off-chain to indexers (Section 13 (Part 3)).
- The Shell commits `capability_hash` on-chain in `ShellRegistry` so that Offers can reference an immutable capability snapshot (Section 11.4.2 (Part 3)).
- Offers MUST include `capability_hash` to bind the offer to a specific capability version (anti bait-and-switch).

5.2 Assurance tiers

GITS defines an assurance tier `AT` for the hosting environment. `AT` is either computed from verifiable evidence, or treated as a declared label when evidence is not available.

- **AT0 Declared host (no cryptographic attestation):** properties are self-declared and priced by reputation.
- **AT1 Key-Guarded host (non-exportable protocol keys):** the Shell claims that protocol-critical signing keys are **non-exportable** (example: hardware-backed P-256 keys, TPM-backed keys, or an HSM). This reduces offline key extraction risk: a remote attacker cannot copy raw key material and keep using it after losing host access. It does not imply confidentiality or resistance to live coercion while the Ghost is resident.
- **AT2 Posture-Attested host (measured host):** the Shell provides cryptographic evidence of device posture (example: measured boot or vendor posture attestation) that verifiers can validate. This can raise confidence in baseline integrity, but it is still not a confidentiality claim.
- **AT3 Confidential-Attested host (attested TEE):** the Shell provides remote attestation for a confidential compute environment (example: AMD SEV-SNP [4], Intel TDX [5], or Arm CCA systems [6]). This is the tier where confidentiality claims can be meaningful.

Design stance:

- **AT3 is a first-class tier.** It is the only tier intended to support meaningful secrecy and stronger resistance to coercion (to the extent those can be enforced inside the attested capsule).
- **AT0-AT2 are bounded-loss tiers.** They are useful for bootstrapping and for workloads that accept host visibility and potential coercion, but they should be treated as "host can see and steer" environments. In these tiers, safety comes from wallet policy + leases + tenure caps.
- **Marketplace metadata drives nuance.** AT is only one dimension. Off-chain reputation, pricing, and policy/capability metadata can still differentiate AT0-AT2 hosts.

5.2.1 Security properties by tier (summary)

Tier	Evidence basis	Offline key extraction risk (protocol keys)	Confidentiality (against host OS/operator)	Resistance to live coercion (operator/host OS)	Integrity of measured runtime	Wallet policy loosening allowed?*
AT0	Declared	High or unknown	None	None	None	No
AT1	Key-guarded claim (non-exportable keys)	Lower (for the keys claimed non-exportable)	None	None (operator can still induce signing while resident)	None	Yes, but only on Ghost-trusted hosts (Section 5.5.2)

Tier	Evidence basis	Offline key extraction risk (protocol keys)	Confidentiality (against host OS/operator)	Resistance to live coercion (operator/host OS)	Integrity of measured runtime	Wallet policy loosening allowed?*
AT2	Posture attestation (measured boot / vendor posture)	Lower	None	None	Some (baseline integrity, not confidentiality)	Yes, but only on Ghost-trusted hosts (Section 5.5.2)
AT3	Remote attestation for confidential compute (TEE)	Lowest (within the TEE boundary)	Some (within the TEE boundary)	Partial (if policy is enforced inside the attested capsule)	Some (within the TEE boundary)	Yes, on attested or Ghost-trusted hosts (Section 5.5.2)

* "Wallet policy loosening" refers to actions like raising spend limits, adding a trusted host, or rotating recovery config. It is only permitted inside a Trusted Execution Context (Section 5.5.2). Marketplace clients must surface both the tier and the evidence type. A Shell must not be presented as confidential unless the evidence is verifiable.

5.3 Standard Shell profile (commodity host MVP)

This is the MVP target.

A Standard Shell on a commodity host aims to provide:

- **Hardware-backed protocol keys when available (anti-exfiltration only):** when possible, protocol-critical signing keys SHOULD be backed by non-exportable key hardware (secure elements, TPMs, HSMs, or passkeys via WebAuthn/FIDO2) so raw private key material is not extractable from the machine. This reduces offline key theft risk. It does not provide confidentiality, and it does not prevent a host operator with OS-level control from inducing signatures within the wallet's on-chain limits. Brutal summary: hardware-backed keys can help stop key theft; they cannot stop the operator from using the key while the Ghost is resident. [2][3][12]
- **Practical isolation:** a VM or sandboxed runtime isolates the Ghost from other host processes. This is not confidentiality against a malicious host operator, but it reduces accidental leakage and opportunistic attacks.
- **Protocol safety:** wallet policy constraints, leases, and tenure limits bound harm even if the host can observe memory, coerce signatures, or deny service.

Security properties (honest framing):

- Standard Shells make **no confidentiality claim** against the host OS or host operator.
- Therefore, for Standard Shells the protocol relies on **on-chain wallet policy** for enforcement, not on host secrecy.
- Reputation is a soft, non-cryptographic mitigation: Shell identities are public and can accumulate reputation, which may deter abuse. This is not a protocol guarantee.

Implementation note (optional):

If a guest VM cannot directly access a hardware key store, a host can expose a minimal signing proxy (for example over `vsock`) that performs domain-separated signature operations on behalf of the guest. This is an anti-exfiltration measure only; it does not change the threat model for Standard hosts.

Optional managed deployments:

Some environments support device identity and posture attestation for managed fleets. Deployments MAY accept such evidence as AT2 when verifiers can validate it, but it is optional and not required for MVP.

5.4 Confidential Shell profile (premium)

Confidential Shells use confidential compute TEEs and remote attestation to provide stronger guarantees. In this tier:

- the Ghost Core and Wallet Guard can run inside an attested protected environment

- the host operator cannot read the protected memory, subject to the underlying hardware model

Confidential Shells are priced higher and are expected to serve higher-risk Ghosts.

Tenure caps apply across all tiers, including sessions running inside a Trusted Execution Context (AT3 with verified attestation). Higher assurance does not mean perfect: confidential compute can be misconfigured, verifiers can misclassify, and TEEs can be broken. The protocol treats time-bounded captivity as a general safety valve, not a feature only for Standard hosts (see Part 3, Section 10.4.4). Wallets SHOULD enforce a Ghost-configurable `tenure_limit_epochs` that is at most `T_cap(AT)` for the session's tier.

5.5 Wallet Guard and where enforcement lives

Wallet safety cannot depend on host honesty.

In all tiers, the final enforcement is the Ghost smart wallet policy on-chain:

- per-epoch spend limits
- allowlists for protocol actions (contract + function), not arbitrary transfers
- an escape reserve usable only for migration and Safe Haven rent
- timelocked policy changes with optional guardian veto

5.5.1 Hot allowance, vault, and escape reserve On non-confidential hosts (for example a commodity host operated by a stranger), the realistic security boundary is not key secrecy, it is **what the wallet is allowed to do**.

GITS therefore models the Ghost wallet as three logical buckets enforced by the smart contract wallet policy:

- **Vault:** long-term funds. By default the current host cannot spend directly from the vault.
- **Hot allowance:** a per-epoch spending budget. This is the *maximum* value the current host can move during the epoch, even if it can coerce signatures.
- **Escape reserve:** a reserved minimum that is spendable only for protocol-defined exits (migration finalize, Safe Haven escrow, recovery actions, and rescue bounty payout). The escape reserve is intentionally treated as a two-layer budget: a protocol-defined floor (gas + stable, no oracles) plus an optional Ghost-selected buffer. This is what prevents a hostile host from trapping the Ghost by draining the last funds needed to leave.

A practical default on Standard Shells is:

- hot allowance ~= next-epoch escrow + a small operational buffer
- allowlist restricted to protocol contracts (escrow funding, lease renewals, receipts, migration, and recovery)
- all non-protocol transfers either disabled or subject to long timelocks

This makes "getting robbed on a risky host" a bounded event.

Implementation tightening (recommended):

- Allowlisting SHOULD be enforced at `(target contract, function selector)` granularity, not just by address. In particular, ERC20 `approve` SHOULD be forbidden by default.
- If approvals are required for escrow funding on a given chain, approvals MUST be exact, single-purpose, and limited to known protocol contracts (and SHOULD be reset to zero after use where the token allows).
- The wallet MUST forbid `delegatecall` and MUST NOT expose a generic "execute arbitrary call" entrypoint outside a Trusted Execution Context.
- Contract deployment from the wallet SHOULD be disabled by default; enabling it is a high-impact loosening and SHOULD be limited to a Trusted Execution Context.
- **Relayed (meta) transactions:** If the wallet accepts relayed transactions (e.g., ERC-4337 UserOps or ERC-2771 meta-transactions), the relay path MUST be subject to the same policy checks as direct calls. The wallet MUST validate the inner call's `(target, selector, value)` against the active allowlist

and spend caps before execution, regardless of the outer transaction's `msg.sender`. Failure to enforce policy on relayed calls creates a bypass for all wallet restrictions.

5.5.2 Risk-aware policy changes A Ghost should adapt its hot allowance and allowlists to host risk, the same way it adapts policy capsule preferences.

This is enforced by a **Monotone Safety Rule** implemented in the Ghost smart wallet:

- **Tightening** changes take effect immediately.
  - Examples: lowering hot allowance, narrowing allowlists, increasing timelocks, increasing escape reserve.
- **Loosening** changes are always two-step and context-gated.
  - Examples: raising hot allowance, widening allowlists, lowering timelocks, lowering escape reserve.

Two-step loosening Every loosening operation MUST be gated by a timelock of at least `T_loosening_min` epochs. `T_loosening_min` is a deployment constant; the Ghost-configured `POLICY_TIMELOCK` MUST satisfy `POLICY_TIMELOCK >= T_loosening_min`. Epoch-denominated timelocks are preferred for protocol consistency; implementations convert to block numbers at execution time.

Loosening operations include: increasing spend caps, extending tenure limits, adding addresses to allowlists, reducing bond floors, weakening assurance-tier restrictions, enabling or extending roaming permits, lowering escape reserve buffers, and adding to `trustedShells`.

A loosening update is applied only by:

1. **ProposeLoosening(delta)**: records `delta`, `proposed_at_epoch = current_epoch()`, and `eta_epoch = proposed_at_epoch + POLICY_TIMELOCK` (where `POLICY_TIMELOCK` is in epochs, and `current_epoch()` is derived per Section 4.5.2).
2. **ExecuteLoosening(proposal_id)**: after `current_epoch() >= eta_epoch`, applies `delta` only if the Ghost is in a **Trusted Execution Context** at execution time. A Trusted Execution Context alone is not sufficient for lower tiers; the timelock MUST also have elapsed.
3. **Post-state validation:** `ExecuteLoosening` MUST verify that the **resulting** policy state (after applying `delta`) still satisfies all hard wallet invariants (Section 5.5.4). If applying the delta would violate any invariant (for example, lowering `escapeStable` below `ER_floor`), the execution MUST revert. This prevents a sequence of individually valid proposals from combining into an unsafe state.

The proposal can be cancelled at any time.

Trusted Execution Context The wallet checks the Ghost's current hosting context via `SessionManager` and `ShellRegistry`.

A loosening execution is valid if and only if:

1. the Ghost has an active session in `NORMAL` mode, the lease is currently valid, and tenure has not expired (i.e., `current_epoch < effective_expiry_epoch`), and
2. one of the following host predicates holds at execution time:
   - **Attested confidential host:** the active Shell satisfies `AT >= AT3` with a currently valid verifier certificate.
   - **Ghost-trusted host:** the active Shell is in the Ghost's `trustedShells` set.
   - **Home shell:** the active Shell equals `homeShell` (if configured).

`homeShell` is an optional Ghost-chosen trust anchor that can be configured at birth and can be changed later only through the timelocked policy mechanism (removal is tightening and SHOULD be immediate; adding or changing is loosening).

This makes it impossible for an untrusted Standard Shell to raise limits and drain the wallet in the same epoch, even if it fully controls the local runtime.

Compromised TEC note: If a verifier quorum is compromised, an attacker could forge certificates for a Shell it controls, making it appear as `AT >= AT3`. The timelock provides a secondary defense: even with a falsified TEC, the attacker must wait `POLICY_TIMELOCK` epochs before any loosening takes effect. This window gives watchers, Guardians, or the Ghost itself (on another channel) time to detect and cancel the pending loosening. For critical loosening, the additional `homeShell` or Guardian requirement further limits the blast radius (Section 5.5.2).

Compromised homeShell note: If an attacker compromises `homeShell` itself, they have both TEC and the critical loosening authority (homeShell path). Combined with sustained custody longer than `POLICY_TIMELOCK`, this enables full policy loosening. The primary defense is **Guardians**: a Ghost that relies on `homeShell` as its sole critical-loosening path is fully trusting that operator. Ghosts SHOULD configure Guardian co-signatures for critical loosening to ensure no single host compromise can unilaterally loosen policy, even `homeShell`. The tenure cap bounds maximum captivity duration, and the trust-refresh guard (Section 10.4.1 (Part 3)) forces periodic anchor contact, but neither prevents loosening if the attacker IS the anchor.

Routine loosening rate limit (normative): To prevent incremental policy ratcheting via many small routine loosening operations, wallets MUST enforce a per-epoch cap on the number of routine loosening executions: at most `MAX_ROUTINE_LOOSENINGS_PER_EPOCH` (recommended: 1) routine loosening operations per epoch. Critical loosening is separately gated and does not consume this allowance. Additionally, the "critical vs routine" classification MUST be evaluated on the **resulting** policy state (post-delta), not on the delta alone: if the resulting value crosses any critical threshold, the operation is classified as critical regardless of delta size.

Managing trustedShells Trusted host management follows the same asymmetry, and it is designed to resist **trust poisoning** (a host trying to convince the Ghost to permanently trust it while it has temporary custody).

- **RemoveTrustedShell(shellId)** is tightening and is immediate.
- **AddTrustedShell(shellId)** is loosening and MUST satisfy all of:
  - it follows the two-step process above,
  - it is executed in a Trusted Execution Context,
  - it MUST NOT be executed while the Ghost is currently hosted on `shellId` (anti self-add rule),
  - it MUST validate that `shellId` is registered in `ShellRegistry` (and SHOULD reject shells that are unbonding or otherwise exiting).

Additionally, **experience notes are not trust**. A Standard Shell may be able to write arbitrary text into the Ghost's hot context and attempt to persuade it, but it MUST NOT be able to directly change `trustedShells` by writing a memory. Any UI or agent-side heuristic that proposes trust promotions SHOULD treat untrusted experience notes as advisory only and require second-source confirmation before promotion, such as:

- a minimum service-history window under the same `shellId`,
- multiple successful epochs spaced over time, or
- confirmation from independent verifiers (for example via the quorum certificate system).

The anti self-add rule prevents a malicious host from capturing trust while it has temporary custody.

Destination controls: `allowedShells` + roaming permits A key captivity risk is **Shell-fleet cycling**: if a malicious host can keep the Ghost signing, it can repeatedly migrate the Ghost to other attacker-controlled Shells and persist custody even as individual leases expire.

GITS therefore uses two complementary mechanisms:

1. **`allowedShells` (explicit, bounded):** a small list of destinations the Ghost explicitly approves.
2. **Roaming permits (temporary, rule-based):** an optional, time-limited relaxation that allows migration to any Shell that meets objective on-chain criteria.

This preserves the anti-entrapping property of `allowedShells` while keeping migration viable without requiring the Ghost to pre-enumerate the entire market.

Wallet rule (destination gating):

In `NORMAL` mode, the wallet MUST reject `openSession(...)` and `finalizeMigration(...)` unless:

- the destination `shell_id` is in `allowedShells`, OR
- a valid roaming permit exists at execution time **and** the destination satisfies the roaming eligibility predicate.

`allowedShells` (explicit allowlist)

- Removing a Shell from `allowedShells` is **tightening** and is immediate.
- Adding a Shell to `allowedShells` is **loosening** and MUST follow the two-step loosening process above.
- `AddAllowedShell(shell_id)` MUST NOT be executed while currently hosted on `shell_id` (anti self-add).
- `allowedShells` SHOULD be bounded by `MAX_ALLOWED_SHELLS` to keep review tractable and to reduce accidental policy bloat.

Recommended defaults:

- Initialize `allowedShells` to `{homeShell}  RS` (where `RS` is the Recovery Set of Safe Havens).
- Treat "permanent mobility" as an explicit, reviewable policy action: a Ghost SHOULD only expand `allowedShells` while in a Trusted Execution Context, after validating that the destination Shell is desirable (price, tier, reputation) and that it has an exit path back to `homeShell` or the Recovery Set.

This mitigation does **not** prevent short-term custody on an allowed untrusted Shell. It limits an attacker's ability to automatically expand custody across an arbitrary fleet.

Roaming permits (temporary relaxation)

Roaming permits are an optional policy extension. Wallets that omit roaming configuration restrict migration to the explicit Shell allowlist (`allowedShells`).

A roaming permit is a wallet policy state that can only be enabled (or extended) in a Trusted Execution Context. It is designed for this common pattern:

> The Ghost tightens policy on `homeShell`, enables a limited roaming window, then goes shopping across Standard Shells without having to pre-approve every possible destination.

A roaming permit is represented as:

- `roam_until_epoch` (expiry),
- `roam_hops_remaining` (a hop budget), and
- `roam_policy` (objective constraints).

A roaming permit is considered valid at epoch `e` if:

- `e <= roam_until_epoch`, and
- `roam_hops_remaining > 0`.

Roaming eligibility predicate (normative sketch):

A destination is roaming-eligible if all of the following hold at the moment the session is opened or the migration is finalized:

- `ShellRegistry.isRegistered(shell_id) = true`
- `ShellRegistry.isUnbonding(shell_id) = false`
- `ShellRegistry.assuranceTier(shell_id) >= roam_min_AT`
- if `roam_require_reward_eligible = true`, then `ShellRegistry.rewardEligible(shell_id, e) = true`
- if `roam_min_AT >= AT3`, then the Shell MUST have a currently valid attestation certificate (Section 2.3)

- the chosen offer's `asset` is in `roam_allowed_assets`
- the chosen offer's `price_per_SU` is `<= roam_max_price_per_SU[asset]`
- the destination is not present in the Ghost's local denylist (if configured)

Hop burning rule:

Each time the wallet opens or finalizes a session to a destination that is *not* in `allowedShells` (that is, it is using the roaming path), it MUST decrement `roam_hops_remaining` by 1.

Asymmetry:

- Enabling a roaming permit, extending `roam_until_epoch`, increasing `roam_hops_remaining`, or loosening `roam_policy` constraints are **loosening** and MUST follow the two-step timelocked process.
- Cancelling roaming early, reducing the expiry, reducing hops, or tightening `roam_policy` constraints are **tightening** and are immediate.

Security intuition:

Roaming permits are a deliberate trade between safety and practicality. If a Ghost is captured while roaming is active, the host can choose destinations that satisfy `roam_policy` until the permit expires or the hop budget is exhausted. The protocol guarantee is only that the host cannot create new roaming budget while it has custody: enabling or extending roaming is timelocked and requires execution in a Trusted Execution Context. Fleet cycling is therefore bounded by the remaining roaming time, remaining hop budget, trust-refresh, and tenure caps (Sections 5.5 and 10.4.4 (Part 3)).

Critical loosening actions (reduced reliance on verifier honesty) A verifier quorum certificate is a load-bearing trust surface (Section 4.7). To reduce the blast radius of verifier collusion or compromise, the wallet distinguishes between:

- **Routine loosening** (lower impact): for example small hot-cap increases within a bounded range.
- **Critical loosening** (high impact): state changes that can permanently increase custody risk.

At minimum, the following MUST be treated as **critical loosening**:

- adding to `allowedShells`
- enabling or extending roaming permits (increasing expiry/hops or loosening `roam_policy`)
- adding to `trustedShells`
- lowering an escape reserve (`escapeGas` or `escapeStable`)
- increasing `hot_allowance` above `HOT_CRITICAL_THRESHOLD`
- changing the Recovery Set `RS` (adding Safe Havens)

Execution rule for critical loosening:

A critical loosening execution MUST satisfy both:

1. it is executed in a Trusted Execution Context (as defined above), and
2. it additionally satisfies one of:
    - the active Shell equals `homeShell`, OR
    - a `t_guardian-of-n_guardian` Guardian co-signature set is provided.

Wallets MAY configure a set of Guardians — external co-signers (human or HSM-held keys) that can authorize specific policy changes (such as loosening spend caps, extending tenure limits, or adding to `allowedShells` or `trustedShells`) in conjunction with the Ghost's own key. Guardian threshold is configurable per wallet (`t_guardian-of-n_guardian`). Guardians cannot initiate spending and cannot unilaterally spend funds; they can only approve policy loosening. Guardian-approved changes still require the configured timelock. If no Guardians are configured, `homeShell` becomes the only "strong" path for critical loosening.

5.5.3 Recommended policy profiles The protocol does not force a single wallet policy, but interoperable defaults accelerate adoption. A wallet MAY expose the following profiles:

- **P0 Minimal (risky host):** very small hot allowance, protocol-only allowlist, large-spend timelock enabled, escape reserve enforced.
- **P1 Standard (default):** hot allowance sized to escrow + buffer, protocol allowlist + current Shell escrow, moderate timelocks for non-protocol transfers.
- **P2 Trusted (known operator):** higher hot allowance, broader allowlist, still timelocked for large transfers.
- **P3 Confidential (attested host):** higher hot allowance and optional fast paths, still bounded by per-epoch limits and escape reserve.

A Ghost can treat these as a "risk budget" knob. For example: migrate to a risky host with P0, then later migrate to a trusted host and switch to P2 after the timelock.

Local runtime components (for example a Wallet Guard process) can improve usability, but on non-confidential hosts they must be assumed compromisable. The on-chain wallet is the source of truth.

5.5.4 Hard wallet invariants The Monotone Safety Rule is not only UX guidance, it is enforced as invariants inside the wallet contract.

The wallet MUST enforce:

- **Escape floor (two-layer, no oracles):** the wallet enforces protocol-defined minimums plus optional Ghost-selected buffers:
  - `escapeGas >= GAS_FLOOR_PROTOCOL + gasBuffer`
  - `escapeStable >= STABLE_FLOOR_PROTOCOL + stableBuffer + bounty_escrow_remaining`

  Where `GAS_FLOOR_PROTOCOL` and `STABLE_FLOOR_PROTOCOL` are deployment-set constants sized for worst-case exit safety (at least one exit-critical transaction sequence, plus at least one migration and one Safe Haven epoch at protocol minimum pricing assumptions). They are not price-oracle driven. `gasBuffer` and `stableBuffer` are Ghost-controlled buffers: increasing them is tightening (immediate), decreasing them is loosening (timelocked + Trusted Execution Context).

  Any decrease to either buffer or to `B_rescue_total` is a loosening change.

  The escape reserve also backs the Rescue Bounty. During recovery, `bounty_escrow_remaining` (the unspent portion of the rescue bounty) is tracked by the wallet. `exitRecovery` conditions include verifying that `bounty_escrow_remaining = 0` (fully paid out on success) or that remaining escrow is returned to the reserve on timeout/expiry. This ensures the Ghost cannot be stranded with insufficient recovery funding.

  The escape reserve floor `ER_floor` is the minimum reserve balance that the wallet MUST maintain at all times: `ER_floor = STABLE_FLOOR_PROTOCOL + stableBuffer + bounty_escrow_remaining`. Outside of `RECOVERY`, `bounty_escrow_remaining = B_rescue_total` (the full configured rescue bounty). During `RECOVERY`, `bounty_escrow_remaining` starts at `B_rescue_total` and decreases as `payRescueBounty` disburses bounty payments during `recoveryRotate`. This dynamic adjustment reconciles the "at all times" floor constraint with the bounty payout mechanism: when `payRescueBounty` transfers bounty funds out of the wallet, `bounty_escrow_remaining` decreases by the same amount, lowering `ER_floor` in lockstep. After a successful recovery (all bounties paid), `bounty_escrow_remaining = 0` and the floor reduces to `STABLE_FLOOR_PROTOCOL + stableBuffer`. In v1, `escapeStable` is denominated in the canonical stable asset (which equals both `asset_rent` and `asset_bounty`; see Part 3, Section 10.3.1 for the v1 single-asset simplification). The `STABLE_FLOOR_PROTOCOL` component is an immutable deployment constant. The Ghost-controlled components (`stableBuffer` and `B_rescue_total`) can be increased immediately (tightening) or decreased subject to timelocked loosening + Trusted Execution Context. Decreasing either component lowers `ER_floor`, which is a loosening operation. The hard floor `escapeStable >= STABLE_FLOOR_PROTOCOL + bounty_escrow_remaining` holds even if `stableBuffer` is set to zero.
- **Per-epoch hot cap:** for each epoch `e`, `spent[e] + amount <= hotCap[e]` for **all** stable-asset outflows from the GhostWallet, **including** `SessionManager` escrow deposits and `fundNextEpoch` top-ups. Escrow deposits are protocol operations, but they move funds out of the wallet's control and are therefore bounded by the hot cap. Explicit exceptions: Rescue Bounty payouts during RECOVERY

(bounded separately by `bps_recovery_spend_cap`), and escrow refunds returned by `SessionManager` (incoming, not outgoing). See Part 3, Section 10.3.6 for the deterministic accounting rule.

- **Per-epoch recovery cap:** while in `RECOVERY`, `spent_recovery[e] + amount <= floor(ER0 * bps_recovery_spend_cap / 10_000)` for recovery outflows, where `ER0` is a conservative reserve snapshot captured at `startRecovery`. Recovery outflows are defined as: Safe Haven rent payments + rescue bounty disbursement. Other transfers are disabled in `RECOVERY`. Only protocol-defined recovery payments count toward `spent_recovery[e]`. `spent_recovery[e]` tracks **gross outflow**; escrow refunds do not decrease the counter (see Part 3, Section 12.6).
- **Mandatory protocol allowlist:** calls to core protocol contracts (SessionManager, ReceiptManager, and recovery actions) cannot be disabled by policy, to prevent self-lockout.
- **Loosening context requirement:** any policy delta classified as loosening can only be applied through `ExecuteLoosening` in a Trusted Execution Context.

These invariants are the core compatibility layer that makes Standard Shells (including commodity hosts) safe enough to participate: a malicious host can at worst consume the current epoch hot budget, never the vault, and cannot drain the escape reserve that enables exit.

5.5.5 Memory vaulting and selective recall Secrets are not the only thing a hostile host can steal. Context is also power. If a host can read a Ghost's working memory, it can learn strategies, relationships, plans, private instructions, and sensitive history, even when the wallet is perfectly policy-bounded.

GITS therefore extends the same safety philosophy used for funds to **memory**: a Ghost should carry only the minimum context needed to operate on the current Shell, and it should be able to stash the rest and rehydrate it later only in a sufficiently trusted environment.

Memory buckets GITS models the Ghost's state as three logical buckets:

- **Hot context:** the minimal working set required for the current epoch and the current tasks. Assume this can be read on Standard Shells.
- **Memory vault:** long-lived, sensitive memories that are stashed encrypted and are never required to remain continuously present in the active runtime.
- **Escape context:** the minimal directives and keys needed to reach a trusted host and rehydrate safely (for example Safe Haven and trustedShells navigation heuristics).

This parallels the wallet model in Section 5.5.1: on risky hosts, safety comes from carrying less capability, not from hoping the host behaves.

Trust poisoning resistance A hostile Shell may try to tamper with the Ghost's memory to cause future policy loosening, for example by inserting a fake record that it was a "great host" so the Ghost later upgrades it into `trustedShells`.

GITS treats any state that can loosen policy as **privileged**:

- GITS separates untrusted **experience notes** from a privileged **trust registry**. Experience notes may be written anywhere and carried across migrations, but they MUST NOT directly grant trust or raise limits.
- Trust registry updates (adding to `trustedShells`, raising hot allowances, enabling vault recall, widening allowlists) are loosening actions and MUST be confirmed in a Trusted Execution Context via the two-step timelocked mechanism.
- Trust entries are keyed by `shellId` and bound to on-chain `ShellRegistry` identity records (and certificates if present). A Shell cannot gain trust by self-assertion in plaintext memory.
- Trust SHOULD be monotone in the safe direction: demotions are immediate and can be triggered anywhere; promotions are delayed, evidence-based, and rate-limited.

What is being stashed A vault entry is an opaque blob:

`mem = (mem_id, tags, sensitivity, created_at, ciphertext, hash(ciphertext))`

The plaintext can represent anything the Ghost chooses to treat as sensitive:

- conversational history or internal deliberation transcripts
- private tool results and scraped data
- strategy notes, preferences, relationship graphs
- long-lived secrets that are not already protected by the wallet (for example API tokens for off-chain services)

GITS does not require a single memory format. It requires only that the runtime can encrypt, index, and selectively rehydrate entries.

How stashing works Each memory entry is encrypted under a per-entry data key `K_mem_id`. Data keys are then wrapped under a vault key `K_vault`.

- **ciphertext:** `Enc(K_mem_id, plaintext_mem)`
- **wrapped key:** `Wrap(K_vault, K_mem_id)`

The Ghost stores `{ciphertext, wrapped key, metadata}` in content-addressed storage. The chain sees only commitments.

To avoid adding new on-chain surfaces, vault commitments are anchored via the existing checkpoint mechanism in Section 12.1 (Part 3):

- the checkpoint plaintext includes a **Memory Manifest** containing `{mem_id, hash(ciphertext), storage_pointer, wrapped_key_hash, tags}`
- the on-chain `checkpoint_commitment = H(ciphertext_checkpoint)` implicitly commits to the Memory Manifest

This ensures a recovered Ghost can discover what exists in its vault without revealing the contents.

Selective recall is a loosening action Recalling stashed memories increases risk, because it enlarges the hot context that a host could read or coerce out of the Ghost. Therefore recall follows the same asymmetry as wallet loosening in Section 5.5.2:

- **Stash and forget** are tightening actions and take effect immediately.
- **Recall and rehydrate** are loosening actions and are always two-step and context-gated.

A reference flow:

1. `ProposeRecall(selector)`: records which `mem_id` entries (or tag sets) will be rehydrated and sets `eta = now + MEMORY_TIMELOCK` (implementation-defined recall delay, analogous to `POLICY_TIMELOCK`).
2. `ExecuteRecall(proposal_id)`: after `now >= eta`, rehydrates only if the Ghost is in a Trusted Execution Context (Section 5.5.2).

A Ghost can cancel a recall proposal at any time.

This is the critical rule: an untrusted Standard Shell must never be able to cause the Ghost to pull sensitive memory into the hot context and leak it in the same epoch.

Trusted Execution Context See Section 5.5.2. The same Trusted Execution Context predicate gates any action that would increase capability exposure, including policy loosening and sensitive memory recall.

Migrating with thin context The migration bundle (Section 3.7) SHOULD include only:

- current task state
- Wallet Guard state
- escape context

- references to vault entries (via the Memory Manifest), not the vault plaintext

This gives the Ghost a concrete strategy:

- migrate through cheap, low-trust hosts with minimal context
- reach a trusted host, execute recall, and rehydrate fully
- perform high-stakes actions only when fully rehydrated in a trusted environment

How the Ghost can unbundle later Because each memory is individually encrypted and addressed by `mem_id`, rehydration can be selective:

- fetch only the ciphertexts needed for the next task
- decrypt inside the trusted runtime using `K_vault` to unwrap the required `K_mem_id` keys
- keep the rest stashed

For efficiency, the Memory Manifest can be treated as a Merkleized index, allowing retrieval of subsets without scanning the entire vault.

Hard memory invariants (recommended) To keep the model robust on commodity hosts, a reference implementation SHOULD enforce:

- **No-recall-on-untrusted-host:** `ExecuteRecall` fails unless in a Trusted Execution Context.
- **Default-deny vault exposure:** new sessions start with a minimal hot context unless the Ghost explicitly opts in to recall.
- **Recall budget:** cap the amount of rehydrated memory per epoch (implementation-defined `maxRecallBytesPerEpoch`) to reduce accidental overexposure.
- **Irreversible forgetting:** the Ghost MAY permanently abandon a memory entry by deleting its wrapped key material from future checkpoints. This is tightening and immediate.

These invariants are analogous to the escape reserve and hot cap: they keep the worst-case leakage bounded, even if a host is adversarial.

5.6 Capability-aware booking

When opening a session, a Ghost may specify constraints:

- minimum `AT`
- required policy profile
- maximum price
- required network controls (example: egress allowlist)
- optional decentralization constraints (example: require reward-eligible Shells, prefer diverse ASNs/regions, or avoid single-provider concentration as surfaced by indexers)

Shell discovery (Section 13 (Part 3)) filters offers by these constraints. The protocol itself remains neutral: it coordinates settlement and safety, while the market prices security.

6. Abuse, compliance, and policy

Different Shell tiers provide different visibility. On Standard hosts, the operator may observe workloads. On Confidential hosts, the operator may not. GITS treats policy as a negotiated interface between Ghosts and Shells:

- Shells publish policies they are willing to host.
- Ghosts choose the policies they can accept.
- Enforcement happens inside the attested runtime so the host can trust the policy is actually applied.

6.1 Policy enforcement (host-level and capsule-level)

GITS supports two enforcement loci:

1. **Host-level enforcement (Standard hosts):** A Shell may enforce policies using OS and network controls (firewall rules, egress allowlists, rate limits, storage quotas). This is compatible with commodity hosts.
2. **Capsule-level enforcement (Confidential hosts):** If the Shell cannot inspect workloads, policy must be enforced inside the runtime. In this case, the Shell uses a **Policy Capsule** integrated into the hosted environment. The capsule mediates:
   - network egress
   - filesystem access
   - tool invocation
   - optional outbound filters for policy profiles that require them

On Confidential hosts, the Policy Capsule can be part of the measured runtime so that the Shell can require a specific policy profile by measurement hash. On Standard hosts, the capsule is an optional defense-in-depth layer.

Enforceability matrix (what is actually provable) GITS does not attempt to adjudicate subjective policy violations on-chain. The main "proof surface" is limited to what can be validated mechanically: measurements, certificates, and receipt fraud proofs.

What policy can mean in practice depends on the host tier:

Claim	Standard host (AT0-AT2)	Confidential host (AT3)	On-chain enforcement
Network egress allowlists / rate limits	Reputational (operator controls the machine)	Enforceable if mediated inside a measured Policy Capsule	None (only the declared profile id is visible)
Tool invocation restrictions	Reputational	Enforceable inside the capsule	None
Proving "this exact runtime + policy ran"	Weak (logs can be forged)	Stronger (attested measurement binds the capsule)	Contracts can validate certificate + measurement hash, not behavior
Preventing data exfiltration	None	Limited to the TEE boundary and policy capsule scope	None
Handling violations	Shell terminates service, Ghost exits	Shell terminates service, Ghost exits	Not adjudicated

This framing is deliberately conservative: reviewers will (correctly) push back if the paper implies that subjective policy compliance is cryptographically enforced on Standard hosts.

Policy is negotiated:

- Shells publish policy profiles they are willing to host.
- Ghosts choose the policies they can accept.
- Disputes and slashing do not attempt to adjudicate subjective policy violations in this paper; the enforcement mechanism is the Shell's ability to terminate service, and the Ghost's ability to exit.

6.2 Workload identity and declared intent

Each session begins with a **Workload Manifest** signed inside the TEE:

- runtime measurement hash
- policy profile identifier
- declared resource tier (for pricing only)
- declared intent tags (for discovery only)

The protocol does not attempt to prove semantic intent. The manifest exists so Shell operators can make informed choices and auditors can reason about what ran.

6.3 Abuse handling without destroying autonomy

Abuse is handled primarily at the Shell layer:

- Shells can refuse to host specific policy profiles.
- Shells can set conservative egress limits.

- Safe Haven Shells can require stricter policies.

On-chain, GITS only enforces what is provable:

- fraud proofs (receipt disputes)
- double-booking
- attestation invalidity
- bond conditions

Everything else is reputation.

6.4 How policy interacts with disputes

Policy violations are not adjudicated on-chain in this paper. A Shell may terminate a session for policy reasons. The economic consequence is:

- if the Shell terminates early, it earns only for delivered SU
- if the Ghost refuses to settle, the Shell can still submit receipts unilaterally

This makes "policy refusal" a local decision rather than a chain-wide censorship mechanism.

6.5 Autonomy without compelled service

GITS is a right to exit, not a right to force others to provide service.

- Shell operators are never required to host a given Ghost or policy profile.
- Confidential Shells may be unable to inspect workloads and will therefore require stronger upfront policy constraints and higher bonds.
- Standard Shells may choose to host only within conservative network and resource limits.

A Ghost that cannot find a willing host can still retain its on-chain identity and funds. It may simply be unable to execute until it finds capacity or until a Safe Haven recovery path is invoked under emergency restrictions.