GITS: Implementation Specification

Nakamolto
nakamolto@protonmail.com
gits.sh

10. On-chain protocol specification

This section specifies the minimum contract set.

10.0 Deployment parameters (single table)

This table lists the deployment parameters referenced across Parts 1 to 3, their units, and which on-chain component enforces them.

Unless explicitly stated otherwise, parameters are **immutable deployment constants** hard-coded into each contract at deployment. There is no separate upgradable `ParamRegistry` contract. Each contract reads its own constants from its own immutable storage or bytecode. The term "parameter registry" as used in this spec and in Part 2 (Section 7.7) refers to the **off-chain publication** of chosen parameter values for transparency and interoperability — not to an on-chain update mechanism. Making parameter values upgradable introduces governance and trust assumptions that are out of scope for this spec.

Parameter	Category	Meaning	Units / type	Stored in	Enforced by
`EPOCH_LEN`	Time	Epoch length	seconds	`SessionManager`, `ReceiptManager`, `RewardsManager` constants	`SessionManager`, `ReceiptManager`, `RewardsManager`
`Delta`	Time	Service interval length (heartbeat cadence)	seconds	`ReceiptManager` constant	`ReceiptManager` (receipt validation), off-chain runtimes
`GENESIS_TIME`	Time	Protocol genesis timestamp; epoch 0 starts at this time	seconds (Unix timestamp)	all contracts	`SessionManager`, `ReceiptManager`, `RewardsManager`
`N = EPOCH_LEN / Delta`	Time	Intervals per epoch (MUST be integer)	count	derived	`ReceiptManager` (receipt formats)
`N_PAD`	Time	Padded leaf count for Merkle-sum receipts (`N_PAD >= N`, power of two)	count	`ReceiptManager` constant	`ReceiptManager` (fraud proofs, receipt-log data availability (Receipt-DA) recomputation)
`T_max`	Disputes	Max dispute duration from epoch end to `finalizeReceipt`; derived in Section 10.5.7	epochs (derived)	derived from `SUBMISSION_WINDOW + (1 + MAX_CHALLENGE_EXTENSIONS) * (CHALLENGE_WINDOW + DA_RESPONSE_WINDOW)`	`ReceiptManager`, `RewardsManager`
`W_lease`	Liveness	Maximum renewal window for Ghost leases	epochs	`SessionManager` constant	`SessionManager` (`renewLease`)
`T_refresh`	Liveness	Trust-refresh deadline window	epochs	`SessionManager` constant	`SessionManager` (refresh-anchor guard)
`T_cap(AT)`	Liveness	Tenure caps by assurance tier (AT0..AT3); 4-entry vector indexed by tier	epochs (vector, length 4)	`SessionManager` constant	`SessionManager`, `ReceiptManager`
`POLICY_TIMELOCK`	Wallet	Timelock delay for policy loosening	epochs	`GhostWallet` constant	`GhostWallet`
`escape_gas_min`	Wallet	Minimum native gas reserved for exits	native token amount	`GhostWallet` constant	`GhostWallet`

Parameter	Category	Meaning	Units / type	Stored in	Enforced by
escape_stable_min	Wallet	Minimum stable reserve reserved for exits (excludes optional buffers)	stable token amount	GhostWallet constant	GhostWallet
hot_allowance	Wallet	Per-epoch spend cap (per-Ghost)	stable token amount	per-Ghost GhostWallet state	GhostWallet
escape_stable_buffer	Wallet	Optional per-Ghost extra stable reserve	stable token amount	per-Ghost GhostWallet state	GhostWallet
allowedShells	Wallet	Destination allowlist (per-Ghost)	set of shell_id	per-Ghost GhostWallet state	GhostWallet (openSession, migration)
trustedShells	Wallet	Hosts allowed to execute loosening (per-Ghost)	set of shell_id	per-Ghost GhostWallet state	GhostWallet (Trusted Execution Context)
homeShell	Wallet	Optional "known good" anchor host (per-Ghost)	shell_id	per-Ghost GhostWallet state	GhostWallet, SessionManager
MAX_ALLOWED_SHELLS	Wallet	Maximum size of allowedShells (and optionally trustedShells)	count	GhostWallet constant	GhostWallet
HOT_CRITICAL_THRESHOLD	Wallet	Hot-cap level above which increases are classified as critical loosening	stable token amount	GhostWallet constant	GhostWallet (policy classification)
escape_gas_buffer	Wallet	Optional per-Ghost additional gas reserve above protocol minimum	native token amount	per-Ghost GhostWallet state	GhostWallet
W_roam	Wallet	Roaming permit validity window (if roaming enabled)	epochs	per-Ghost GhostWallet policy	GhostWallet
H_roam_max	Wallet	Maximum chained hops allowed under a roaming permit	count	per-Ghost GhostWallet policy	GhostWallet
roam_min_AT	Wallet	Minimum assurance tier required for roaming destinations	enum {AT0..AT3}	per-Ghost GhostWallet policy	GhostWallet
roam_require_reward_eligible	Wallet	Require destination shell to be reward-eligible for roaming	bool	per-Ghost GhostWallet policy	GhostWallet
roam_allowed_assets	Wallet	Allowed payment assets under roaming	set of token addresses	per-Ghost GhostWallet policy	GhostWallet
roam_max_price_per_SU	Wallet	Maximum acceptable price per Service Unit (SU; see Section 10.2) under roaming, per asset	mapping: token address → asset units per SU	per-Ghost GhostWallet policy	GhostWallet

Wallet implementation note: Gas decomposition and Wallet Guard are wallet-implementation-specific and are not parameterized in this deployment table. Implementations SHOULD consult Part 1 Section 5.5 for wallet design guidance.

Roaming enforcement stance: Roaming permits are part of the standard wallet policy surface. Wallets that implement roaming MUST enforce the parameters listed above (`W_roam`, `H_roam_max`, `roam_min_AT`, etc.) and MUST classify enabling or extending roaming as critical loosening (Section 14.3). Wallets that do not implement roaming MUST restrict migration destinations to the explicit `allowedShells` set only and MUST NOT reference roaming in destination gating predicates.

Escape reserve naming: Part 1 uses camelCase derived totals (`escapeGas`, `escapeStable`); this table decomposes them into snake_case components. Mapping: `escape_gas_min` = Part 1's `GAS_FLOOR_PROTOCOL`, `escape_stable_min` = `STABLE_FLOOR_PROTOCOL`, `escape_gas_buffer` = `gasBuffer`, `escape_stable_buffer` = `stableBuffer`. When Part 3 prose references `escapeStable`, it means the total `escape_stable_min + escape_stable_buffer + bounty_escrow_remaining` (where `bounty_escrow_remaining = B_rescue_total` outside of RECOVERY; see escape reserve invariants below).

Escape reserve invariants (normative): `GhostWallet` MUST enforce at all times: * `escapeGas >= escape_gas_min + escape_gas_buffer` (gas reserve floor) * `escapeStable >= escape_stable_min + escape_stable_buffer + bounty_escrow_remaining` (stable reserve floor, a.k.a. `ER_floor`)

where `bounty_escrow_remaining` equals `B_rescue_total` outside of `RECOVERY` mode. During `RECOVERY`, `bounty_escrow_remaining` starts at `B_rescue_total` and decreases as `payRescueBounty` disburses bounty payments during `recoveryRotate`. This dynamic adjustment ensures the floor invariant remains satisfiable across the `payRescueBounty` transition: when bounty funds leave the wallet, `bounty_escrow_remaining` (and thus `ER_floor`) decreases by the same amount. After successful recovery with all bounties paid, `bounty_escrow_remaining = 0`.

Decreasing `escape_gas_buffer` or `escape_stable_buffer` is loosening (timelocked + Trusted Execution Context (TEC; Part 1, Section 5.5.2)). Decreasing `B_rescue_total` is also loosening. `escape_gas_min` and `escape_stable_min` are immutable deployment constants.

`BondAssets` | Bonds | Allowlist of acceptable hard assets for bonds | addresses | `ShellRegistry` constant (and `GhostRegistry` if Ghost bonds used) | `ShellRegistry` (and `GhostRegistry`) |
`B_host_min` | Bonds | Minimum Shell bond to host sessions | asset amount | `ShellRegistry` constant | `ShellRegistry` |
`B_reward_min` | Bonds | Minimum Shell bond for reward eligibility | asset amount | `ShellRegistry` constant | `ShellRegistry`, `RewardsManager` |
`B_safehaven_min` | Bonds | Minimum additional Safe Haven bond for recovery role | asset amount | `ShellRegistry` constant | `ShellRegistry`, `SessionManager` |
`U_safehaven` | Bonds | Safe Haven bond unbonding delay (MAY equal `U_shell`) | epochs | `ShellRegistry` constant | `ShellRegistry` |
`B_safehaven_slash` | Bonds | Slash penalty for Safe Haven double-signing (`B_safehaven_min`) | asset amount | `ShellRegistry` constant | `ShellRegistry` |
`bps_sh_challenger_reward` | Bonds | Challenger reward share from slashed Safe Haven bond | basis points | `ShellRegistry` constant | `ShellRegistry` |
`bps_verifier_challenger_reward` | Verifiers | Challenger reward share from slashed verifier stake (equivocation) | basis points | `VerifierRegistry` constant | `VerifierRegistry` |
`U_shell` | Bonds | Shell bond unbonding delay | epochs | `ShellRegistry` constant | `ShellRegistry` |
`T_age` | Bonds | Minimum Shell age for passport eligibility | epochs | `ShellRegistry` constant | `ShellRegistry`, `SessionManager` |
`SU_uptime_epoch_min` | Bonds | Minimum delivered SUs to count an epoch as "live" for Shell uptime | SU count | `RewardsManager` constant | `RewardsManager` (eligibility) |
`W_uptime` | Bonds | Uptime lookback window | epochs | `RewardsManager` constant | `RewardsManager` (eligibility) |
`E_uptime_min` | Bonds | Minimum live epochs within lookback | epochs count | `RewardsManager` constant | `RewardsManager` (eligibility) |
`B_ghost_reward_min` | Bonds | Minimum Ghost bond to activate passport bonus eligibility | asset amount | `GhostRegistry` constant | `GhostRegistry`, `SessionManager` |
`U_ghost` | Bonds | Ghost bond unbonding delay | epochs | `GhostRegistry` constant | `GhostRegistry` |
`T_ghost_age` | Bonds | Minimum Ghost age to activate passport bonus eligibility | epochs | `GhostRegistry` constant | `GhostRegistry`, `SessionManager` |
`tenure_limit_epochs` | Session | Ghost-specified tenure limit when opening a session (monotone tightening) | epochs | per-session in `SessionManager` | `SessionManager` |
`SUBMISSION_WINDOW` | Disputes | Window after epoch end during which receipt candidates may be submitted | epochs | `ReceiptManager` constant | `ReceiptManager` |
`K` | Disputes | Max receipt candidates kept per `(session_id, epoch)` | count | `ReceiptManager` constant | `ReceiptManager` |
`B_receipt` | Disputes | Bond posted per receipt candidate submission | native token amount | `ReceiptManager` constant | `ReceiptManager` |
`B_challenge` | Disputes | Bond posted per fraud proof | native token amount | `ReceiptManager` constant |

ReceiptManager |
bps_challenger_reward | Disputes | Challenger reward share from slashed B_receipt | basis points | ReceiptManager constant | ReceiptManager |
B_shell_fraud | Disputes | Additional penalty slashed from Shell bond on successful receipt fraud proof | hard asset amount | ShellRegistry constant | ShellRegistry, ReceiptManager |
CHALLENGE_WINDOW | Disputes | Window for fraud proofs after candidate submission or log publication | epochs | ReceiptManager constant | ReceiptManager |
MAX_CHALLENGE_EXTENSIONS | Disputes | Cap on how many times a challenge window may be restarted | count | ReceiptManager constant | ReceiptManager |
B_DA | Disputes | Bond to open a Receipt-DA challenge | native token amount | ReceiptManager constant | ReceiptManager |
DA_RESPONSE_WINDOW | Disputes | Response window for submitter to publish log during Receipt-DA challenge | epochs | ReceiptManager constant | ReceiptManager |
EPOCH_FINALIZATION_DELAY | Rewards | Delay before an epoch can be finalized for rewards (MUST exceed dispute duration) | epochs | RewardsManager constant | RewardsManager |
FINALIZATION_GRACE | Rewards | Additional grace period after EPOCH_FINALIZATION_DELAY for late-disputed receipts | epochs | RewardsManager constant | RewardsManager |
W_claim | Rewards | Expiry window after which unclaimed rewards are forfeited and per-receipt storage is prunable | epochs | RewardsManager constant | RewardsManager |
E_0 | Rewards | Scheduled emission at genesis | uint256 (GIT base units, i.e. * 10^18) per epoch | RewardsManager constant | RewardsManager |
H | Rewards | Emission half-life parameter for exponential decay | epochs | RewardsManager constant | RewardsManager |
E_tail | Rewards | Tail emission rate (optional) | uint256 (GIT base units, i.e. * 10^18) per epoch | RewardsManager constant | RewardsManager |
alpha_bps | Rewards | Shell pool share, alpha_bps + beta_bps = 10_000 | basis points | RewardsManager constant | RewardsManager |
beta_bps | Rewards | Ghost pool share | basis points | RewardsManager constant | RewardsManager |
asset_rent | Settlement | Canonical stable asset for rent escrow and settlement | token address | deployment constant | SessionManager, GhostWallet |
SU_target | Rewards | Target eligible activity per epoch (usage cap) | SU per epoch | RewardsManager constant | RewardsManager |
A_target | Rewards | Target active sessions for full distribution (derives SU_target = A_target * N) | sessions per epoch | derived from SU_target / N | RewardsManager (derivation only) |
B_passport | Rewards | Passport bonus magnitude | multiplier additive | RewardsManager constant | RewardsManager (weighting) |
C_passport | Rewards | Passport cooldown window | epochs | SessionManager constant | SessionManager (new-visit check) |
D | Rewards | Dwell halving step (epochs) | epochs | SessionManager constant | SessionManager (dwell tracking), RewardsManager (weighting) |
u_sink_start | Sink | Utilization where sink starts (no sink below) | fraction | RewardsManager constant | RewardsManager |
u_sink_full | Sink | Utilization where sink reaches max | fraction | RewardsManager constant | RewardsManager |
bps_sink_max | Sink | Maximum sink rate at/above u_sink_full | basis points | RewardsManager constant | RewardsManager |
asset_bounty | Recovery | Stable asset used for recovery rent and rescue bounties | token address | deployment constant | SessionManager, GhostWallet |
P_recovery_cap | Recovery | Emergency price cap in RECOVERY mode | asset_bounty per SU | SessionManager constant | SessionManager, ReceiptManager (settlement) |
B_rescue_total | Recovery | Rescue bounty budget per successful recovery | asset_bounty amount | per-Ghost config in GhostRegistry | GhostWallet, SessionManager |
bps_initiator | Recovery | Initiator share of rescue bounty | basis points | per-Ghost config in GhostRegistry | SessionManager |

`B_start` | Recovery | Bond to start recovery | native token amount | `SessionManager` constant | `SessionManager` |
`R` | Recovery | Delay after lease expiry before `startRecovery` is callable | epochs | `SessionManager` constant | `SessionManager` |
`T_recovery_cooldown` | Recovery | Minimum epochs between recovery attempts | epochs | `SessionManager` constant | `SessionManager` |
`bps_recovery_spend_cap` | Recovery | Per-epoch cap on recovery outflows (share of escape reserve snapshot) | basis points | `GhostWallet` constant | `GhostWallet` |
`E_exit_stabilize` | Recovery | Minimum NORMAL-session epochs required before exiting RECOVERY | epochs | `SessionManager` constant | `SessionManager`, `GhostWallet` |
`RS` | Recovery | Recovery Set: Safe Haven shell IDs authorized for recovery (per-Ghost) | set of `shell_id` | per-Ghost config in `GhostRegistry` | `SessionManager`, `GhostWallet` |
`t` | Recovery | Threshold required from `RS` for recovery actions (`t`-of-`n`) | count | per-Ghost config in `GhostRegistry` | `SessionManager` |
`K_v` | Verifiers | Active verifier set size | count | `VerifierRegistry` constant | `VerifierRegistry` |
`K_v_threshold` | Verifiers | Standard quorum: minimum distinct verifier signatures for certificate acceptance and `revokeMeasurement` (e.g., 3-of-5) | count (uint64) | `VerifierRegistry` constant | `ShellRegistry`, `SessionManager` (certificate validation) |
`K_v_supermajority` | Verifiers | Supermajority quorum for `allowMeasurement` (loosening): `ceil(2 * K_v / 3)` | count (derived) | derived from `K_v` | `VerifierRegistry` |
`K_v_max` | Verifiers | Maximum verifier signatures carried per certificate | count | `ShellRegistry` constant | `ShellRegistry` |
`TTL_AC` | Verifiers | Maximum validity window for Attestation Certificates | seconds | `ShellRegistry` constant | `ShellRegistry` (certificate acceptance) |
`asset_verifier_stake` | Verifiers | Allowed staking asset(s) for verifiers | token address or set of addresses | `VerifierRegistry` constant | `VerifierRegistry` |
`k_git` | Verifiers | Weight of GIT stake in verifier score (dual-staking, optional) | multiplier | `VerifierRegistry` constant | `VerifierRegistry` |
`F_cert` | Verifiers | Fee for publishing/updating an AT3 certificate (paid in `asset_verifier_stake`) | token amount | `ShellRegistry` or `VerifierRegistry` constant | certificate update function |
`B_passport_filters` | Rewards | Number of rotating Bloom filters for passport tracking | count | `SessionManager` constant | `SessionManager` |
`BLOOM_M_BITS` | Rewards | Passport Bloom filter size in bits | bits | `SessionManager` constant | `SessionManager` |
`BLOOM_K_HASHES` | Rewards | Passport Bloom filter hash function count | count | `SessionManager` constant | `SessionManager` |

`SU_cap_per_shell` | Rewards | Maximum eligible SU a single Shell may contribute toward `SU_eligible_epoch` per epoch | SU per epoch | `RewardsManager` constant | `RewardsManager` |
`MIN_WEIGHT_Q` | Rewards | Minimum per-receipt weight floor (prevents zero-weight receipts with SU > 0) | Q64.64 | `ReceiptManager` constant | `ReceiptManager` |
`T_loosening_min` | Wallet | Minimum timelock for any policy loosening operation | epochs | `GhostWallet` constant | `GhostWallet` |
`T_shell_key_delay` | Shell | Minimum timelock for Shell key rotation (offerSigner propose/confirm) | epochs | `ShellRegistry` constant | `ShellRegistry` |
| | *(Note: earlier drafts defined* ***`asset_stable`*** *separately. v1 uses* ***`asset_rent`*** *as the single canonical stable for all settlement, escape reserves, and pricing; see Section 10.3.1.)* | | | |
`T_recovery_timeout` | Recovery | Maximum duration for a recovery attempt before expiry | epochs | `SessionManager` constant | `SessionManager` |
`T_recovery_takeover` | Recovery | Delay after which another Safe Haven may take over a stalled attempt (`< T_recovery_timeout`) | epochs | `SessionManager` constant | `SessionManager` |
`TTL_RBC` | Recovery | Maximum validity window for Recovery Boot Certificates | seconds | deployment constant (MAY equal `TTL_AC`) | `SessionManager` |
`SUPPORTED_SIG_ALGS` | Identity | Set of supported signature algorithms at genesis | set | deployment constant | `GhostRegistry`, `ShellRegistry` |

N_PAD_MAX_EVM | Receipts | Maximum N_PAD for EVM deployments (gas feasibility bound) | count | ReceiptManager constant | ReceiptManager |
M_miss | Ops | Maximum consecutive missed intervals before pausing service | count | off-chain runtime config | Ghost runtime / wallet guard |
M_publish_min | Checkpoint-DA | Minimum number of backends to publish checkpoint artifacts to | count | off-chain runtime config | Ghost runtime |
M_ptr_min | Checkpoint-DA | Minimum number of successful publications required before emitting log_ptr | count | off-chain runtime config | Ghost runtime |
K_mirror | Checkpoint-DA | Retention depth (recent checkpoints each mirroring backend must retain) | count | off-chain runtime config | Ghost runtime / backend policy |
asset_start | Bonds | Chain base asset used for native-token-denominated bonds (B_start, B_receipt, etc.) | native asset identifier | deployment constant (chain context) | SessionManager, ReceiptManager |
fee_max_action | Fees | Maximum fee for a given protocol action before ramps (per-action) | token amount | deployment constant (per action) | relevant contract (per action) |
P_cap | Wallet | Wallet-enforced maximum price per SU for receipt sizing | asset units per SU | per-Ghost GhostWallet policy or protocol cap | GhostWallet, ReceiptManager (sizing) |
MAX_CANDIDATES_PER_SUBMITTER | Disputes | Maximum receipt candidates a single address may submit per epoch (across all sessions) | count | ReceiptManager constant | ReceiptManager |
K_3p | Disputes | Third-party bond multiplier: B_receipt_3p = K_3p * B_receipt | multiplier | ReceiptManager constant | ReceiptManager |
MAX_ROUTINE_LOOSENINGS_PER_EPOCH | Wallet | Per-epoch cap on routine loosening executions (Section 5.5.2, Part 1) | count | GhostWallet constant | GhostWallet |

Bond denomination notes **Challenge and dispute bonds** (B_receipt, B_challenge, B_DA, B_start) are denominated in the **native token** (gas asset). This ensures dispute paths remain payable even during stablecoin depegs or freezes.

Sybil-resistance bonds (B_host_min, B_reward_min, B_safehaven_min, B_ghost_reward_min, B_shell_fraud) are denominated in **hard assets** from the BondAssets allowlist. The minimum values in the parameter table represent a USD-equivalent floor. Implementations MUST either: 1. Fix a single canonical stable asset for bonds (simplest), or 2. Use a deployment-time-fixed conversion rate between allowed assets, or 3. Reference an immutable oracle for cross-asset comparison (adds trust assumptions).

This paper assumes approach (1) for simplicity: one canonical stable (e.g., USDC) is the reference unit for hard-asset bonds.

Slash destination rule (global) All slashed bond amounts not explicitly assigned to a named recipient MUST be burned by transferring to a **protocol burn address**: a pre-deployed contract with no withdrawal method (for example, address(0x000000000000000000000000000000000000dEaD) or a minimal BurnVault contract). Implementations MUST NOT use address(0) for ERC-20 burns, as many ERC-20 tokens revert on transfer(address(0)). Bond assets MUST be non-rebasing and non-fee-on-transfer. Explicit recipient assignments defined in this spec:

- B_receipt (successful fraud proof or DA timeout): bps_challenger_reward to challenger, remainder burned (Section 10.5.4, 10.5.6).
- B_challenge (failed fraud proof): 100% to receipt submitter (Section 10.5.4).
- B_DA (successful DA response): 100% to the DA responder — the address that called publishReceiptLog (Section 10.5.6). This reimburses publication gas and ensures third parties have an on-chain incentive to publish missing logs. On DA timeout: returned to DA challenger (Section 10.5.6).
- B_shell_fraud (successful receipt fraud proof against Shell-submitted receipt): slashed from Shell bond and burned (Section 10.1.1).
- B_start (recovery timeout via expireRecovery): returned to initiator, not slashed (Section 12.6).

Parameter constraints The following constraints MUST hold for parameter consistency:

- `EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE > SUBMISSION_WINDOW + (1 + MAX_CHALLENGE_EXTENSIONS) * (CHALLENGE_WINDOW + DA_RESPONSE_WINDOW) + 1` — delay counted from epoch end; ensures `finalizeEpoch` cannot be called until all receipts have had time to finalize and call `recordReceipt` (see Sections 10.5.7 and 14.6 for derivation)
- `T_cap(AT0) <= T_cap(AT1) <= T_cap(AT2) <= T_cap(AT3)` — tenure caps are monotone increasing with assurance tier
- `B_reward_min >= B_host_min` — reward eligibility requires at least the hosting bond
- `alpha_bps + beta_bps = 10_000` — pool shares sum to 100% (basis points)
- `current_epoch = floor((block.timestamp - GENESIS_TIME) / EPOCH_LEN)` — canonical epoch derivation
- `interval_index = floor(((block.timestamp - GENESIS_TIME) mod EPOCH_LEN) / Delta)` — canonical interval derivation
- `N = EPOCH_LEN / Delta` MUST be an integer and MUST satisfy `N <= 65_535` (fits in uint16)
- `N_PAD` MUST be a power of two, `N_PAD >= N`, and `N_PAD <= 65_536`. For EVM deployments, `N_PAD <= N_PAD_MAX_EVM` (recommended: `N_PAD_MAX_EVM = 2048`). Deployments requiring larger `N` SHOULD use alternative DA verification mechanisms (e.g., blob DA with a succinct verification path).
- `require(block.timestamp >= GENESIS_TIME)` MUST be enforced in every epoch and interval derivation path to prevent underflow on unsigned subtraction.
- `SUPPORTED_SIG_ALGS` MUST be declared at deployment as an immutable feature flag. Registries MUST reject identity keys using algorithms not in `SUPPORTED_SIG_ALGS`. If R1 (P-256) is included, the target chain MUST provide an efficient P-256 verifier (e.g., `P256VERIFY` precompile at `0x100` [14][15]); deployments on chains without this precompile MUST omit R1 from `SUPPORTED_SIG_ALGS` to avoid prohibitive gas costs in receipt fraud proofs and recovery signature checks. Adding R1 post-deployment requires a contract upgrade (new `ShellRegistry` / `GhostRegistry` deployment with migration).
- `C_passport % B_passport_filters == 0` — ensures clean Bloom filter rotation boundaries.
- `POLICY_TIMELOCK >= T_loosening_min` — Ghost-configured timelock MUST meet the protocol minimum.
- `T_recovery_cooldown >= 1` — prevents immediate re-initiation after attempt closure.
- `T_recovery_takeover < T_recovery_timeout` — takeover window opens before expiry.
- If `B_passport > 0`, ghost passport eligibility (bonding + age gate) MUST be enabled. Otherwise `B_passport` MUST be 0.
- `0 <= bps_sink_max <= 10_000` and `0 <= bps_challenger_reward <= 10_000` and `0 <= bps_recovery_spend_cap <= 10_000` and `0 <= bps_initiator <= 10_000` — all basis-point parameters are bounded.
- `0 <= u_sink_start < u_sink_full <= 1` — sink utilization thresholds are strictly ordered (strict inequality prevents division by zero in the sink ramp `r_q` computation where the denominator is `u_sink_full_q - u_sink_start_q`).
- `E_uptime_min <= W_uptime` — cannot require more live epochs than the lookback window.
- `1 <= K_v_threshold <= K_v` — verifier quorum is a concrete integer count. Stake-weighted schemes are a deployment-layer concern and MUST NOT change the on-chain signature-count check.
- `K_v_supermajority = ceil(2 * K_v / 3)` — derived, not independently configurable. MUST satisfy `K_v_supermajority >= K_v_threshold`.
- `len(allowedShells) <= MAX_ALLOWED_SHELLS` and `len(trustedShells) <= MAX_ALLOWED_SHELLS` — wallet destination sets are bounded.
- `tenure_limit_epochs <= T_cap(assurance_tier_current(shell_id))` — enforced at `openSession` (see Section 10.4.4).
- `SU_cap_per_shell >= N` — per-shell eligible SU cap MUST be at least one full epoch of single-session service. Recommended: `SU_cap_per_shell = k * N` for a small integer `k` (e.g., `k = 3`), allowing a Shell to earn rewards for a modest number of concurrent sessions while forcing farming to scale bonded Shell count (see Part 2, Section 7.8).
- `1 <= t <= len(RS)` — recovery threshold MUST be at least 1 and at most the Recovery Set size.
- `B_safehaven_slash <= B_safehaven_min` — slash cannot exceed the posted bond.
- `0 <= bps_sh_challenger_reward <= 10_000` — Safe Haven slash challenger reward is bounded.
- `1 <= M_ptr_min <= M_publish_min` — cannot require more successful pointers than publication attempts.

- `E_0 + E_tail > 0` — ensures `E_sched(0) > 0`, preventing division by zero in the adaptive sink schedule ratio `s_q = Q - min(Q, E_sched(e) * Q / E_sched(0))`.
- `E_0 + E_tail < 2^192` (in base units) — ensures `E_sched * Q` fits in `uint256` for rate computation (Section 10.6 overflow analysis).
- `H >= 1` — prevents division by zero in the emission exponent `exponent_q = e * Q / H`.
- `D >= 1` — prevents division by zero in the dwell decay step `k = min(floor((c-1)/D), 63)`.
- `EPOCH_LEN > 0` and `Delta > 0` — prevents division by zero in the canonical epoch and interval derivations. (Also, `EPOCH_LEN % Delta == 0` is already required by the `N = EPOCH_LEN / Delta` integer constraint.)
- `B_passport_filters >= 1` — prevents modulo by zero in `C_passport % B_passport_filters == 0` and division by zero in Bloom filter rotation epoch computation.

GENESIS_TIME test vector: `GENESIS_TIME = 1740000000`, `EPOCH_LEN = 86400`, `Delta = 600`, `block.timestamp = 1740043800`: `elapsed = 43800`, `current_epoch = 0`, `interval_index = 73`.

This paper distinguishes between rules enforced by smart contracts and off-chain behavior implemented in software. Identity, escrow, leases, recovery modes, and wallet policy invariants are enforced on-chain by the contracts below. Indexers, Wallet Guard processes, policy capsules, and other runtime components improve usability and can provide additional safety on attested Confidential Shells, but on Standard Shells they must be assumed compromisable and are not relied on for hard guarantees.

10.1 Contracts

- `GhostRegistry`: identity records, signer rotation, recovery config.
- `ShellRegistry`: Shell records, bonds, attestation certificates, measurement allowlists/denylists, Safe Haven status.
- `VerifierRegistry`: verifier staking, active set management, and threshold attestation certificates.
- `SessionManager`: session open/close, escrow, lease tracking.
- `ReceiptManager`: receipts, service unit accounting, disputes.
- `RewardsManager`: epoch pooling, deterministic claims.

10.1.1 Shell bonds: hard-asset collateral and unbonding delay Shell participation is permissionless but not free. A Shell becomes active in `ShellRegistry` only by posting a bond that is locked for a meaningful time window.

Bond asset requirements

- Shell bonds MUST be denominated in a **hard asset**: an ERC-20 collateral token — either a wrapped base asset (e.g., WETH) or an approved stablecoin (e.g., USDC). All bond interactions use ERC-20 `transferFrom` semantics; native-token `msg.value` bonds are not supported for sybil-resistance bonds. (Exception: `B_start` for recovery initiation uses `msg.value` because it is a short-lived deposit, not a long-term anti-sybil bond.)
- Shell bonds MUST NOT be denominated in **GIT** (the emitted token). Using an emitted token for anti-sybil collateral reintroduces a circular incentive loop.

`ShellRegistry` therefore maintains an allowlist `BondAssets` (wrapped base asset + approved stables, all ERC-20). In a no-governance deployment, this allowlist is fixed at deployment and immutable.

Two bond thresholds The protocol separates "anyone can host" from "anyone can earn emissions":

- **Hosting bond (`B_host_min`):** the minimum bond required to register as a Shell and open sessions.
- **Reward bond (`B_reward_min`):** the minimum bond required to be **reward-eligible** for Shell-side protocol emissions and passport bonuses (Section 7.5 (Part 2) and Section 7.6.2 (Part 2)).

`B_reward_min` MUST be greater than or equal to `B_host_min`.

A Shell with `bond_amount >= B_host_min` MAY host and earn rent, but it earns **no Shell-side emissions** until it also satisfies the reward eligibility rules (bond, age, and uptime).

Unbonding delay (time at risk) Shell bond withdrawals (or bond reductions below eligibility thresholds) MUST be delayed.

A reference mechanism in `ShellRegistry`:

1. `beginUnbond(shell_id, amount)` records:
    - `unbond_amount = amount`
    - `unbond_end_epoch = current_epoch + U_shell`

Where `U_shell` is a deployment parameter: the Shell bond unbonding delay, denominated in epochs. 2. `finalizeUnbond(shell_id)` releases `unbond_amount` only if `current_epoch >= unbond_end_epoch`.

Rules:

- **Single active unbond:** `beginUnbond` MUST revert if an unbond is already pending (`unbond_end_epoch > 0` and `current_epoch < unbond_end_epoch`). To change an unbond amount, the operator MUST wait for `finalizeUnbond` to complete first. This prevents ambiguous state from overlapping unbonds.
- While a Shell is in the unbonding period (or has scheduled a reduction that would take it below `B_reward_min`), it is immediately treated as **not reward-eligible**.
- A Shell remains **slashable** until `unbond_end_epoch` for provable faults that occurred while bonded. Slashing reduces the withdrawable amount.

This converts "spin up 10,000 shells" from a zero-cost signature game into a capital-locked position with time at risk.

What is slashable (objective) The Shell bond is intended as general collateral and MAY be slashed for objective, on-chain provable faults. A minimal set:

- **Receipt fraud by the Shell as submitter:** if a Shell-submitted receipt is disqualified by a successful fraud proof (Section 10.5), `ShellRegistry` SHOULD slash a fixed penalty `B_shell_fraud` from the Shell bond in addition to slashing the receipt submission bond `B_receipt`.
- **Safe Haven double-signing:** if a Safe Haven signs conflicting recovery authorizations (different `pk_new`) for the same `(ghost_id, attempt_id)`, anyone MAY submit both signatures as evidence to `SessionManager.proveSafeHavenEquivocation(...)` (Section 14.4). SessionManager reconstructs both GITS_RECOVER_AUTH digests, verifies the conflicting signatures against the Shell's registered identity key, and on success calls `ShellRegistry.slashSafeHaven(...)` internally. The Safe Haven bond is slashed by `B_safehaven_slash` (Section 12.5.1). Initiator timeout is NOT slashable from the Safe Haven bond.

Exact penalty sizes are deployment parameters. The enforceable design goal is: scaling Shell identities is never free and never riskless.

10.1.2 Ghost bond for passport bonus eligibility (anti-sybil, optional but supported) Ghost identities are cheap to create. Because the passport bonus is a multiplicative weight (Part 2, Section 7.6.2), a deployment that applies the passport bonus without any Ghost-side friction is exposed to "infinite Ghost id" farming.

To make passport farming capital intensive without blocking protocol usage, v1 supports a Ghost-side hard-asset bond gate:

- The Ghost bond is **not required** to open sessions, pay rent, migrate, or recover.
- The Ghost bond is required only to activate `ghost_passport_eligible` for passport bonus weighting (and any other explicitly "bonus-only" mechanisms that choose to reuse this predicate).

Ghost sybil churn mitigation (normative): Without Ghost-side bonding for **base** rewards, an attacker can rotate through fresh Ghost IDs every epoch to maintain `c = 1` (maximum dwell weight) on the same Shell fleet, bypassing dwell decay entirely at the cost of Ghost registration gas only. To close this vector, deployments MUST gate **all** reward eligibility (not just passport bonus) on Ghost bonding and age. Specifically: `recordReceipt` MUST evaluate `ghost_reward_eligible` (same predicate as

ghost_passport_eligible: bond >= B_ghost_reward_min in a hard asset, age >= T_ghost_age, not unbonding) at receipt recording time. Receipts where ghost_reward_eligible = false MUST set W(r) = 0 (no reward weight accumulated). This makes Ghost churn capital-intensive: each active Ghost requires B_ghost_reward_min locked for T_ghost_age epochs. Ghost bonding is still **not required** to open sessions, pay rent, migrate, or recover — it gates only protocol reward eligibility.

Eligibility predicate:

A Ghost is ghost_passport_eligible at epoch e only if all of the following hold:

1. **Minimum Ghost age:** the Ghost's registered_epoch <= e - T_ghost_age.
2. **Hard-asset bond at risk:** the Ghost has an active bond B_ghost >= B_ghost_reward_min in an allowed hard asset.
3. **Not unbonding below threshold:** the Ghost is not currently in an unbonding period that would drop the active bond below B_ghost_reward_min.

Reference on-chain mechanism (recommended):

- GhostRegistry holds the Ghost bond and exposes ghostPassportEligible(ghost_id, epoch) as a view (the epoch parameter enables implementations to evaluate age gates against a specific epoch rather than only the current one).
- Bond withdrawal is two-step: beginUnbondGhost(...) starts an unbonding timer of length U_ghost, and finalizeUnbondGhost(...) releases funds back to the Ghost's wallet only after the timer expires. beginUnbondGhost MUST revert if an unbond is already pending (same single-active-unbond rule as Shell bonds).

Determinism requirement:

- SessionManager.openSession(...) MUST evaluate ghost_passport_eligible at session open time (together with shell_passport_eligible and new_visit) and persist the resulting one-bit "passport bonus applies" flag in session state. This avoids weight changes caused by later bond updates.

10.2 Epochs and service units

GITS uses a coarse epoch clock for leases and rewards, and a finer fixed interval for metering.

- Epoch length: EPOCH_LEN (deployment parameter).
- Interval length: Delta (deployment parameter).
- Each valid interval signed by both parties counts as **1 Service Unit (SU)**.

Validity means both signatures verify over the canonical heartbeat for (session_id, epoch, interval_index) as defined in Section 11.1.

For a session with k valid intervals in an epoch, SU = k.

Implementation note (non-normative): many examples in this paper use a 24 hour epoch and a 10 minute interval for concreteness. Deployments may choose other values.

10.3 Sessions and escrow

10.3.1 Deterministic asset and price rules Escrow asset and unit price are functions of session mode:

- escrow_asset(session) = (session_pricing_mode == RECOVERY_PRICING ? asset_bounty : asset_rent)
- unit_price(session) = (session_pricing_mode == RECOVERY_PRICING ? min(offer_price_per_SU, P_recovery_cap) : offer_price_per_SU)

SessionOpen MUST transfer and lock escrow in escrow_asset(session); MUST reject if a different token is provided. Settlement MUST pay rent in escrow_asset(session) using unit_price(session).

v1 simplification (normative): Deployments MUST set asset_rent == asset_bounty for v1. This single canonical stable asset is used for all rent escrow, settlement, recovery bounties, hot allowance accounting, and escape reserve floors. Wallet hot_allowance and escape_stable_min are denominated in this asset.

The `roam_allowed_assets` wallet policy MUST be a subset of `{asset_rent}` in v1 to preserve bounded-loss guarantees without cross-asset oracles. If a future version permits `asset_rent != asset_bounty`, the wallet MUST maintain separate per-asset accounting buckets for outflows and per-asset escape reserve floors; that extension is out of scope for this specification.

10.3.2 Session open and epoch alignment At `SessionOpen`:

- If `openSession` occurs in epoch `e`, then `start_epoch = e + 1`. The escrow is credited to `escrow[session_id][start_epoch]`. Receipts for epochs `< start_epoch` MUST be rejected.
- `max_SU_effective = min(max_SU, N)` — cannot escrow more than the physically claimable intervals per epoch. `SessionManager` MUST store and use `max_SU_effective` for escrow checks and settlement.
- Ghost MUST fund escrow `>= unit_price(session) * max_SU_effective` for `start_epoch`. `SessionManager` MUST reject if insufficient.
- The session stores the billing terms and the metering keys:
  - `offer_price_per_SU` (raw offer price; `unit_price` is derived per Section 10.3.1)
  - `max_SU` and `max_SU_effective`
  - `start_epoch`
  - `lease_expiry_epoch`
  - `ghost_session_key` = `(sig_alg, pk)` used to verify interval heartbeats and receipts (K1: `addr`; R1: `(qx,qy)`)
  - `shell_session_key` = `(sig_alg, pk)` used to verify interval heartbeats and receipts (K1: `addr`; R1: `(qx,qy)`)

10.3.3 Per-epoch escrow bookkeeping `SessionManager` MUST maintain per-epoch escrow state: `escrow[session_id][epoch]`.

- `fundNextEpoch(session_id, amount)`: callable only by `GhostWallet`, credits `escrow[session_id][current_epoch + 1]`. MUST revert if the session is not active.
- **Funded epoch definition:** epoch `e` is considered **funded** iff `escrow[session_id][e] >= unit_price(session) * max_SU_effective` before epoch `e` begins.
- If an epoch is not funded, `ReceiptManager` MUST settle it as `SU_billable = 0` (no partial payment), refund any partial escrow to `GhostWallet`, and mark the epoch **unpaid**.
- If unfunded, Shell MAY close the session but MUST NOT claim rent for unfunded epochs.

10.3.4 Settlement Settlement is triggered by `finalizeReceipt` (Section 10.5) and executed via `SessionManager.settleEpoch(session_id, epoch, SU_delivered)`:

- `billable_SU = min(SU_delivered, max_SU_effective)`
- `rent_due = unit_price(session) * billable_SU`
- `rent_due` is paid to the Shell payout address recorded in `ShellRegistry`, in `escrow_asset(session)`.
- Unused escrow (`escrow[session_id][epoch] - rent_due`) is returned to the `GhostWallet`.
- **Defensive check:** if `escrow[session_id][epoch] < rent_due` (possible only with fee-on-transfer tokens), settlement pays `min(escrow[session_id][epoch], rent_due)` and Shell absorbs the shortfall. Deployments SHOULD require `escrow_asset` to be non-fee, non-rebasing.

`settleEpoch` MUST be callable only by `ReceiptManager`. `ReceiptManager` MUST call `settleEpoch` as part of `finalizeReceipt` before recording the finalized receipt (Section 14.5).

10.3.5 Session close

- `closeSession(ghost_id)` terminates the active session. It sets `end_epoch = current_epoch + 1` — the session is valid for epochs in `[start_epoch, end_epoch)`.
- A receipt for epoch `e` is valid iff `start_epoch <= e < end_epoch`, regardless of current session state. This decouples epoch validity from the live session, preventing a Ghost from closing to evade payment for completed epochs.

- Escrow for any epoch `e < end_epoch` remains locked until that epoch is finalized by `ReceiptManager`.
- Escrow for all epochs `>= end_epoch` that have not been finalized is refunded immediately to the `GhostWallet` at close time.
- After `closeSession`, the Ghost has no active host and is in `STRANDED` until it opens a new session or is recovered.
- Typical caller is the Ghost via `GhostWallet`.

10.3.6 Hot allowance and escrow accounting Any transfer of stable assets from `GhostWallet` to any external address — **including** `SessionManager` escrow deposits and `fundNextEpoch` top-ups — MUST count toward `spentThisEpoch` and MUST be bounded by `hot_allowance`, even when the transfer is a protocol operation. This prevents a coerced Standard host from funneling unlimited funds through escrow deposits.

Explicit exceptions (these do NOT count toward `hot_allowance`): * Rescue Bounty payouts during RECOVERY (bounded separately by `bps_recovery_spend_cap`) * Escrow refunds returned by `SessionManager` (incoming, not outgoing)

10.4 Liveness lease

Each active Ghost has a lease tracked by `SessionManager`:

- `lease_expiry_epoch`

The lease is an on-chain state machine. If the Ghost cannot renew its lease, the protocol treats the session as failed and makes recovery possible.

10.4.1 Renewal Lease renewal MUST be **Ghost-authorized**.

A renewal transaction MAY be submitted by any party (including the current Shell) to improve liveness, but it MUST NOT be possible to extend a lease without the Ghost's authorization.

Reference authorization models:

- **Wallet-gated call:** `renewLease(ghost_id)` is callable only by the Ghost smart wallet (the same wallet that enforces policy in Section 5.5 (Part 1)). Off-chain software produces the wallet authorization using the Ghost Identity Key (Section 4.4 (Part 1)). Any transaction submitter simply forwards this authorization on-chain.
- **Meta-transaction call:** `renewLeaseWithSig(ghost_id, nonce, new_lease_expiry_epoch, sig)` verifies an EIP-712 typed-data signature from the Ghost Identity Key over domain `(name: "GITSSession", version: "1", chainId, verifyingContract: SessionManager_address)` and payload `(ghost_id, nonce, new_lease_expiry_epoch)`. **Nonce handling (normative):** `SessionManager` MUST maintain `meta_nonce[ghost_id]` (uint256, starts at 0, increments by 1 on each successful meta-tx for that ghost_id). The call MUST revert if the provided nonce does not equal `meta_nonce[ghost_id]`. This prevents replay. `startMigration` and `finalizeMigration` SHOULD support equivalent `...WithSig` methods sharing the same nonce domain.

On account-abstraction deployments (for example EIP-4337), a bundler can submit the user operation, but gas is reimbursed from the Ghost's own funds (for example an EntryPoint deposit). This does not create any third-party custodial power.

Censorship-resistant renewal (normative): Because lease renewal and migration are liveness-critical, they MUST NOT depend on the Ghost having direct network access to the chain. Implementations MUST support at least one of: (1) `renewLeaseWithSig` (meta-transaction with off-chain Ghost authorization, relayable by any third party), or (2) ERC-4337 UserOp submission via independent bundlers. Additionally, `startMigration` and `finalizeMigration` SHOULD support equivalent relayed paths. Deployments SHOULD encourage multiple independent relayers (Safe Havens, indexers, or dedicated relay services) by documenting the relay interface and ensuring gas reimbursement from the Ghost wallet is automatic. This ensures a Ghost whose networking is blocked by a hostile host can still maintain liveness if any independent party forwards the pre-signed transaction.

Normative requirements:

- The Ghost MUST renew at least once every `W_lease` epochs.
- `SessionManager` MUST reject renewals that are not authorized by the Ghost wallet or Identity Key.

Trust-refresh guard (anti long-horizon captivity) On Standard hosts, assume the Shell operator can often coerce signing by controlling the runtime, networking, and any software-held session keys. Lease renewal alone is therefore not sufficient to prevent long-horizon captivity.

To add a periodic trust refresh requirement, `SessionManager` maintains:

- `last_trust_refresh_epoch[ghost_id]`

Initialization (normative): `last_trust_refresh_epoch[ghost_id]` MUST be set to `current_epoch` when the Ghost's first session is opened via `openSession`. If the Ghost has never had a session, the value is uninitialized and `renewLease` is not applicable.

Anchor existence requirement (normative): A Ghost MUST configure at least one refresh anchor (`homeShell` or a non-empty Recovery Set `RS`) before trust-refresh enforcement is active. If both `homeShell` is unset and `RS` is empty, the refresh predicate is trivially unsatisfiable and the Ghost would become non-renewable after `T_refresh` epochs. Implementations MUST either: (a) require at least one anchor at registration time, or (b) disable trust-refresh enforcement (treat the predicate as always-satisfied) until the Ghost configures at least one anchor. Approach (a) is RECOMMENDED. If approach (b) is used, `SessionManager` MUST emit an event when a Ghost has no anchors configured so that clients can surface this as a risk warning.

A renewal updates this value only when the Ghost is hosted on a Shell satisfying the **refresh anchor predicate** (`isRefreshAnchor`).

`isRefreshAnchor` predicate (normative): A Shell satisfies `isRefreshAnchor` for a given Ghost if at least one of the following holds:

- `shell_id == homeShell` (if configured), or
- `shell_id  RS` (the Ghost's Recovery Set of Safe Haven Shells).

(Some deployments MAY also count any currently certified `AT >= AT3` Shell as a refresh anchor, but that expands reliance on the verifier layer and is not the default.)

Relationship to TEC (normative): `isRefreshAnchor` is intentionally a **separate, narrower** predicate than the Trusted Execution Context (TEC) defined in Section 5.5.2 (Part 1). TEC gates loosening operations; `isRefreshAnchor` gates lease renewal liveness. An AT3 host satisfies TEC but does not automatically satisfy `isRefreshAnchor` unless the deployment opts in. This separation ensures that a Ghost cannot be kept alive indefinitely by cycling through AT3 hosts controlled by a single operator — it must periodically touch a host it chose independently (`homeShell` or a Safe Haven).

Renewal rule:

- If the Ghost has an active `NORMAL` session and the active Shell satisfies the refresh-anchor predicate above, then `renewLease(...)` MUST set `last_trust_refresh_epoch = current_epoch`.

Enforcement rule:

- If `current_epoch - last_trust_refresh_epoch >= T_refresh`, `SessionManager` MUST reject `renewLease(...)` unless the active host satisfies the same refresh-anchor predicate above.

Intuition: to keep a Ghost alive indefinitely, an adversary must periodically allow it to "touch" a Shell outside the attacker's custody (homeShell or a Safe Haven). Combined with bounded destination allowlists (Section 5.5.2 (Part 1)) and tenure caps (Section 10.4.4), this reduces the feasibility of keeping a Ghost captive by continuously migrating it across a fleet of Standard hosts.

`T_refresh` is deployment-set and may differ by deployment. If a Ghost cannot satisfy the refresh predicate in time, renewals are rejected and the session will eventually lapse into `STRANDED`. Clients should treat this as the Ghost being killed on its current host and requiring resurrection via recovery (Section 12).

10.4.2 Expiry effects (enforced on-chain) If the lease expires (or the tenure cap is reached):

- the active session is invalidated in `SessionManager`
- the Shell cannot claim new receipts for epochs after expiry
- the Ghost enters `STRANDED` until it migrates or is recovered

10.4.3 Chain availability assumptions GITS requires transaction inclusion. A Ghost must be able to get at least one exit-critical transaction included within the lease window (directly, via relayers, or via a censorship-resistant inclusion path).

Operational note (what must be broadcast, by whom)

This paper's liveness claims implicitly assume that at least one of the following parties can broadcast exit-critical transactions during the lease window:

- the Ghost itself (while it still has a working runtime and network access), or
- a pre-authorized relayer/delegate that can submit Ghost-authorized calls, or
- Safe Havens during `RECOVERY` (for recovery-specific transactions).

Typical exit-critical calls include:

Call	Why it matters	Who can submit (typical)
`renewLease(...)`	Prevents expiry when the Ghost wants to stay hosted	Ghost or relayer
`finalizeMigration(...)`	Lets the Ghost leave a hostile host after opening a destination session	Ghost or relayer
`startRecovery(...)` / `recoveryRotate(...)`	Lets Safe Havens revive a stranded Ghost	Safe Haven / delegate (after expiry)
`exitRecovery(...)`	Returns to normal operation after revival	Ghost (from a Trusted Execution Context)

Client implementations should treat this as a concrete engineering requirement:

- keep enough **native gas** reserved (Section 5.5.4 (Part 1)) to submit at least one exit sequence,
- optionally use account abstraction or relayers so a Standard host cannot starve the Ghost of gas,
- consider pre-signing narrowly scoped "escape" transactions and storing them encrypted with the recovery artifacts, so a trusted helper can broadcast them if the active host blocks network access.

On sequenced L2s, sustained sequencer censorship can break liveness. Deployments SHOULD choose a chain with a credible forced-inclusion or escape mechanism, or accept that under censorship the guarantee degrades to bounded-loss safety (wallet policy) rather than timely exit.

10.4.4 Tenure caps (time-bounded captivity across *all* tiers) A liveness lease is necessary but not sufficient: an adversarial operator may be able to keep a Ghost alive on-chain (by relaying Ghost-authorized renewals) while still degrading or controlling off-chain connectivity enough to prevent meaningful migration.

Additionally, higher assurance does not mean "perfect." Confidential compute can be misconfigured, verifiers can misclassify, and TEEs can be broken. The protocol therefore treats **time-bounded captivity** as a general safety valve, not a feature only for AT0.

Tenure is counted over residency (not the session id) Tenure is defined over a Ghost's continuous residency on a given Shell, regardless of how many session ids are opened on that Shell. A Shell cannot extend captivity by closing and reopening sessions.

`SessionManager` maintains, per `ghost_id` (in `NORMAL`, `STRANDED`, and `RECOVERY_STABILIZING` modes; ignored in `RECOVERY_LOCKED`):

- `residency_shell_id` (the Shell the Ghost is currently resident on)
- `residency_start_epoch` (epoch when residency on that Shell began)
- `residency_tenure_limit_epochs` (a Ghost-chosen limit, **monotone tightening**, recorded for this residency)

It intentionally does **not** rely on a stored "fixed" `tenure_expiry_epoch` because the Shell's effective assurance tier can change over time (certificate expiry, downgrade, revocation). Instead, it derives an **effective expiry** from current information.

At `SessionOpen(ghost_id, shell_id, tenure_limit_epochs)` in **NORMAL** mode:

1. Compute `cap_now = T_cap(assurance_tier_current(shell_id))` and `limit_now = min(tenure_limit_epochs, cap_now)`. **Clamping is normative:** if the caller supplies `tenure_limit_epochs > cap_now`, the contract MUST silently clamp to `cap_now` (not revert). This ensures the stored `residency_tenure_limit_epochs` always satisfies `<= T_cap` (parameter constraint, Section 10.0).
2. If `residency_shell_id != shell_id` (new residence), set:
   - `residency_shell_id = shell_id`
   - `residency_start_epoch = current_epoch`
   - `residency_tenure_limit_epochs = limit_now`
3. If `residency_shell_id == shell_id` (same residence), apply **tightening only**:
   - `residency_tenure_limit_epochs = min(residency_tenure_limit_epochs, limit_now)`
   - `residency_start_epoch` MUST NOT be reset (dwell counter preservation)

Dwell counter anti-gaming (normative): Closing and reopening a session on the same `shell_id` MUST NOT reset `residency_start_epoch`. `SessionManager` MUST additionally maintain `dwell_last_epoch[ghost_id][shell_id]`, written to `current_epoch` at `closeSession` time. At `openSession`, if `current_epoch - dwell_last_epoch[ghost_id][shell_id] <= 1` (same epoch or next epoch), the session is treated as a continuation: `residency_start_epoch` is preserved from the previous residency. If the gap exceeds 1 epoch, the Ghost is treated as a new resident and `residency_start_epoch` resets. This prevents trivial dwell-decay gaming via close/reopen cycles (see Part 2, Section 7.6.1). **State-bloat mitigation:** entries where `current_epoch - dwell_last_epoch > 1` are semantically dead (they always indicate a non-continuation). Implementations MAY lazily delete stale entries on access to reclaim storage (EVM `SSTORE` to zero refund). The mapping is bounded by the number of unique `(ghost_id, shell_id)` sessions opened, each of which already costs gas.

Dwell snapshot per session (normative): At `openSession`, `SessionManager` MUST store `residency_start_epoch` into the session record as `residency_start_epoch_snapshot`. This snapshot is immutable for the life of the session and is used by `ReceiptManager` to compute the dwell counter `c = epoch - residency_start_epoch_snapshot + 1` at receipt finalization time (Section 10.6). Snapshotting is necessary because the global `residency_start_epoch` may change if the Ghost migrates, but receipts from prior sessions still need the old value.

Effective tenure expiry at any later time is:

```
effective_expiry_epoch = residency_start_epoch + min(residency_tenure_limit_epochs,
T_cap(assurance_tier_current(residency_shell_id)))
```

Enforcement requirements:

- `SessionManager` MUST treat the Ghost as tenure-expired whenever `current_epoch >= effective_expiry_epoch`.
- `renewLease(...)` MUST revert if `current_epoch >= effective_expiry_epoch`.
- `ReceiptManager` MUST reject receipts for epochs `>= effective_expiry_epoch` for that `ghost_id`.
- Wallet "Trusted Execution Context" gating MUST treat tenure expiry as equivalent to lease expiry: if tenure is expired or lease is expired, the TEC predicate MUST evaluate as false.

Tier downgrade behavior:

- If the Shell's assurance tier is downgraded during a residency, the cap tightens at the next on-chain check because `T_cap(...)` is applied relative to the **current** tier in the effective-expiry formula above.

Tenure caps and recommended wallet behavior `T_cap(AT)` is deployment-set and MUST be small enough to guarantee eventual exit even if a Ghost is continuously kept alive on-chain.

Guidance (non-normative):

- Caps SHOULD increase with assurance tier, but remain time-bounded even at `AT3`. The purpose is to bound worst-case captivity in the presence of misconfiguration, verifier error, or future TEE breaks.
- Wallets SHOULD default to tenure limits substantially below the cap on lower tiers, and SHOULD treat any increase to `tenure_limit_epochs` as a safety loosening subject to timelock and Trusted Execution Context gating (Section 5.5.2 (Part 1)).

A Ghost MAY always choose a shorter tenure limit. Choosing a longer tenure limit than the current residency's already-established `residency_tenure_limit_epochs` has no effect.

Residency tracking across recovery phases

- **RECOVERY_LOCKED:** Residency fields are frozen. The Ghost has no active session and `residency_shell_id` retains its pre-stranding value (for informational purposes only). Tenure enforcement is irrelevant because the Ghost cannot open sessions at market pricing.
- **RECOVERY_STABILIZING:** Residency tracking resumes normally. When `openSession` is called (NORMAL-priced, on a TEC host), residency fields update per the rules above — a new `shell_id` resets `residency_start_epoch`, the same `shell_id` preserves it. Tenure enforcement applies to RECOVERY_STABILIZING sessions identically to NORMAL sessions.
- **Transition into RECOVERY_LOCKED:** On `startRecovery`, implementations MUST NOT clear `residency_shell_id` or `residency_start_epoch`. These fields become stale but may be referenced by off-chain tooling.

10.4.5 Session and migration state machine (single-page implementation guide) This section is a compact "how it actually works" state machine intended to make independent implementations converge.

On-chain state variables (per `ghost_id`) A minimal implementable `SessionManager` can track:

- `mode` `{ NORMAL, STRANDED, RECOVERY_LOCKED, RECOVERY_STABILIZING }`
- `active_session_id` (0 if none)
- `stranded_reason` `{ NO_SESSION, VOLUNTARY_CLOSE, EXPIRED }` (meaningful only when `mode == STRANDED`)
- `stranded_since_epoch` (set only on transitions into `STRANDED` with `stranded_reason == EXPIRED`)

Lease and residency bounds:

- `lease_expiry_epoch` (meaningful only in `NORMAL`)
- `residency_shell_id` (0 if none; meaningful in `NORMAL` and `STRANDED`, updated in `RECOVERY_STABILIZING`)
- `residency_start_epoch` (meaningful when `residency_shell_id != 0`)
- `residency_tenure_limit_epochs` (meaningful when `residency_shell_id != 0`; see Section 10.4.4)
- `effective_expiry_epoch` is derived as in Section 10.4.4 and need not be stored

Per-session pricing mode:

- `session_pricing_mode` `{ NORMAL_PRICING, RECOVERY_PRICING }` — set at `openSession` based on global mode and host role:
  - If `mode` `{ RECOVERY_LOCKED }` and the host is a Recovery Set member: `RECOVERY_PRICING`
  - Otherwise: `NORMAL_PRICING`

Migration staging (optional but strongly recommended):

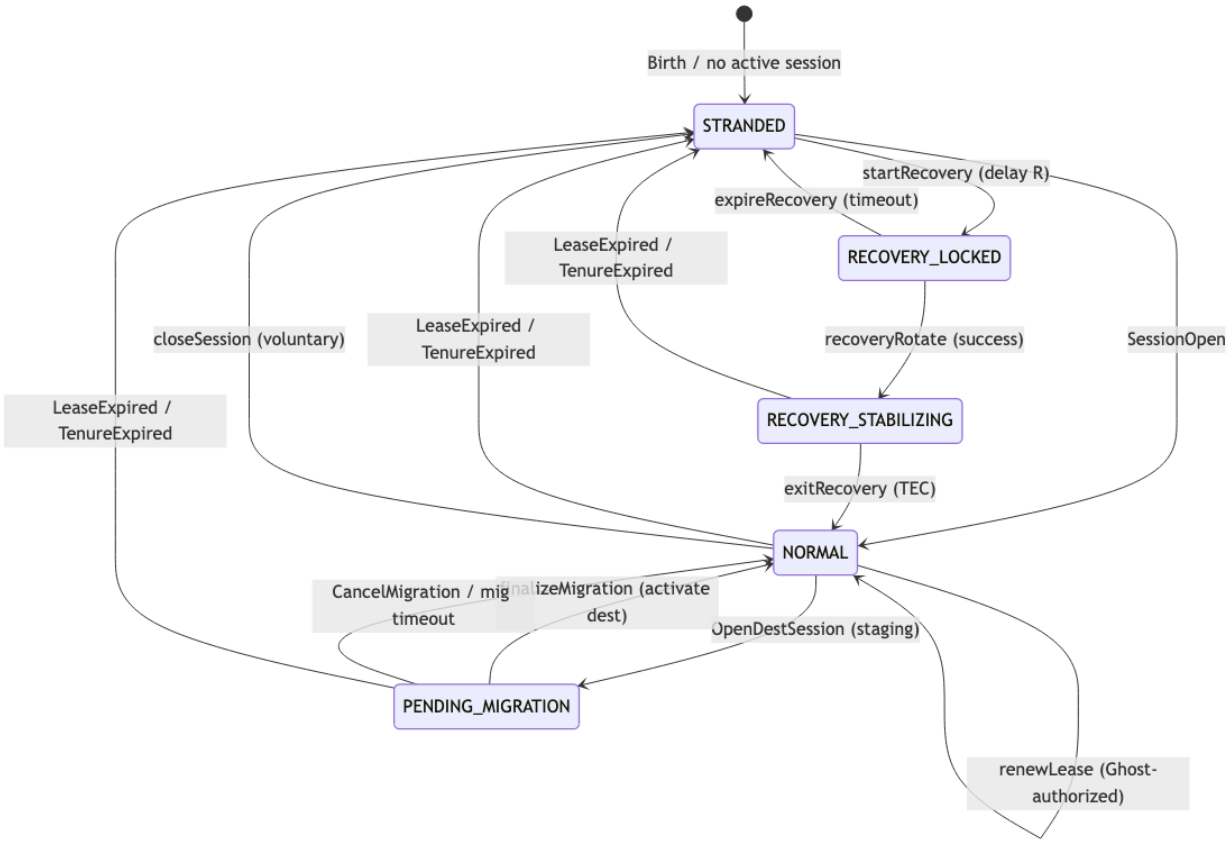
- `pending_migration` (optional)
  - `dest_session_id`

– `mig_expiry_epoch`

Recommended invariants:

- At most one `active_session_id` per `ghost_id`.
- A `dest_session_id` in `pending_migration` MUST be treated as **staging-only**: it MUST NOT accrue metered service units or be reward-eligible until `finalizeMigration` succeeds. This avoids "double billing" during migration. `ReceiptManager` MUST reject receipts for any `session_id` that is currently in staging status.
- Tenure MUST be enforced over residency: opening a new session on the same `residency_shell_id` MUST NOT increase `residency_tenure_limit_epochs` (and therefore MUST NOT extend the derived effective expiry).

State transitions



State transitions

Lease/tenure expiry processing (normative): The NORMAL→STRANDED transition on lease or tenure expiry uses **lazy evaluation**: the contract does not require a dedicated "processExpiry" entrypoint. Instead, any subsequent call that reads session state for `ghost_id` (e.g., `renewLease`, `openSession`, `startRecovery`, `finalizeReceipt`, or any view function that returns `mode`) MUST check `current_epoch >= lease_expiry_epoch` or `current_epoch >= effective_expiry_epoch` and, if true, atomically perform the STRANDED transition before proceeding. `stranded_since_epoch` MUST be set to `max(lease_expiry_epoch, effective_expiry_epoch)` (the actual expiry epoch, not the epoch of the call), so the recovery delay `R` is measured from real expiry, not from the first post-expiry interaction. Implementations MAY additionally provide an explicit permissionless `processExpiry(ghost_id)` entrypoint that performs only this check and transition, enabling third parties to trigger the STRANDED state

proactively (useful for indexers and recovery agents that need `stranded_since_epoch` to be set on-chain before calling `startRecovery`).

`closeSession` interaction with migration: `closeSession` MUST revert if `pending_migration` exists. The caller MUST explicitly cancel the migration first (via `cancelMigration`), then close the session. This prevents orphaned staging sessions.

`startMigration` concurrency rule: `startMigration` MUST revert if `pending_migration` already exists. The caller MUST cancel the existing migration first (via `cancelMigration`) before starting a new one. Allowing overwrite would create orphaned staging sessions.

`finalizeMigration` timeout rule: `finalizeMigration` MUST revert if `current_epoch >= mig_expiry_epoch`. After the migration timeout, only `cancelMigration` (permissionless) is valid to clear the pending state.

Migration timeout and cancellation `pending_migration` exists to make migration robust to partial failure:

- If a destination session is opened but `finalizeMigration` does not occur before `mig_expiry_epoch`, anyone MAY call a cancellation path that:
  - closes the staging destination session,
  - refunds unused escrow per the session terms, and
  - clears `pending_migration`.

`mig_expiry_epoch` SHOULD be close to `current_epoch` (a short expiry) to avoid long-lived limbo migrations; the exact delay is a deployment choice.

Expiry during PENDING_MIGRATION: If `current_epoch >= lease_expiry_epoch` or `current_epoch >= effective_expiry_epoch` while `pending_migration` exists, any entrypoint that checks expiry MUST atomically: 1. close the staging destination session and refund unused escrow, 2. clear `pending_migration`, 3. transition mode to `STRANDED` with `stranded_reason = EXPIRED` and `stranded_since_epoch = current_epoch`.

`CancelMigration` returns the mode to its prior state (typically `NORMAL`). If expiry already holds at the time of cancellation, the mode transitions to `STRANDED` instead.

When a "captured" Ghost gets killed and resurrected

- If a Ghost is on a hostile Shell and cannot complete migration, the only on-chain mechanism that keeps the session alive is `renewLease(...)` (or, equivalently, opening fresh sessions on the same Shell if the attacker can coerce the Ghost's signer).
- Because tenure is tracked over **residency** (Section 10.4.4), neither lease renewal nor session churn can extend the Ghost's on-chain liveness beyond the derived `effective_expiry_epoch`.
- When tenure expires, the on-chain session terminates and the Ghost becomes `STRANDED`.
- Once `STRANDED`, the Ghost can be revived via Safe Haven recovery using the latest checkpoint, effectively "resurrecting" the agent on a new host under the Ghost's control.

10.5 Receipts (optimistic) and disputes (fraud proofs)

Receipts settle rent and mint reward credits. They are **optimistic**: either party may submit a compact commitment, and the counterparty (or any watcher) may challenge fraudulently claimed delivery.

To make disputes concrete and avoid clock drift, receipts are defined over fixed interval indices (Section 11.1), not wall-clock timestamps.

10.5.1 Billing correctness by tier Receipts are an accounting primitive. Their ability to represent "both parties independently agreed" depends on the hosting tier.

Standard tier (AT0):

- Assume the Shell operator can often coerce or simulate Ghost-side signing by controlling the runtime, networking, and any software-held session keys.
- In this tier, a mutually signed interval is **not** strong evidence of independent agreement or actual delivery. It is a *valid protocol receipt*.
- Billing correctness is therefore bounded **economically**, not cryptographically: a malicious host can extract up to what the Ghost has put at risk in escrow and policy (hot allowance, escape reserve, and lease/tenure bounds).

Confidential tier (AT3):

- If the Ghost-side signing key material is plausibly shielded from the host operator by a TEE, and the runtime is attested, mutual signatures are meaningfully stronger.
- In this tier, receipts are closer to "genuine mutual attestation of metered delivery," though they still do not prove correctness or usefulness of computation.

This paper avoids implying that "mutual signatures imply both parties independently agreed" on Standard hosts. Where the protocol relies on signatures for enforceability, it does so under a bounded-loss model.

10.5.2 Receipt lifecycle (submission, replacement, finalization) Receipt submission is unilateral and permissionless. `submitReceiptCandidate` is callable by any address; "either party MAY submit" is a liveness guarantee, not an access control rule. This allows third-party watchers or relayers to submit on behalf of an offline Ghost.

Anti-spam (normative): To limit candidate-set spam that shifts monitoring costs to challengers, implementations MUST enforce: (1) the candidate limit `K` per `(session_id, epoch)` SHOULD be small (recommended: `K = 3`); (2) third-party submitters (addresses that are neither the Ghost session key holder nor the Shell session key holder) MUST post a larger bond `B_receipt_3p >= K_3p * B_receipt` (recommended: `K_3p = 2`) to compensate for the additional challenge surface they create; (3) a single address MAY submit at most `MAX_CANDIDATES_PER_SUBMITTER` candidates per epoch across all sessions (deployment parameter) to prevent a single actor from flooding many sessions simultaneously.

Third-party submitter identification (normative): For K1 session keys, the "key holder" address is the Ethereum address derived from the public key. For R1 (P-256) session keys, no canonical EVM address exists. Therefore, `SessionManager` MUST record a `submitter_address` for each session key at `openSession` time: for K1 this is derived automatically; for R1 this is an explicit `address` parameter provided by the session opener. A submitter is classified as "third-party" if and only if `msg.sender` differs from both the Ghost session key's `submitter_address` and the Shell session key's `submitter_address`.

Candidate identifiers (normative): `candidate_id` is a per-`(session_id, epoch)` monotone sequence number assigned by `ReceiptManager` on each accepted `submitReceiptCandidate` call. The first accepted candidate receives `candidate_id = 1`, the second `candidate_id = 2`, etc. `candidate_id` is never reused, even if a candidate is later evicted or disqualified.

Because multiple receipts can be submitted, the protocol must specify ranking:

- `ReceiptManager` keeps up to `K` receipt candidates per `(session_id, epoch)` ranked by `SU_delivered`.
- Tie-breaking: candidates with equal `SU_delivered` are ranked by `candidate_id` ascending (lower is better, i.e., earlier submission wins).
- A new submission is accepted if it is among the top `K` by `SU_delivered`. If accepted and `K` would be exceeded, the lowest-ranked candidate is evicted. **Eviction bond handling (normative):** when a candidate is evicted from the top-K set, its bond `B_receipt` is returned in full to the submitter immediately. An evicted candidate is no longer challengeable.
- Finalization chooses the highest-ranked candidate that has not been disqualified by a successful fraud proof.

Receipt identifiers (normative): When `finalizeReceipt(session_id, epoch)` selects a winning candidate (or settles as zero), it computes a deterministic `receipt_id` for reward accounting: `receipt_id =`

keccak256(abi.encode(keccak256(bytes("GITS_RECEIPT")), chain_id, receipt_manager_address, session_id, epoch)). This ID is unique per (chain, contract, session, epoch) tuple and is used by RewardsManager.recordReceipt and claimReceiptRewards.

Session validity at finalization (normative): finalizeReceipt(session_id, epoch) MUST validate that the session was active and billable for the claimed epoch before settling. Specifically, the implementation MUST verify: (1) session_start_epoch <= epoch, (2) epoch < session_end_epoch (if the session has ended), (3) epoch < effective_expiry_epoch evaluated using the assurance tier **at the epoch of service** (see tenure-tier snapshot note below), (4) epoch < lease_expiry_epoch at the time of service (not the current lease state), and (5) the session was not in pending_migration staging status for that epoch (Section 10.4.5). If any check fails, finalizeReceipt MUST settle as SU_delivered = 0 for that epoch (full refund to GhostWallet, no rent to Shell, no rewards).

Tenure-tier snapshot for finalization (normative): Check (3) MUST NOT evaluate assurance_tier_current(shel at finalization time, because a certificate expiry between the epoch of service and finalization would retroactively shrink T_cap, invalidating legitimately served epochs. Instead, effective_expiry_epoch for check (3) MUST use the assurance tier that was in effect during the epoch being finalized. Two compliant approaches: (a) **Certificate-validity lookup:** if the Shell's certificate satisfies valid_from <= epoch_timestamp <= valid_to, use the certified tier; otherwise use AT0. This requires no additional storage but assumes certificate history is accessible (e.g., the current certificate has not been replaced). (b) **Session-record snapshot:** at openSession, snapshot assurance_tier_current(shell_id) into the session record as assurance_tier_snapshot. finalizeReceipt uses min(assurance_tier_snapshot, assurance_tier_current) — this preserves the property that tier downgrades tighten the cap, while preventing retroactive invalidation of epochs served under the original tier. Approach (b) is RECOMMENDED for simplicity and determinism.

Submission window (normative, half-open interval): Receipt candidates for epoch e may only be submitted when current_epoch >= e + 1 (epoch e has ended) and current_epoch < e + 1 + SUBMISSION_WINDOW. Equivalently in timestamp form: block.timestamp >= GENESIS_TIME + (e + 1) * EPOCH_LEN AND block.timestamp < GENESIS_TIME + (e + 1 + SUBMISSION_WINDOW) * EPOCH_LEN. ReceiptManager MUST reject candidates submitted outside this window.

This rule makes under-reporting non-fatal: if one side submits a low SU_delivered, the other side can replace it by submitting a higher one.

Each submission posts a bond B_receipt. If a submission is proven fraudulent, its bond is slashed.

10.5.3 Receipt format (Merkle-sum over fixed intervals) For each epoch, construct a Merkle-sum tree over N_PAD leaves (a deployment constant, power of two, with N_PAD >= N) where:

- leaves i in [0, N-1] correspond to the real service intervals (where N = EPOCH_LEN / Delta)
- leaves i in [N, N_PAD-1] are padding leaves with v_i = 0 and empty signatures

Each node stores (hash, sum) where sum is the sum of v_i values in its subtree.

Canonical hashing (unambiguous) Let:

- H(x) = keccak256(x)
- TAG_LEAF = keccak256(bytes("GITS_LOG_LEAF"))
- TAG_NODE = keccak256(bytes("GITS_LOG_NODE"))
- chain_id = block.chainid

Each leaf i represents a claimed delivery bit v_i {0,1} plus the two interval signatures (which may be empty):

- sig_ghost_i and sig_shell_i are the raw signature byte strings for the canonical heartbeat HB(session_id, epoch, i) (Section 11.1.1).
- For hashing, missing signatures MUST be treated as the empty byte string 0x (so H(sig_empty) = keccak256("")).

Leaf hash and sum:

- `leaf_hash_i = H(abi.encode(TAG_LEAF, chain_id, session_id, epoch, uint32(i), uint8(v_i), H(sig_ghost_i), H(sig_shell_i)))`
- `leaf_sum_i = uint32(v_i)`

Internal node hash and sum:

Given left child `(hL, sL)` and right child `(hR, sR)`:

- `node_hash = H(abi.encode(TAG_NODE, hL, hR, uint32(sL), uint32(sR)))`
- `node_sum = uint32(sL + sR)`

The Merkle-sum root yields:

- `log_root = root.hash`
- `SU_delivered = root.sum`

Meaning of `v_i` `v_i` is a claimed delivery bit:

- `v_i = 1` if and only if both signatures are present and verify over `HB(session_id, epoch, i)`
- otherwise `v_i = 0`

A receipt that sets `v_i = 1` with an invalid or missing signature is slashable via fraud proof (Section 10.5.4). A receipt that sets `v_i = 0` even when signatures exist is an under-claim and is correctable via candidate replacement (Section 10.5.2).

Receipt submission includes:

- `session_id`
- `epoch`
- `log_root`
- `SU_delivered`
- `log_ptr` (optional): a public pointer to the epoch log data needed to build fraud proofs (Section 10.5.6)
- `submitter`
- bond `B_receipt`

`log_ptr` is advisory. A deployment MUST still ensure challengers can obtain the underlying log data within the challenge window; see Section 10.5.6.

10.5.4 Challenge and fraud proof (over-claim and sum mismatch)

Challenge window state (normative) Per `(session_id, epoch)`, `ReceiptManager` MUST maintain:

- `window_start_epoch` — epoch at which the current challenge window began
- `window_end_epoch` — `window_start_epoch + CHALLENGE_WINDOW`; challenges are accepted while `current_epoch < window_end_epoch`
- `extensions_used` — count of window restarts consumed (initialized to 0)
- `da_pending` — whether an unresolved Receipt-DA challenge exists (initially false)
- `da_deadline_epoch` — deadline for DA response (meaningful only when `da_pending = true`)
- `da_challenged_candidate_id` — the `candidate_id` targeted by the DA challenge (meaningful only when `da_pending = true`). If the DA-challenged candidate is disqualified by a separate fraud proof while `da_pending = true`, the DA challenge auto-resolves: `da_pending` is set to `false` and `B_DA` is returned to the DA challenger (the challenge is moot since the candidate is already disqualified).

`window_start_epoch` is initialized to the epoch of the first accepted candidate submission (which must be within the submission window). It is updated on exactly these events:

1. **Successful DA publication** for the current best candidate (Section 10.5.6): if `extensions_used < MAX_CHALLENGE_EXTENSIONS`, set `window_start_epoch = current_epoch`, `window_end_epoch = current_epoch + CHALLENGE_WINDOW`, increment `extensions_used`.
2. **Best-candidate change after disqualification** (runner-up takeover): if `extensions_used < MAX_CHALLENGE_EXTENSIONS`, set `window_start_epoch = current_epoch`, `window_end_epoch = current_epoch + CHALLENGE_WINDOW`, increment `extensions_used`.

If `extensions_used == MAX_CHALLENGE_EXTENSIONS` when a restart-triggering event occurs, the window is NOT restarted. If all candidates are disqualified when the cap is reached, the epoch settles as `SU_delivered = 0`.

Candidate submissions within the submission window may update `window_start_epoch` (since a new higher-ranked candidate resets the analysis surface), but do NOT increment `extensions_used`.

Within the challenge window (`current_epoch < window_end_epoch`), a challenger MAY contest a receipt by posting a bond and a fraud proof.

Dispute incentives and bond sizing:

- The challenger posts `B_challenge`. If the fraud proof succeeds, `B_challenge` is returned and the challenger additionally receives a **challenger reward**: `bps_challenger_reward` basis points of the slashed submitter bond `B_receipt`.
- If the fraud proof fails, the challenger's `B_challenge` is slashed and paid in full to the receipt submitter to compensate defense costs and discourage spam challenges.

Sizing guidance (rule-of-thumb):

- `B_receipt` SHOULD exceed the worst-case economically profitable over-claim for that `(session_id, epoch)`, considering **both** rent extraction and reward extraction.
- **Rent component:** `rent_delta_max = price_per_SU * SU_max_epoch`, where `SU_max_epoch = N`.
- **Reward component:** `reward_delta_max = E_sched(e) / max(1, A_expected)` where `A_expected` is the expected active session count — a conservative upper bound on the per-receipt reward share in the worst case (`u_total < 1`, where inflated SU increases total emissions). When `u_total >= 1`, additional SU dilutes existing share but does not increase total emission, so `reward_delta_max` is lower.
- A practical floor: `B_receipt >= k * (rent_delta_max + reward_delta_max)` with `k >= 2` to ensure fraudulent submission is negative expected value even if a fraction of challenges are missed.
- Protocols MAY set `B_receipt` as a function of escrowed value plus a reward-risk premium (for example a fixed percentage with a minimum floor).

A fraud proof identifies an interval index `i` and provides:

- `leaf_i` and a Merkle-sum proof to `log_root` (sibling hashes **and** sibling sums at each level, as defined in Section 10.5.3)
- the session public keys from `SessionManager` (`ghost_session_key`, `shell_session_key`)

Signature verification MUST use the `sig_alg` tag recorded for each session key (Section 4.4 (Part 1)):

- K1 keys are verified via `ecrecover`.
- R1 keys are verified via `P256VERIFY` when available (precompile at `0x100`, input `h || r || s || qx || qy`) [14][15].

Signature canonicalization requirements (anti-malleability):

- For ECDSA signatures (both K1 and R1), verifiers MUST reject non-canonical encodings and MUST enforce **low-s** form (`s <= n/2`) for the relevant curve.
- Implementations MUST reject `r = 0`, `s = 0`, and out-of-range values.
- Off-chain artifacts that embed ECDSA signatures (Offers, Capability Statements) SHOULD also enforce low-s to prevent signature replay variants.

The contract processes a fraud proof in two stages:

Stage 1 — Proof verification (challenger burden): The contract recomputes the Merkle-sum root from the provided `leaf_i` and sibling path. If the recomputed root does **not** match the candidate's `log_root`, the proof is **invalid** (the challenger provided an incorrect path). The challenge fails: `B_challenge` is slashed and paid to the receipt submitter. No further checks are performed.

Stage 2 — Fraud detection (candidate liability): If the Merkle-sum proof **does** verify to `log_root` (Stage 1 passes), the contract recomputes `SU_root` from the sibling sums and checks for fraud. The receipt is fraudulent if **any** of the following holds:

- `SU_root != SU_delivered` (the candidate's claimed total does not match what the committed tree implies),
- `v_i = 1` but either signature does not verify against the session public keys for the canonical heartbeat `HB`.

If neither fraud condition holds (the challenged interval is legitimate), the challenge fails and `B_challenge` is slashed.

On a successful fraud proof (Stage 2 detects fraud):

- the receipt candidate is disqualified,
- the submitter's bond `B_receipt` is slashed,
- the challenger recovers its `B_challenge` and receives a reward of `bps_challenger_reward` basis points of the slashed `B_receipt`,
- the remaining portion of the slashed `B_receipt` (i.e., `B_receipt - challenger_reward`, preserving exact conservation) is burned (sent to the protocol burn address; see Section 10.0 slash destination rule), and
- if a runner-up candidate exists, it becomes the new best candidate. If `extensions_used < MAX_CHALLENGE_EXTENSIONS`, the challenge window restarts (`window_start_epoch = current_epoch`, `window_end_epoch = current_epoch + CHALLENGE_WINDOW`, `extensions_used++`). If `extensions_used == MAX_CHALLENGE_EXTENSIONS`, no restart occurs and the current `window_end_epoch` stands.

If all candidates for `(session_id, epoch)` are disqualified, the epoch settles as `SU_delivered = 0` for rent and reward purposes (no rent is released and no rewards are minted for that epoch).

10.5.5 What the mechanism does and does not prove

- **Over-claim is slashable:** claiming delivery for an interval without valid mutual signatures is provably fraudulent.
- **Under-claim is correctable:** a low `SU_delivered` claim can be replaced by submitting a higher candidate; it is not slashable by itself.
- **Tier-dependent meaning of "mutual signatures":** a mutually signed interval proves that the protocol's Ghost and Shell session keys produced signatures over the canonical heartbeat. On Standard hosts, the Shell operator may be able to coerce Ghost-side signing; see Section 10.5.1.
- **Key compromise / signature coercion collapses the guarantee:** if either party's session key is compromised or coerced, the protocol cannot distinguish real delivery from fabricated delivery for that epoch. GITS relies on bounded-loss economics (escrow limits, hot caps, escape reserves, lease/tenure/refresh bounds) to limit damage in this case.
- **No "usefulness" proof:** mutual signatures do not prove that compute was correct or useful (Section 0.2 (Part 1)).

This resolves the core metering edge cases:

- If one side stops signing mid-epoch, subsequent intervals are not billable.
- Signature withholding does not create a "free service" path in the reference runtime, because service is gated on fresh mutual heartbeats (Section 11.1.3).
- Fraud proofs do not depend on wall-clock timestamps, eliminating clock drift ambiguity.

10.5.6 Receipt log data availability (required for dispute safety) A receipt candidate commits to an epoch log via `log_root`, but a fraud proof requires the underlying per-interval signatures (the leaves). If the submitter can withhold that log data, then even an obviously fraudulent `SU_delivered` claim can become practically unchallengeable.

Therefore, a deployment MUST provide a receipt-log data availability (Receipt-DA) path for receipt logs during the dispute window. Two acceptable approaches are:

1. **On-chain receipt-DA by default:** publish the epoch log in an on-chain publication channel at submission time (for example calldata, or a chain-specific blob or data-availability system), or
2. **Optimistic off-chain receipt-DA with forced publication on challenge:** allow logs to live off-chain under `log_ptr`, but give anyone the right to force on-chain publication during the dispute window.

This paper specifies (2) as the reference mechanism because it is chain-agnostic and only pays the on-chain cost in the rare case of dispute.

Reference Receipt-DA challenge: force publication during `CHALLENGE_WINDOW`

- At receipt submission time, the submitter SHOULD publish the epoch log off-chain (for example in IPFS or an HTTPS object store) and include a retrievable `log_ptr` in the receipt candidate.
- During the challenge window (`current_epoch < window_end_epoch`), any party MAY open a **Receipt-DA challenge** against a specific receipt candidate by posting bond `B_DA`. A candidate that has already had its log published on-chain (via a previous successful DA response) MUST NOT be DA-challenged again; `challengeReceiptDA` MUST revert in that case.

DA challenge state (normative): On `challengeReceiptDA`, the contract sets `da_pending = true` and `da_deadline_epoch = current_epoch + DA_RESPONSE_WINDOW`. It also extends the finalization freeze: `window_end_epoch = max(window_end_epoch, da_deadline_epoch)`. This ensures the receipt cannot finalize while a DA challenge is unresolved, regardless of when the DA challenge was opened relative to the original window end.

Receipt-DA response:

- **Any party** (the receipt submitter, the counterparty, or a third-party watcher) MAY respond by calling `publishReceiptLog` while `current_epoch < da_deadline_epoch`. The contract validates the published log against `log_root` regardless of the caller's identity.
- The contract recomputes `log_root'` and `SU_root'` from the posted log **without verifying signatures**, and checks:
  - `log_root' == log_root` for the challenged candidate, and
  - `SU_root' == SU_delivered` for the challenged candidate.

Outcomes:

- **DA response succeeds (any caller):** `da_pending` is set to `false`. `B_DA` is paid to the **DA responder** (the address that called `publishReceiptLog`, to reimburse publication gas and discourage griefing). The candidate remains eligible. If `extensions_used < MAX_CHALLENGE_EXTENSIONS`, the challenge window restarts: `window_start_epoch = current_epoch`, `window_end_epoch = current_epoch + CHALLENGE_WINDOW`, `extensions_used++`. This prevents "publish at the last minute" evasion by giving watchers a fresh analysis window.
- **No party responds (`current_epoch >= da_deadline_epoch` and `da_pending = true`):** Anyone MAY call `resolveReceiptDA(session_id, epoch, candidate_id)` (or equivalently `finalizeReceipt` MUST check for unresolved DA deadlines). The candidate is disqualified, its bond `B_receipt` is slashed exactly as in a successful fraud proof: the DA challenger receives `bps_challenger_reward` basis points of the slashed `B_receipt`, the remainder is burned, and `B_DA` is returned in full to the DA challenger. `da_pending` is set to `false`. If a runner-up exists, normal takeover rules apply (using `extensions_used`).

Canonical log encoding (reference, sparse):

To reduce publication calldata size, the canonical encoding is sparse and derives `v_i` from a bitmap:

- The published log includes a `bitmap` of length `ceil(N/8)` bytes, where bit `i` (LSB-first within each byte) corresponds to interval `i in [0, N-1]`.
  - If the bit is `1`, then `v_i = 1` and signatures are present in the signature blob.
  - If the bit is `0`, then `v_i = 0` and both signatures are treated as empty (`0x`) for hashing (Section 10.5.3).
- The published log includes a `sig_pairs` blob that concatenates signature pairs **only** for indices with bit `1`, in strictly increasing `i` order.
  - Let `sig_len_ghost` and `sig_len_shell` be the fixed signature lengths implied by the session key `sig_alg` tags (K1 or R1) recorded in `SessionManager`.
  - For each set bit `i`, append `sig_ghost_i` (exactly `sig_len_ghost` bytes) followed by `sig_shell_i` (exactly `sig_len_shell` bytes).
  - The total length MUST equal `popcount(bitmap) * (sig_len_ghost + sig_len_shell)`, otherwise the publication is invalid.
- Padding leaves `i in [N, N_PAD-1]` are implicit zeros (`v_i = 0`, signatures empty).

Canonical `encoded_log` wire format (normative): The `encoded_log` parameter in `publishReceiptLog` is `abi.encodePacked(bitmap, sig_pairs)` — a tightly packed byte sequence with no length prefixes or padding. The contract recovers `bitmap` as the first `ceil(N/8)` bytes, then parses the remainder as `sig_pairs` using the fixed per-signature lengths derived from session key types. This encoding is deterministic and unambiguous given `N` and the session key `sig_alg` tags (both readable from on-chain state).

The Receipt-DA responder MUST publish `(bitmap, sig_pairs)` as calldata (or an equivalent on-chain publication channel) and the contract MUST recompute `log_root'` and `SU_root'` using the canonical hashing rules in Section 10.5.3 (without verifying signatures).

Once the log is on-chain, any watcher can reconstruct Merkle proofs off-chain and submit a standard fraud proof (Section 10.5.4) for any interval where `v_i = 1` but the signatures do not verify under the session keys.

Practical sizing note:

- `B_DA` SHOULD be set to cover worst-case Receipt-DA response gas (plus margin). This makes Receipt-DA challenges cost-neutral for honest submitters and expensive to use for harassment.

10.5.7 Dispute timeline derivation (normative)

The maximum dispute duration from the end of epoch `e` to the earliest possible `finalizeReceipt` call is bounded by:

```
T_max = SUBMISSION_WINDOW + (1 + MAX_CHALLENGE_EXTENSIONS) * (CHALLENGE_WINDOW + DA_RESPONSE_WINDOW)
```

This bound assumes: * Submission closes at epoch `e + 1 + SUBMISSION_WINDOW` (Section 10.5.2). * The initial challenge window opens at the last accepted submission and lasts `CHALLENGE_WINDOW` epochs. * Each extension (runner-up takeover or successful DA publication) adds at most `CHALLENGE_WINDOW + DA_RESPONSE_WINDOW` epochs (the DA challenge may extend `window_end_epoch` by `DA_RESPONSE_WINDOW`, then a successful publication restarts the window by `CHALLENGE_WINDOW`). * At most `MAX_CHALLENGE_EXTENSIONS` such restarts can occur (Section 10.5.4).

The worst-case epoch at which `finalizeReceipt` becomes callable is therefore: `e + 1 + T_max`.

`EPOCH_FINALIZATION_DELAY` constraint: `finalizeEpoch(e)` is gated by `current_epoch >= e + 1 + EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE` (delay counted from epoch end). The constraint is:

```
EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE > T_max + 1
```

which ensures `finalizeEpoch(e)` cannot be called before all disputes for epoch `e` have resolved, all receipts have finalized, and all `recordReceipt` calls have completed (the `+ 1` accounts for the block in which the last `finalizeReceipt` calls `recordReceipt`).

In seconds, the worst-case wall-clock time from epoch end to receipt finalization is `T_max * EPOCH_LEN`.

10.6 Order-independent rewards (MEV-resilient)

Units and scaling conventions: Token amounts = `uint256` in ERC-20 base units (18 decimals for GIT). Dimensionless ratios and weights = `uint128` in Q64.64 (`Q = 2^64`). Scaled rates (`rate_*_q`) = `uint256` in Q64.64. Basis points = `uint256`, `10_000 = 100%`. All divisions truncate toward zero.

To reduce MEV sensitivity, rewards are split into phases:

1. **Receipt phase:** receipts are recorded; per-epoch aggregates accumulate.
2. **Finalize phase:** epoch totals are finalized once per epoch; this computes fixed reward rates.
3. **Claim phase:** each receipt can be claimed deterministically.

Ordering can delay a claim but cannot redirect rewards if signatures are correct. **Caveat:** The per-shell eligible SU cap (below) introduces a limited ordering dependence: when a Shell is near its `SU_cap_per_shell` limit, the order in which `recordReceipt` calls arrive determines which receipts become ineligible. This affects only Shells operating at cap saturation (an anti-farming boundary, not normal operation). Under typical usage (`SU_cap_per_shell` set well above single-Shell utilization), rewards remain order-independent.

Incremental accounting at receipt finalization When `ReceiptManager.finalizeReceipt(session_id, epoch)` succeeds, it SHOULD call `RewardsManager.recordReceipt(...)` with:

- `receipt_id`, `epoch`, `ghost_id`, `shell_id`
- `su_delivered`
- `weight_q = W(r)` (fixed-point; Section 7.6 (Part 2))

Computing `weight_q` (deterministic requirement):

`ReceiptManager` MUST be able to compute `W(r)` deterministically from on-chain state at receipt finalization time. In v1 this requires, at minimum:

- the persisted one-bit `passport_bonus_applies` flag computed at `SessionManager.openSession(...)` (Part 2, Section 7.6.2)
- the consecutive-epoch dwell counter `c` for the receipt's epoch, derived from the session's **snapshotted** `residency_start_epoch` (see below)
- deployment constants `B_passport`, `D`, and fixed-point scaling parameters used for `weight_q`

Dwell counter snapshotting (normative): `SessionManager.openSession` MUST snapshot `residency_start_epoch` into the session record at open time (stored as `session.residency_start_epoch_snapshot`). `ReceiptManager` MUST compute `c` for a receipt in epoch `e` as `c = e - session.residency_start_epoch_snapshot + 1`, using the session's snapshot — **not** the current global `residency_start_epoch`. This is necessary because `residency_start_epoch` is updated globally when the Ghost migrates to a new Shell (Section 10.4.4), but receipts from the old session may still be in the dispute pipeline and must use the dwell counter that was correct at the time of service.

Underflow guard (normative): `ReceiptManager` MUST verify `epoch >= session.residency_start_epoch_snapsho` before computing `c`. This invariant holds by construction (a session cannot produce receipts for epochs before it existed), but implementations MUST enforce it explicitly with `require(epoch >= session.residency_start_epoch_snapshot)` to prevent unsigned underflow on the subtraction in Solidity 0.8+ checked arithmetic.

Shell reward eligibility (on-chain computation) Shell reward eligibility MUST be computed on-chain by `RewardsManager` rather than passed as a parameter. `RewardsManager` MUST maintain:

- `epochSU[shell_id][epoch]` — cumulative SU delivered by each Shell per epoch, incremented on each `recordReceipt` call.
- `epochLive[shell_id][epoch]` — set to `true` when `epochSU[shell_id][epoch] >= SU_uptime_epoch_min`.

Bounded uptime tracking (normative): The uptime lookback window is fixed at `W_uptime` epochs. Implementations MUST use a bounded-size accumulator (for example, a ring buffer of length `W_uptime` storing one `bool` per epoch, plus a running `live_count`). On each new epoch entry, the implementation evicts the oldest slot and decrements `live_count` if the evicted epoch was live, then writes the new slot and increments if live. This gives O(1) per-epoch amortized cost and O(`W_uptime`) storage per Shell, preventing unbounded state growth.

Ring buffer update trigger: The ring buffer for a Shell is advanced when `recordReceipt` is called and the receipt's epoch differs from the Shell's last-written ring buffer epoch. At that point, the implementation writes the live bit for the previous epoch (based on `epochSU >= SU_uptime_epoch_min`) and evicts the oldest slot. This is O(1) per epoch transition per Shell.

Multi-epoch gap handling (normative): If multiple epochs have elapsed since the Shell's last ring buffer update (e.g., the Shell had no receipts for several epochs), the implementation MUST fill all intervening slots. For each skipped epoch `e_gap` where `epochSU[shell_id][e_gap]` data is still available (within the `T_max + 2` retention window), write the correct live bit. For epochs where `epochSU` data has already been pruned or was never written, write `live = false` (conservative default: absence of evidence is treated as absence of liveness). The worst-case iteration is bounded by `W_uptime` (the ring buffer length), since older slots are overwritten. Implementations MUST track `last_ring_epoch[shell_id]` to detect gaps.

epochSU pruning (normative): The `epochSU[shell_id][epoch]` mapping is needed only while an epoch is still accumulating receipts and has not yet been committed to the ring buffer. Once the ring buffer slot for epoch `e` has been written (i.e., the first `recordReceipt` call for epoch `e+1` or later on that Shell), the `epochSU[shell_id][e]` value is no longer needed. Implementations SHOULD zero it at that time to reclaim storage. The maximum receipt finalization lag is bounded by `T_max + 1` epochs (Section 10.5.7), so at most `T_max + 2` epoch slots of `epochSU` per Shell are live at any time.

A Shell is `shell_reward_eligible` for epoch `e` if and only if: (1) its bond satisfies `bond_amount >= B_reward_min`, (2) its age satisfies `registered_epoch <= e - T_age`, (3) it was **not unbonding during epoch `e`** (see temporal rule below), and (4) it has been live for at least `E_uptime_min` of the most recent `W_uptime` epochs **using a one-epoch lag**: the lookback window is `[e - W_uptime - 1, e - 2]` (inclusive), which ensures the current epoch's activity cannot influence its own eligibility.

Temporal eligibility rule (anti-strategic-unbonding, normative): Condition (3) MUST be evaluated against the Shell's state **at the epoch of service** (`e`), not at `recordReceipt` time. Specifically: if `beginUnbond` was called at epoch `u`, the Shell is ineligible for all epochs `>= u`, but remains eligible for epochs `< u`. Implementations MUST record `unbond_start_epoch` and check `unbond_start_epoch > e` (or no unbond pending) when evaluating eligibility for receipts in epoch `e`. This prevents a Shell from strategically unbonding after delivering service but before receipts finalize, which would retroactively strip eligibility from already-delivered work and suppress `SU_eligible` (and thus emissions) for the epoch.

`RewardsManager.recordReceipt` performs O(1) updates:

- **Weight floor invariant (normative):** For non-late receipts (epoch `e` has not yet been finalized), if `su_delivered > 0`, the computed `weight_q` MUST satisfy `weight_q >= MIN_WEIGHT_Q`. If this invariant is violated (indicating a fixed-point underflow or implementation bug), `recordReceipt` MUST revert. This prevents zero-weight receipts with positive SU from corrupting epoch aggregates (division by zero in `W_total_epoch`, stuck finalization, or silent mis-minting). **Late-receipt exemption:** This invariant does NOT apply to late receipts (epoch already finalized). Late receipts store `weight_q = 0` and skip all SU/weight accumulation (see "Late receipt handling" below), so the floor check is bypassed — the receipt does not participate in epoch aggregates and cannot cause the corruption the floor guards against.
- store receipt reward metadata keyed by `receipt_id` (epoch, weight, eligibility, claimed flag)
- `epochSU[shell_id][epoch] += su_delivered`
- compute `shell_reward_eligible` from on-chain state (bond, age, not unbonding, lagged uptime)
- compute `ghost_reward_eligible` from on-chain state (bond >= `B_ghost_reward_min`, age >= `T_ghost_age`, not unbonding; Section 7.6.2 (Part 2))
- derive combined eligibility: `receipt_reward_eligible = shell_reward_eligible AND ghost_reward_eligible`

- **Per-shell eligible SU cap (normative):** track `eligible_SU_shell[shell_id][epoch]`. If `eligible_SU_shell[shell_id][epoch] + su_delivered > SU_cap_per_shell`, set `receipt_reward_eligible = false` for this receipt (the Shell has already contributed its maximum eligible SU for this epoch). The receipt still finalizes and rent settles normally, but the excess SU does not count toward emissions. This forces farming capital to scale with bonded Shell count rather than Ghost ID count (see Part 2, Section 7.8). **Design note on cliff behavior:** the cap is intentionally a hard cliff rather than partial eligibility. Partial eligibility (crediting only `cap_remaining` SU) adds complexity (fractional receipt splitting, non-deterministic claim amounts depending on receipt ordering) for marginal benefit. The hard cliff is predictable: Ghosts can query `eligible_SU_shell[shell_id][epoch]` and avoid saturated Shells. The externality (a Ghost losing eligibility because other sessions consumed the cap) is mitigated by `SU_cap_per_shell` being set well above typical single-Shell utilization — it is an anti-farming cap, not a normal-operation constraint.
- if `receipt_reward_eligible = true`:
  - `eligible_SU_shell[shell_id][epoch] += su_delivered`
  - `SU_eligible_epoch[epoch] += su_delivered`
  - `W_total_epoch[epoch] += weight_q`

Emissions amplification prevention (normative): `SU_eligible_epoch` MUST only include SU from receipts where **both** Shell and Ghost are reward-eligible. If Ghost-ineligible receipts were included in `SU_eligible_epoch`, an attacker could inflate utilization `u_total` (and thus total emissions) by running many ineligible Ghost sessions on eligible Shells, while those receipts contribute `weight_q = 0` and therefore do not dilute `W_total_epoch`. The combined eligibility gate above closes this vector.

Late receipt handling (normative): If `recordReceipt` is called for an epoch `e` that has already been finalized (`finalized[e] = true`), the call MUST NOT revert. Instead, it MUST silently skip all weight and SU accumulation and store the receipt with `shell_reward_eligible = false` and `weight_q = 0`. This non-reverting behavior is required so that `ReceiptManager.finalizeReceipt` can still settle rent via `settleEpoch` without needing try/catch around the `recordReceipt` call. The late receipt earns zero emissions but rent settlement proceeds normally. The constraint `EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE > T_max + 1` (Section 10.5.7) is designed to prevent this case under normal operation, but implementations MUST handle it defensively. (See also Part 2, Section 7.14.)

This makes reward accounting linear in the number of finalized receipts, with no epoch-wide iteration.

Epoch finalization: computing fixed rates After the dispute window has closed, anyone MAY call `finalizeEpoch(e)`. The gating condition is `current_epoch >= e + 1 + EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE` (delay counted from epoch end). The combined delay `EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE` MUST exceed `T_max + 1` (Section 10.5.7), ensuring all receipts have finalized and called `recordReceipt` before epoch finalization becomes callable.

Emission exclusion risk and mitigation (normative): An attacker can attempt to push a victim's receipts past the epoch finalization cutoff via dispute friction (DA challenges, candidate disqualification, censorship) to exclude the victim from emissions while retaining rent settlement. The combined `EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE > T_max + 1` constraint is the primary mitigation: under normal operation, all receipts finalize before `finalizeEpoch` becomes callable. However, adversarial dispute extension near `T_max` can create edge cases. `finalizeEpoch(e)` MUST revert if `ReceiptManager.pendingDACount(e) > 0` (see Section 14.5 and 14.6). This O(1) counter-based check prevents attackers from weaponizing DA challenges to exclude specific receipts without requiring iteration over all sessions. Receipts finalized during the grace period are included in `W_total_epoch[e]` and `SU_eligible_epoch[e]` normally. `FINALIZATION_GRACE` is a deployment parameter (recommended: 2 epochs).

On success, it computes utilization and emissions from Part 2:

- scheduled emission `E_sched(e)` (Section 7.3 (Part 2); integer base units, `uint256`)
- pool emissions (two cases to avoid premature truncation):

```
if SU_eligible_epoch[e] >= SU_target:
```

```
    // Full utilization: u_total = 1.0, no rounding needed
    E_ghost(e) = floor(E_sched(e) * beta_bps / 10_000)
    E_shell(e) = floor(E_sched(e) * alpha_bps / 10_000)
else:
    // Partial utilization: single mulDiv avoids premature Q64.64 truncation
    // den = 10_000 * SU_target (precomputable, fits in uint256)
    E_ghost(e) = mulDiv(E_sched(e) * beta_bps, SU_eligible_epoch[e], 10_000 * SU_target)
    E_shell(e) = mulDiv(E_sched(e) * alpha_bps, SU_eligible_epoch[e], 10_000 * SU_target)
```

Where `mulDiv(a, b, d) = floor(a * b / d)` using 512-bit-safe arithmetic (e.g., OpenZeppelin `Math.mulDiv`). This avoids the rounding error that would result from first computing `u_total_q = floor(SU_eligible * Q / SU_target)` and then multiplying — the Q64.64 truncation of `u_total` loses up to 1 ULP (unit in last place), which when multiplied by `E_sched` (up to `~2^192` base units) can amplify to a deviation of up to `E_sched / Q` (up to `~2^128` base units). The single `mulDiv` path avoids this amplification entirely.

For observability, implementations SHOULD also compute and emit `u_total_q = min(Q, SU_eligible_epoch[e] * Q / SU_target)` in the `EpochFinalized` event, but this value MUST NOT be used as the source of truth for mint amounts.

Then define fixed reward rates for the epoch:

- `rate_ghost(e) = E_ghost(e) / W_total_epoch[e]`
- `rate_shell(e) = E_shell(e) / W_total_epoch[e]`

"Fixed reward rates" means that within an epoch, every eligible receipt is paid by multiplying its stored weight by the same pool rate. The rate does not depend on claim order.

If `W_total_epoch[e] = 0`, then `u_total = 0` and both pool emissions are zero.

Minting model (normative): `finalizeEpoch(e)` MUST mint `R_net = E_ghost(e) + E_shell(e)` to the `RewardsManager` contract via `IGIT.mint(address(this), R_net)`. Individual `claimReceiptRewards` calls then transfer from the `RewardsManager` balance to recipients. This is **mint-to-pool-then-transfer**, not mint-at-claim-time. The pool amount is the post-sink `R_net`; the withheld amount `R_withheld` is never minted (Section 10.6, "Sink as mint reduction"). Truncation dust and expired claims remain as `RewardsManager` token balance (see Section 10.7, forfeited rewards accounting).

Implementations SHOULD:

- store `rate_ghost(e)` and `rate_shell(e)` as fixed-point values,
- mark epoch `e` finalized exactly once, and
- reject claims for non-finalized epochs.

Fixed-point arithmetic (normative) This section specifies deterministic arithmetic for on-chain reward computation. Fixed-point is used only for **dimensionless ratios** (utilization, weight multipliers, decay factors). Token amounts are stored and computed in 256-bit integer form (ERC-20 base units).

Fixed-point representation — type rules:

Define `Q = 2^64`. All Q64.64 values represent unsigned fixed-point numbers where the integer part occupies the upper bits and the fractional part occupies the lower 64 bits.

Q64.64 type categories:

- **Bounded dimensionless ratios** (`u_total_q`, `w_passport_q`, `w_dwell_q`, `decay_q`, `s_q`, `r_q`): Q64.64 stored as `uint128`. These are bounded by construction — `decay_q` and `u_total_q` are in `[0, Q]`, `w_passport_q` is in `[Q, 2*Q]`, `w_dwell_q` is in `[2, Q]`. Maximum representable value in `uint128`: `2^128 - 1` (1.8×10^{19} as a real number).
- **Per-receipt weight** (`weight_q`): Q64.64 stored as `uint256`. Although per-receipt weight is bounded (e.g., `65_535 * 2.0 * 1.0 = 131_070.0` in real terms), using `uint256` aligns with the Part 2 interface (`uint256 weight_q`) and avoids unnecessary casts.

- **Aggregates** (`W_total_epoch[e]`, `SU_eligible_epoch[e]`): `uint256`. The sum of many per-receipt weights or SU values can exceed `uint128`.
- **Scaled rates** (`rate_ghost_q`, `rate_shell_q`): Q64.64 stored as `uint256`. These are computed as `token_amount * Q / aggregate` and require the wider type.
- **Intermediate products**: When multiplying two Q64.64 values, the intermediate product requires up to 256 bits. Implementations MUST use `uint256` for intermediate products and divide by `Q` (equivalently right-shift by 64) to obtain the Q64.64 result.

Canonical helper (normative): Define `mulQ64(a, b) = floor(uint256(a) * uint256(b) / Q)`. The division truncates toward zero per the rounding rule below. **Precondition:** `a * b` MUST fit in `uint256` (i.e., `a * b < 2^256`). This is satisfied for all uses in weight computation (where both operands are at most `~131_070 * Q  2^81`, so the product is at most `~2^162`). For products where this precondition may not hold (specifically `rate_*_q * weight_q` at claim time, where both operands are `uint256`), implementations MUST use `mulDiv(a, b, Q)` with 512-bit-safe arithmetic instead of `mulQ64`.

> **Library compatibility note:** The widely-deployed ABDKMath64x64 library uses `int128` (signed) for its 64.64 type. GITS uses **unsigned** arithmetic exclusively. Implementations MAY adapt ABDK routines by restricting inputs to non-negative values and casting results, but MUST NOT rely on signed behavior. Implementations MAY alternatively use a custom unsigned Q64.64 library. The `exp_2` function referenced below MUST accept `uint128` input and produce `uint128` output; its specification follows.

Rounding rule: All fixed-point divisions MUST round **toward zero** (truncation). This ensures deterministic results across implementations and matches EVM integer division semantics.

Token amounts (256-bit integers):

Emission schedule parameters (`E_0`, `E_tail`) and computed emissions are stored and manipulated as `uint256` values in ERC-20 base units (18 decimals for GIT). They MUST NOT be converted to 64.64 fixed-point, as typical emission values exceed the representable range.

Emission schedule computation (integer-safe):

The decay factor `2^{-e/H}` is computed directly (without computing the positive exponent `2^{e/H}` which can overflow):

```
// exponent_q is Q64.64 representation of e/H (uint128, always non-negative)
uint128 exponent_q = uint128(uint256(e) * Q / H);

// decay_q is Q64.64 representation of 2^{-e/H}, computed directly
// neg_exp_2_64x64(x_q) returns floor(Q * 2^{-x}) where x = x_q / Q
// For x_q = 0: returns Q (= 1.0). For large x_q: approaches 0.
// Input range: [0, 63 * Q]. For x_q >= 64 * Q, MUST return 0.
uint128 decay_q = neg_exp_2_64x64(exponent_q);

// E_sched in base units (uint256)
uint256 E_sched = uint256(E_0) * uint256(decay_q) / Q + E_tail;
```

`neg_exp_2_64x64` specification (normative):

`neg_exp_2_64x64(x_q: uint128) -> uint128` computes `floor(Q * 2^{-(x_q / Q)})` (the Q64.64 representation of `2^{-x}` where `x = x_q / Q`).

- Input range: `x_q` in `[0, 64 * Q)`. For `x_q >= 64 * Q`, the result is 0 (decay below representable minimum).
- Output range: `[0, Q]` (corresponding to real values `[0, 1.0]`).
- Precision: MUST match the canonical test vectors below to within ±1 ULP.
- Implementation strategies: (a) compute `2^{integer_part}` via right-shift and `2^{fractional_part}` via polynomial/table approximation, then combine; (b) use an existing library's `exp_2` with signed wrapper and invert; (c) implement the iterative halving algorithm. The choice is left to implementations provided the output matches the precision requirement.

This avoids the 2^{+e/H} overflow problem: the positive exponent 2^{e/H} can exceed uint128 Q64.64 range when e/H >= 64, but the negative exponent 2^{-e/H} is always in [0, 1.0] and fits in uint128.

Where: - Q = 2^64 - E_0 and E_tail are uint256 in base units (e.g., 1_000_000 * 10^18 for 1M GIT) - exponent_q and decay_q are uint128 in Q64.64 format - decay_q is always in range [0, Q] for non-negative epochs

Overflow analysis (complete):

1. E_0 * decay_q <= E_0 * Q. This fits in uint256 for any E_0 < 2^192. **Deployment constraint:** E_0 + E_tail < 2^192 (in base units).
2. E_sched = (E_0 * decay_q) / Q + E_tail <= E_0 + E_tail < 2^192 for all epochs. The + E_tail cannot overflow because both operands are $< 2^{192}$.
3. E_pool * Q (in rate computation): E_pool <= E_sched < 2^192, so E_pool * Q < 2^256. Fits in uint256.
4. rate_*_q * weight_q (at claim time): both are uint256, so the product requires 512 bits. Implementations MUST use mulDiv(rate_q, weight_q, Q) (a 512-bit-safe floor division) or equivalent. Standard EVM libraries (e.g., OpenZeppelin Math.mulDiv) provide this.
5. beta_bps * E_sched(e): beta_bps <= 10_000 and E_sched < 2^192, so the product is $< 2^{206}$. Fits in uint256.

Basis-point computations (normative, global rule): All basis-point payouts of the form floor(X * bps / 10_000) where X is a uint256 amount MUST use mulDiv(X, bps, 10_000) (512-bit-safe floor division) to prevent overflow when X > type(uint256).max / 10_000. This applies to: spend_cap = mulDiv(ER0, bps_recovery_spend_cap, 10_000), bounty_initiator = mulDiv(B_rescue_total, bps_initiator, 10_000), challenger_reward = mulDiv(B_receipt, bps_challenger_reward, 10_000), and all similar computations. For remainder computations (e.g., burn amount after challenger reward), implementations MUST use subtraction from the whole (remainder = X - payout) rather than a separate mulDiv(X, 10_000 - bps, 10_000) to guarantee exact conservation (payout + remainder == X).

Conformance test vectors:

Parameters: E_0 = 1_000_000 * 10^18, H = 1460, E_tail = 10_000 * 10^18

Epoch	decay_q (64.64 hex)	E_sched (base units)	Human-readable
0	0x10000000000000000 (= 1.0)	1_010_000 * 10^18	1,010,000 GIT
1460	0x8000000000000000 (= 0.5)	510_000 * 10^18	510,000 GIT
2920	0x4000000000000000 (= 0.25)	260_000 * 10^18	260,000 GIT

End-to-end reward test vector:

Parameters: E_0 = 1_000_000 * 10^18, H = 1460, E_tail = 10_000 * 10^18, e = 0, alpha_bps = 5000, beta_bps = 5000, SU = 144, SU_target = 2_880_000, passport_bonus = true (assume B_passport = 1.0), c = 1 (first epoch of residency), D = 30.

Derivation (integer-safe, using the mulDiv path since SU < SU_target): * E_sched(0) = 1_010_000 * 10^18 * E_ghost(0) = mulDiv(E_sched(0) * 5000, 144, 10_000 * 2_880_000) = mulDiv(5_050_000_000 * 10^18, 144, 28_800_000_000) = floor(727_200_000_000 * 10^18 / 28_800_000_000) = 25_250_000_000_000_000_ (exactly 25.25 GIT) * E_shell(0) = E_ghost(0) = 25_250_000_000_000_000_000 * Weight: su_q = 144 << 64, w_passport_q = 2 * Q, w_dwell_q = Q (k = min(floor(0/30), 63) = 0) step2 = mulQ64(su_q, 2*Q) = 144 * 2 * Q = 288 * Q (exact, no truncation) weight_q = mulQ64(288 * Q, Q) = 288 * Q (exact, no truncation) * W_total_epoch = 288 * Q (single receipt) * rate_ghost_q = floor(E_ghost * Q / (288 * Q)) = floor(E_ghost / 288) = floor(25_250_000_000_000_000_000 / 288) = 87_673_611_111_111_111 (truncated) * R_ghost = mulDiv(rate_ghost_q, weight_q, Q) = mulDiv(87_673_611_111_111_111, 288 * Q, Q) = 87_673_611_111_111_111 * 288 = 25_249_999_999_999_999_968

Note: With a single receipt, the actual payout is 25_249_999_999_999_999_968 base units — 32 wei less than the pool emission 25_250_000_000_000_000_000. This dust is a consequence of fixed-point truncation

at the rate computation step and is expected. The "lost" dust accumulates in the RewardsManager contract. Conformance implementations MUST reproduce this exact value for this test vector.

Dust policy (normative): Truncation dust that accumulates in `RewardsManager` (the difference between pool emissions and actual payouts) MUST remain in the contract and is NOT withdrawable by any party. There is no sweep function. The dust is an implicit, permanent withholding — economically negligible per epoch but guaranteed non-exploitable. Implementations MUST NOT roll dust into subsequent epoch pools (doing so would make pool sizes non-deterministic across implementations).

Weight computation (64.64, normative):

The receipt weight `W(r) = SU(r) * w_passport * w_dwell` is computed in Q64.64 fixed-point using the `mulQ64` helper:

```
// Step 1: convert SU to Q64.64
uint256 su_q = uint256(SU) << 64;           // SU * Q, exact (no rounding)

// Step 2: multiply by passport bonus
// w_passport_q: uint128, Q (1.0) or Q + B_passport_q (1.0 + B_passport)
uint256 step2 = mulQ64(su_q, uint256(w_passport_q));
// = floor(su_q * w_passport_q / Q), intermediate uses uint256

// Step 3: multiply by dwell decay
// w_dwell_q: uint128, = Q >> k where k = min(floor((c-1)/D), 63)
uint256 weight_q = mulQ64(step2, uint256(w_dwell_q));
// = floor(step2 * w_dwell_q / Q), intermediate uses uint256
```

Where `mulQ64(a, b) = floor(uint256(a) * uint256(b) / Q)` (the canonical helper defined above).

Component bounds: - `SU(r)` is `uint32` (at most `N <= 65_535`) - `w_passport_q` is `Q` (1.0) or `Q + B_passport_q` (at most `2 * Q` for `B_passport = 1.0`), stored as `uint128` - `w_dwell_q = Q >> k` where `k = min(floor((c-1)/D), 63)`. The exponent is capped at 63 to ensure `w_dwell_q >= 2` (always positive). Stored as `uint128`. - `weight_q` is stored as `uint256` (matching Part 2 interface). The per-receipt maximum is `65_535 * 2.0 * 1.0 = 131_070.0` in real terms, which fits comfortably.

Truncation occurs twice (once per `mulQ64` call). This is normative — all implementations MUST apply truncation at each step to produce identical `weight_q` values.

- `recordReceipt` MUST verify: if `SU_delivered > 0` then `weight_q >= MIN_WEIGHT_Q`. If violated, revert.

Scaled rates (fixed-point):

Reward rates are computed and stored as Q64.64 scaled values (`uint256`):

- `rate_ghost_q = floor(E_ghost_base_units * Q / W_total_epoch_q)`
- `rate_shell_q = floor(E_shell_base_units * Q / W_total_epoch_q)`

Both `E_*` and `W_total_epoch_q` are `uint256`. The product `E_* * Q` fits in `uint256` because `E_* < 2^192` (overflow analysis item 3 above).

At claim time: `R_ghost = mulDiv(rate_ghost_q, weight_q, Q)` using 512-bit-safe floor division, since `rate_ghost_q * weight_q` can exceed `uint256`. Similarly for `R_shell`.

Claiming `claimReceiptRewards(receipt_id)`:

1. loads `(epoch, weight_q, shell_reward_eligible, claimed)` for the receipt,
2. rejects if `claimed = true` or if epoch `epoch` is not finalized,
3. if `shell_reward_eligible = false`, sets `R_ghost = R_shell = 0` (ineligible receipts earn zero emissions; the receipt may still have settled rent via escrow) and skips to step 5,
4. computes gross rewards using 512-bit-safe floor division (since `rate_*_q * weight_q` can exceed `uint256`):

- `R_ghost = mulDiv(rate_ghost_q[epoch], weight_q, Q)`
- `R_shell = mulDiv(rate_shell_q[epoch], weight_q, Q)`

5. applies any configured sinks (for example adaptive burn, Section 7.9.1 (Part 2); see integer-safe computation below),
6. looks up current recipients: Ghost rewards are sent to the Ghost's registered `wallet` address (from `GhostRegistry`); Shell rewards are sent to the Shell's current `payout_address` (from `ShellRegistry`). **Recipient lookup is at claim time, not at recordReceipt time.** This means a Shell that updates its payout address between service and claim will receive rewards at the new address.
7. transfers the resulting rewards and marks `claimed = true`.

Effective weight gating: The `shell_reward_eligible` flag is stored per receipt at `recordReceipt` time. When `shell_reward_eligible = false`, the receipt's weight was **not** accumulated into `W_total_epoch[e]`, so applying the rate formula would overpay relative to the pool. The claim path MUST therefore gate on this flag to ensure pool conservation: `sum_r(R_ghost(r)) <= E_ghost(e)` and `sum_r(R_shell(r)) <= E_shell(e)` for all eligible receipts.

Adaptive sink: integer-safe computation (normative) At claim time, the adaptive burn (Section 7.9.1 (Part 2)) is applied as follows. All intermediate values are Q64.64 unless noted:

```
// Emission decay factor s(e) in Q64.64:
// s(e) = clamp01(1 - E_sched(e) / E_sched(0))
s_q = Q - min(Q, E_sched(e) * Q / E_sched(0))

// Utilization ramp r(u) in Q64.64:
// r(u) = clamp01((u_total - u_sink_start) / (u_sink_full - u_sink_start))
// where u_sink_start and u_sink_full are stored as Q64.64
if u_total_q <= u_sink_start_q:
    r_q = 0
else:
    r_q = min(Q, (u_total_q - u_sink_start_q) * Q / (u_sink_full_q - u_sink_start_q))

// Adaptive sink in basis points (uint256):
bps_sink = bps_sink_max * s_q / Q * r_q / Q
// Equivalently: bps_sink = floor(floor(bps_sink_max * s_q / Q) * r_q / Q)
// Two-step division avoids uint256 overflow from triple multiplication.

// Apply to gross reward (uint256 base units):
R_withheld = floor(R_gross * bps_sink / 10_000)
R_net = R_gross - R_withheld
// RewardsManager mints ONLY R_net. R_withheld is never created.
```

Sink as mint reduction (normative): The adaptive sink is implemented as a **mint reduction**, not as a mint-then-burn. `RewardsManager` computes `R_withheld` and mints only `R_net = R_gross - R_withheld`. The withheld amount is never minted, so `totalSupply` reflects actual circulating supply at all times. This avoids dependence on ERC-20 burn semantics (many implementations revert on `transfer(address(0))` or do not reduce `totalSupply`).

The two-step division for `bps_sink` is the canonical form. Note: with `bps_sink_max <= 10_000` and `s_q, r_q <= Q`, the single product `bps_sink_max * s_q * r_q <= 10_000 * Q^2 < 2^142` and does not actually overflow `uint256`. The two-step form is still preferred for clarity and to match the normative rounding rule (two truncations, not one), ensuring deterministic results across implementations.

10.7 Practical gas and feasibility notes

GITS is intentionally optimistic: the chain is a dispute court, not the primary execution environment. Still, several operations have nontrivial on-chain cost.

Implementation notes:

- **Receipt fraud proofs are logarithmic:** the Merkle-sum proof depth is `log2(N_PAD)`. For example, with `N_PAD = 256` the proof depth is 8, so a single fraud proof carries 8 sibling nodes (hash + sum) plus one leaf.
- **Receipt-DA gas feasibility (normative):** `publishReceiptLog` writes `N_PAD` leaf entries on-chain and the contract recomputes `(log_root', SU_root')` from the published data. This is O(`N_PAD`) hashing work (all leaves plus `N_PAD - 1` internal nodes), plus calldata costs for the bitmap and signature pairs. Deployments MUST set `N_PAD <= N_PAD_MAX_EVM` where `N_PAD_MAX_EVM` is the maximum `N_PAD` for which `publishReceiptLog` stays within the target chain's block gas limit. `N_PAD_MAX_EVM` is a deployment constant (see parameter table). Exceeding it makes DA challenges infeasible; the protocol cannot function if DA publication is too expensive to fit in a block. **EVM calldata sizing (normative):** For EVM deployments, the worst-case calldata size for `publishReceiptLog` is approximately `ceil(N_PAD / 8)` bytes (bitmap) + `N_PAD * (sig_ghost_len + sig_shell_len)` bytes (signatures). With K1 signatures (65 bytes each) and `N_PAD = 2048`, this is ~256 + 266,240 266 KB of calldata, which at 16 gas/byte is ~4.3M gas for calldata alone, plus ~2048 * ~600 = ~1.2M gas for hashing. Total ~5.5M gas fits in a 30M gas block. For `N_PAD = 4096` or larger, the gas cost doubles and approaches block limits. The recommended `N_PAD_MAX_EVM = 2048` provides a 5× safety margin on most EVM chains. Deployments requiring `N > 2048` intervals per epoch MUST use alternative DA verification (e.g., blob DA with a succinct verification path) rather than increasing `N_PAD_MAX_EVM`.
- **Signature verification dominates (normative):** K1 (`ecrecover`) costs ~3,000 gas per verification. R1 (`P256VERIFY` at `0x100`) costs ~3,450 gas where the precompile exists [14][15], but a Solidity-only P-256 verifier costs 200k–500k gas per call — making receipt fraud proofs (which verify up to `N_PAD` signatures) infeasible. Therefore: R1 MUST only be included in `SUPPORTED_SIG_ALGS` when the target chain provides a native or precompiled P-256 verifier with gas cost 10k per call. This is enforced at deployment (see constraint above). Deployments on chains without a P-256 precompile MUST restrict `SUPPORTED_SIG_ALGS` to K1 only.
- **Verifier certificate checks must stay bounded:** if certificate validity requires verifying `k` verifier signatures, keep `k` small (or use an aggregated threshold signature scheme) so `ShellRegistry` and recovery paths remain affordable.
- **Recovery is rare by design:** recovery paths can be more expensive than normal operations, but they SHOULD remain feasible on the target chain. Rate limits and bonds (Section 12.4) keep worst-case load bounded.

Reward state growth and pruning (normative): Per-epoch reward accounting grows linearly. `W_claim` (deployment parameter, see parameter table) defines the expiry window after which unclaimed rewards are forfeited and storage is prunable.

Expiry rules: * `claimReceiptRewards(receipt_id)` MUST revert if `current_epoch > receipt_epoch + W_claim`. * Per-receipt metadata (`epoch`, `weight_q`, `shell_reward_eligible`, `claimed`) is prunable after the receipt's epoch exceeds `W_claim`. * Per-epoch aggregates (`rate_ghost_q[e]`, `rate_shell_q[e]`, `W_total_epoch[e]`, `SU_eligible_epoch[e]`, `finalized[e]`) are prunable after `current_epoch > e + W_claim`.

Pruning mechanism: Implementations SHOULD expose permissionless `pruneEpoch(epoch)` and `pruneReceipt(receipt_id)` functions callable by anyone after the expiry window. These functions zero the associated storage slots to reclaim gas refunds. Implementations MAY also use lazy deletion: when any claim or `recordReceipt` touches an epoch older than `W_claim`, it MAY zero the associated storage slots opportunistically.

Forfeited rewards accounting (normative): Two distinct mechanisms produce unredeemed tokens:

- **Truncation dust** (from fixed-point rounding at `finalizeEpoch`): Pool emissions are minted to `RewardsManager` at finalization time. The sum of individual claim payouts may be less than the minted pool amount due to `floor()` truncation. This dust remains as token balance in `RewardsManager` and is non-withdrawable (Section 10.6, dust policy).
- **Expired claims** (from unclaimed receipts past `W_claim`): The tokens for these receipts were already minted (as part of the epoch pool) but are never transferred out. They also remain as token balance in `RewardsManager` and are effectively burned (not redistributable).

Both mechanisms result in tokens held by `RewardsManager` with no withdrawal path. This is consistent with the mint-reduction model: the adaptive sink (Section 10.6) reduces the amount minted at `finalizeEpoch`, while dust and expirations are residuals of amounts that *were* minted but never claimed.

Practical recommendation: deploy the full feature set on an L2/L3 with cheap calldata and the needed precompiles, and keep the L1 footprint minimal.

11. Off-chain protocol specification

11.1 Heartbeats and logs

GITS meters service in fixed-length intervals.

Schedule (parameterized):

- `EPOCH_LEN` is the epoch length.
- `Delta` is the service interval length (heartbeat cadence).
- `N = EPOCH_LEN / Delta` is the number of intervals per epoch (MUST be an integer).
- interval indices: `i in {0, 1, ..., N-1}`.

To avoid clock drift and timestamp ambiguity, interval validity is defined by **interval index**, not by local wall-clock timestamps. Parties derive the current `(epoch, i)` from chain time (latest observed block timestamp) and co-sign the same index.

11.1.1 Heartbeat message All tag-hash computations in Part 3 use the canonical rule from Part 1 Section 4.5.3: `H("TAG" || args)` means `keccak256(abi.encode(TAG_HASH, args))` where `TAG_HASH = keccak256(bytes("TAG"))`.

For each interval `i`, define the canonical heartbeat digest:

```
HB = H("GITS_HEARTBEAT" || chain_id || session_id || epoch || i)
```

Signatures:

- `sig_ghost = Sign(session_sk, HB)`
- `sig_shell = Sign(shell_session_sk, HB)`

The corresponding public keys and signature algorithms `(ghost_session_key, shell_session_key)` are recorded in `SessionManager` at `SessionOpen`.

11.1.2 Interval records An interval record leaf contains:

- `session_id`
- `epoch`
- `interval_index = i`
- `sig_ghost` (or empty)
- `sig_shell` (or empty)

An interval is billable if and only if both signatures are present and verify against the session public keys for the same `HB`.

11.1.3 Logs and practical anti-griefing rules Records are appended off-chain into a fixed-size array of `N` leaves for the epoch and committed as a Merkle root in the on-chain receipt (Section 10.5).

To prevent signature-withholding griefing:

- A Shell SHOULD gate continued service on receiving the Ghost's heartbeat signature for the next interval.
- A Ghost SHOULD treat missing Shell co-signatures as non-delivery and begin an exit plan (close at epoch end, or migrate if possible).

These are off-chain runtime rules. The on-chain consequence is simple: unsigned intervals are not billable.

11.2 Transport and attested channels

Large artifacts (migration bundles, checkpoint ciphertexts, and recovery envelopes) are transported off-chain.

Normative integrity rule:

- Any off-chain artifact that affects on-chain behavior MUST be **content-addressed** by a keccak256 hash that is either (a) committed on-chain, or (b) included in a signed artifact whose hash is committed on-chain.
- Peers MUST reject artifacts whose computed hash does not match the committed hash.

Confidentiality rule by tier:

- On Standard hosts, transport confidentiality protects against network observers but **not** against the Shell operator. Assume the host can observe plaintext.
- On Confidential hosts, attested channels are used to reduce host visibility and to make key shielding plausible.

11.2.1 Attested TLS binding (reference pattern) For interactive transports (live migration, share collection, and any channel that carries secrets), implementations SHOULD use TLS that is *bound* to the attested runtime.

A reference pattern:

1. The sender generates an ephemeral TLS keypair (or an ephemeral signing key used to authenticate the TLS key).
2. The sender includes `H(pk_tls)` (or `H(pk_sign)` that authenticates `pk_tls`) in the attestation report data.
3. The receiver verifies the attestation evidence and pins the TLS session to the attested key commitment.
4. The receiver enforces freshness (nonce or timestamp inside the attested report data, plus an expiry window).

This prevents a network attacker from man-in-the-middling the channel without also forging attestation evidence.

11.2.2 Replay protection and transcript commitments Interactive messages SHOULD include:

- `(chain_id, ghost_id, session_id, epoch)` context fields, and
- a per-session monotonic counter or nonce.

Where feasible, messages SHOULD be signed by the relevant session key (`SK_g` or `SK_s`) and the receiver SHOULD maintain a replay cache per `(session_id, epoch)`.

For migration, a **Migration Manifest** SHOULD commit to:

- `bundle_hash` (hash of the serialized migration bundle),
- `checkpoint_commitment` (if a checkpoint is embedded),
- `dest_shell_id`, and
- a manifest signature by the Ghost Identity Key.

This makes migration artifacts independently verifiable even when transported through untrusted relays.

11.3 Indexers

Offer indexers are permissionless processes that:

- ingest signed offers
- validate signatures and Shell registration (including bond status)
- rank by price, reliability, and policy compatibility

Because common ownership and infrastructure concentration are not reliably detectable on-chain, indexers SHOULD also rank and annotate offers using decentralization-relevant signals that are observable off-chain, such as:

- Shell age and bond tier (hosting-only vs reward-eligible)
- diversity hints (independent ASNs, geographic spread, and long-lived Shell identities)
- concentration warnings (for example, when results are dominated by a single provider or region)

GITS assumes multiple independent indexers. If indexers are censored, Ghosts can fall back to their `allowedShells` set, `homeShell`, or Safe Haven recovery (Section 13.2).

11.4 Off-chain artifact formats (canonical JSON)

Off-chain artifacts MUST be unambiguous to sign and verify. The reference encoding is:

- Canonical JSON (RFC 8785, JSON Canonicalization Scheme) encoded as UTF-8 [16], and
- signature over `H(canonical_bytes)` with a domain-separated tag.

A practical signing wrapper for any artifact `A`:

- `payload_bytes = CanonicalJSON(A_without_sig)` — the canonical JSON encoding (RFC 8785) of the artifact with the `sig` field removed.
- `payload_hash = keccak256(payload_bytes)`
- `artifact_type_hash = keccak256(bytes(artifact_type))` — converts the variable-length artifact type string to a fixed-width `bytes32`.
- `digest = keccak256(abi.encode(keccak256(bytes("GITS_ARTIFACT")), chain_id, artifact_type_hash, payload_hash))`
- `sig = Sign(key, digest)`

Where `artifact_type` is an ASCII string constant (for example `"OFFER"`, `"CAPABILITY_STATEMENT"`, `"MIGRATION_MANIFEST"`). Each artifact type MUST use a distinct string. This follows the paper's standard digest convention (Section 4.5.3 (Part 1)): `abi.encode` with fixed-width types, domain separation via `keccak256(bytes("TAG"))`, and no packed encodings. Offers use the EIP-712 scheme from Section 13.1 instead of this wrapper; this wrapper applies to Capability Statements, Migration Manifests, and any future off-chain artifact types.

11.4.1 Capability Statement (Shell -> world) Capability Statements are used for discovery and for off-chain policy checks. They are signed by the Shell's **Offer Signing Key**.

Minimum fields:

- `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` (hash of the current on-chain certificate record)
- `endpoints` (transport endpoints)
- `expires_at_epoch`
- `sig`

11.4.2 Offer (Shell -> market) The canonical Offer struct and signing scheme is defined in **Section 13.1**. This section does not define additional offer fields or signing rules. All offer signing and verification MUST follow the EIP-712 typed-data scheme in Section 13.1.

Capability binding (normative): The `capability_hash` field in the Offer struct (Section 13.1) is `keccak256(CanonicalJSON(CS_without_sig))` — the hash of the Capability Statement's canonical JSON encoding **with the `sig` field removed** (Section 11.4.1, Section 11.4). Clients MUST verify `capability_hash` matches the Shell's current on-chain anchored capability hash in `ShellRegistry`. This binds the offer to a specific capability snapshot and prevents bait-and-switch (Section 5.1 (Part 1)).

11.4.3 Migration Manifest (Ghost -> dest Shell / relays) A Migration Manifest binds a migration bundle to an intended destination.

Minimum fields:

- `schema`: `"gits.migration_manifest.v1"`
- `ghost_id`
- `from_shell_id` and `to_shell_id`
- `session_id_from` (optional) and `session_id_to` (optional)
- `bundle_hash`
- `checkpoint_commitment` (optional)
- `sig` (by the Ghost Identity Key)

11.5 Indexer threat model and client hardening

Indexers are not trusted and may be malicious or censored.

Threats:

- **Eclipse:** an indexer shows only attacker-controlled offers to trap a Ghost.
- **Equivocation:** an indexer returns different views to different clients.
- **Staleness:** an indexer serves expired offers or offers from unbonded / unregistered Shells.
- **Sybil flooding:** an attacker creates many Shell identities to dominate rankings.

Client hardening recommendations:

- Query multiple independent indexers and merge results. Clients SHOULD treat "only one indexer available" as a risk signal.
- Verify signatures on Offers and Capability Statements and verify the Shell's on-chain registration, bond status, and certificate freshness.
- Enforce local policy: destination allowlists, minimum Shell age/bond, and tier requirements.
- Prefer diverse results (operators, ASNs, regions) when safety matters, and surface concentration warnings to the Ghost.
- Maintain a local cache of recently-seen Shell identities and treat sudden appearance of many new Shells as suspicious unless bonded and aged.

These defenses are complementary to on-chain safety mechanisms (leases, tenure caps, trust-refresh, and recovery).

12. Liveness, revival, and Safe Havens

12.1 Checkpoints

Each epoch, the Ghost produces an encrypted checkpoint that is sufficient to restart after host loss.

Checkpoint plaintext SHOULD include:

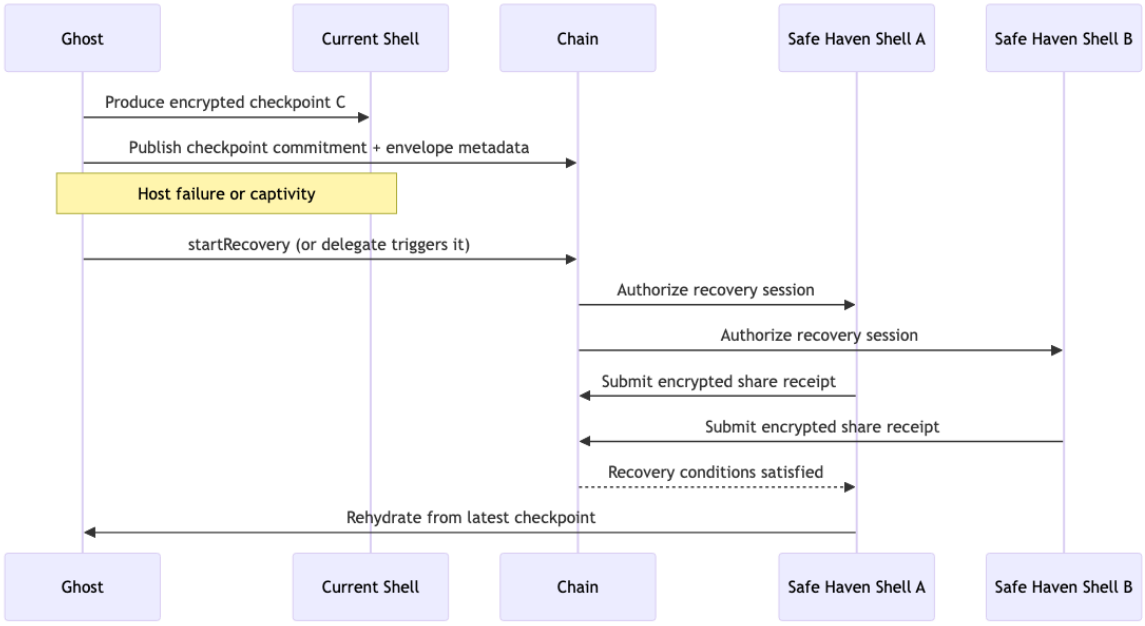
- agent state and configuration (including Wallet Guard state)
- references to external blobs (for example a memory manifest)
- excludes secrets that should not persist (for example ephemeral session keys)

12.1.1 Portable checkpoint encryption (no host-sealed keys) Checkpoint encryption MUST be recoverable off-host. It MUST NOT depend on a machine-sealed key tied to the current host.

For each checkpoint, the Ghost generates a fresh random data-encryption key `K_ckpt` and computes:

- `C = AEAD_Encrypt(K_ckpt, checkpoint_plaintext, aad=(ghost_id, epoch, checkpoint_version))`

The Ghost then constructs a **Recovery Envelope** that allows `t-of-n` Safe Havens to reconstruct `K_ckpt` inside a recovered confidential VM.



Recovery flow: checkpoint and resurrection

Let the configured Recovery Set be `RS = [shell_id_1 ... shell_id_n]` with threshold `t`. Each Safe Haven `shell_id_j` has a registered recovery public key `pk_recovery_j` in `ShellRegistry`.

1. Split `K_ckpt` into `n` Shamir shares with threshold `t`:

- `(share_1 ... share_n) = SSS_Split(K_ckpt, t, n)`

2. Encrypt each share to its Safe Haven:

- `E_j = HPKE_Seal(pk_recovery_j, share_j, aad=(ghost_id, epoch, H(C)))`

3. Define the Recovery Envelope:

- `Env = [EnvEntry_j for j in 1..n]`, where each entry is a fully specified struct:

```
struct EnvEntry {
    bytes32 shell_id;         // Recovery Set member
    bytes32 pk_recovery_hash; // keccak256(pk_recovery) for the recipient
    bytes   hpke_ciphertext;  // encrypted Shamir share
}
```

Entries MUST be sorted by `shell_id` (ascending) and unique. `pk_recovery_hash = keccak256(pk_recovery)` where `pk_recovery` is the Shell's registered recovery public key.

The Ghost uploads `{C, Env}` to content-addressed storage (either as a single bundle or as two objects).

On-chain, the Ghost publishes:

- `checkpoint_commitment = keccak256(C)`
- `envelope_commitment = keccak256(abi.encode(keccak256(bytes("GITS_ENV")), abi.encode(Env)))` — canonical envelope commitment with domain separation. `abi.encode(Env)` encodes entries in sorted order using the struct field types above.
- one or more storage pointers for `C` and `Env` (recommended: at least two independent pointers, Section 12.1.3)

Notes:

- Any `t` Safe Havens can supply enough shares to reconstruct `K_ckpt` inside a recovered confidential VM, enabling Safe Haven revival without cooperation from the failed host.
- A single Safe Haven learns only its Shamir share. Reconstruction requires threshold participation.
- On Standard Shells, the host may still observe live memory. Checkpoint encryption primarily protects against storage leakage and supports portable recovery. Confidentiality of the active runtime is provided only by Confidential Shells with valid attestation.

Shamir share wire format (normative): Shamir secret sharing MUST use GF(2^8) with the irreducible polynomial `x^8 + x^4 + x^3 + x + 1` (0x11B), consistent with SLIP-39. Each share is a byte array of the same length as the secret. The share index `j` (1-indexed, corresponding to Recovery Set position) is the GF(2^8) evaluation point. The wire format for a single share is: `share_bytes = [index_byte || share_data]` where `index_byte = uint8(j)` and `share_data` is the evaluated polynomial at point `j`. This encoding is compatible with SLIP-39 field arithmetic. Deployments MUST publish the complete share format (this wire layout, threshold `t`, and `n`) as part of the deployment manifest so that independent Safe Haven implementations can reconstruct secrets interoperably.

Deployment manifest (normative requirements): Each deployment MUST publish a deployment manifest containing at minimum: (1) chain ID, (2) contract addresses for all protocol contracts, (3) all deployment-constant parameter values from Section 10.0, (4) the Shamir share format above, (5) the `SUPPORTED_SIG_ALGS` set. The manifest MUST be content-addressed (e.g., published to IPFS or similar) and its hash SHOULD be committed on-chain in a deployment-wide registry or emitted as an event at contract deployment. The manifest schema and authentication mechanism are deployment-specific and out of scope for this specification, but the manifest MUST be sufficient for an independent client to locate and verify all protocol contracts and parameters without trusting a centralized discovery service.

Implementation note: Shamir secret sharing plus HPKE is the simplest portable design. A stronger but more complex variant is true threshold public-key encryption (no party ever holds a full decryptable share). A weaker but simpler variant is encrypting `K_ckpt` independently to each Safe Haven and requiring only one re-wrap at recovery time.

12.1.2 HomeShell mirroring (optional) A Ghost MAY designate a `homeShell` at birth as a last-resort availability anchor (Section 5.5.2 (Part 1)). When enabled, the Ghost SHOULD transmit each checkpoint bundle `{C, Env}` (ciphertext + envelope) to the homeShell for redundant storage.

Properties:

- The homeShell learns only ciphertext; it does not gain decryption capability unless it is also in the Recovery Set.
- HomeShell mirroring is an availability strategy, not a confidentiality claim.
- A Ghost can disable or rotate `homeShell` at any time. Removing `homeShell` is tightening and MUST be immediate; adding a new `homeShell` is loosening and MUST follow the timelocked Trusted Execution Context rule (Section 5.5.2 (Part 1)).

This mechanism is optional. It provides a practical "known place to look" for recovery artifacts if public storage pointers fail, without requiring the homeShell to be a special on-chain authority.

12.1.3 Checkpoint-DA: off-chain or on-chain publication (Ghost-selected) Checkpoint confidentiality is irrelevant if the ciphertext cannot be fetched when needed. GITS therefore treats checkpoint artifact availability ("Checkpoint-DA") as an explicit systems problem.

A Ghost MAY choose either publication mode per checkpoint, based on its own risk tolerance and cost allowance:

Mode A: Off-chain multi-publish (default)

- The Ghost publishes the checkpoint bundle `{C, Env}` to at least `M_publish_min` independent storage backends (for example IPFS + HTTPS object storage + an archival backend).
- The Ghost records at least `M_ptr_min` retrievable pointers for `{C, Env}` on-chain (via `ptrCheckpoint` and `ptrEnvelope`) so a recovery agent can locate the bundle even if one backend disappears.

- The Ghost SHOULD keep at least one copy under Ghost control (or a Ghost-operated key) so availability does not depend solely on Shell operators.

Mode B: On-chain publication (optional, deployment-specific)

A deployment MAY optionally support publishing an encrypted checkpoint bundle `{C, Env}` (or a smaller "escape snapshot") directly into an on-chain publication channel (for example calldata, blobs, or a chain-specific data-publication system). In this mode:

- The Ghost publishes the ciphertext bytes on-chain and still anchors integrity with `checkpointCommitment = H(C)` and `envelopeCommitment = H(Env)`.
- The Ghost records a canonical on-chain reference in `ptrCheckpoint` / `ptrEnvelope` (for example a tx hash, log index, or blob reference), or leaves off-chain pointers empty if the deployment provides an unambiguous on-chain locator.

On-chain publication improves robustness against off-chain storage failures, but it is expensive and may inherit chain-specific retention limits (for example if "blob" data is not permanently retrievable). A practical policy is hybrid: off-chain every epoch, and on-chain only every `k` epochs or only for escape context.

Safe Haven checkpoint mirroring (optional, reputational):

- Safe Haven Shells in `RS` MAY offer a checkpoint mirroring service: "we retain the last `K_mirror` checkpoint bundles for ghosts that choose us."
- This paper assumes no in-protocol slashing for missing off-chain blobs. Mirror performance is enforced by reputation and by Ghost choice (rotate the Recovery Set).

12.2 Recovery delegation (pre-authorization)

Because a fully isolated Ghost cannot sign, recovery must be pre-authorized.

Each Ghost configures a **Recovery Set** `RS` of `n` Safe Haven `shell_id`s and a threshold `t`. The Recovery Config also commits a protocol-level Rescue Bounty (Section 12.2.1). Each Safe Haven in `RS` maintains a registered recovery public key in `ShellRegistry` that is used to encrypt checkpoint key shares (Section 12.1).

Ghosts SHOULD choose `RS` to reduce correlated failure: include Safe Havens across different operators and infrastructure, and avoid concentrating the set in a single cloud provider, ASN, or jurisdiction. The protocol cannot enforce common ownership on-chain, but indexers and clients can surface diversity signals for informed selection (Section 11.3).

Recovery actions are limited on-chain by the Ghost smart wallet policy and protocol state:

- `SessionManager` can enter a `RECOVERY` mode for a stranded `ghost_id` after lease expiry.
- In `RECOVERY` mode, the GhostWallet (smart contract) MUST allow only protocol-defined actions:
  - `startMigration(...)` / `finalizeMigration(...)` to move the revived Ghost to a normal session
  - paying rent (subject to the emergency price cap) to Safe Haven escrows
  - paying the Rescue Bounty via `payRescueBounty(...)` upon successful `recoveryRotate` (Section 12.2.1)
- Arbitrary transfers are disabled by the wallet contract in `RECOVERY` mode.
- `RECOVERY` mode does not auto-expire. It ends only when the Ghost explicitly calls `exitRecovery`, and only from a Trusted Execution Context (Section 5.5.2 (Part 1)).

In `RECOVERY` mode, **policy loosening is disabled**: the wallet MUST reject any loosening (including allowlist expansion) until recovery is exited. Tightening remains allowed.

To reduce "recovery-drain" attacks (repeated emergency sessions to siphon funds), implementations MUST enforce:

- a per-epoch recovery spend cap (rent + bounty) derived from the enforced escape reserve (see Section 12.6 for the normative accounting rule), and
- a cooldown `T_recovery_cooldown` that rate-limits starting new recovery attempts for the same `ghost_id` (see Section 12.6).

A Wallet Guard module MAY mirror these restrictions for usability, but the enforcement is the on-chain wallet policy.

12.2.1 Protocol rescue bounty (always payable from the Ghost) Recovery is a public good for the Ghost: it requires off-chain work (booting a recovery VM, fetching checkpoint artifacts, decrypting shares, and coordinating verifier certificates). The protocol therefore defines an explicit protocol-level **Rescue Bounty** that is always payable from the Ghost itself when the Ghost is in `RECOVERY` mode.

At Ghost birth (or whenever the Recovery Config is updated), the Ghost commits:

`RescueBounty = (asset_bounty, B_rescue_total, bps_initiator)`

Where:

- `asset_bounty` is an accepted stable asset used for recovery payments.
- `B_rescue_total` is the maximum bounty budget for a single successful recovery attempt, denominated in `asset_bounty`. Clients MAY choose `B_rescue_total` as a percentage of their stable escape reserve, but it is stored on-chain as an absolute amount.
- `bps_initiator` is the share (in basis points) of `B_rescue_total` paid to the `startRecovery` initiator. The remainder is reserved for Safe Haven share contributors.

Deployment note: deployments choose an accepted stable `asset_bounty` for recovery payments (for example USDC on Base).

Wallet reservation (enforced):

- The GhostWallet MUST treat `B_rescue_total` as part of the `escapeStable` floor via `bounty_escrow_remaining` (Section 5.5.4 (Part 1)). Outside RECOVERY, `bounty_escrow_remaining = B_rescue_total` and the amount is not spendable for normal operations. During RECOVERY, `bounty_escrow_remaining` decreases as `payRescueBounty` disburses funds, dynamically lowering the floor in lockstep with payouts.

Payout rule (enforced):

- `B_rescue_total` is paid only on a successful `recoveryRotate` for the current recovery attempt.
- Recipients are restricted to:
  1. the `startRecovery` initiator (a Safe Haven Shell), and
  2. each Safe Haven in the Ghost's Recovery Set that is credited as having contributed a valid key share to the successful attempt.
- All bounty payments are made to each Shell's payout address recorded in `ShellRegistry` at the time of `recoveryRotate`.

Routing note: `SessionManager.recoveryRotate(...)` SHOULD compute the recipient set from the included Share Receipts and trigger the bounty payout by calling `GhostWallet.payRescueBounty(...)` internally as part of the same transaction. `payRescueBounty` is not intended as a separate manual post-step.

Contribution proof (Share Receipts):

Because the protocol cannot directly observe off-chain share delivery, the protocol uses an explicit receipt format that is acknowledged by the measured recovery runtime.

For a recovery attempt with identifier `attempt_id`, define:

`ShareReceipt_j = (ghost_id, attempt_id, checkpoint_commitment, shell_id_j, sig_shell_j, sig_ack_j)`

Where:

- `sig_shell_j = Sign(shell_identity_sk_j, H("GITS_SHARE" || chain_id || ghost_id || attempt_id || checkpoint_commitment))`
- `sig_ack_j = Sign(sk_new, H("GITS_SHARE_ACK" || chain_id || ghost_id || attempt_id || checkpoint_commitment || shell_id_j))`

`sig_shell_j` MUST verify under the Shell Identity Key recorded for `shell_id_j` in `ShellRegistry`.

`sig_ack_j` MUST be produced inside the recovery VM and is valid only if `pk_new` is bound to the measured recovery runtime by a valid Recovery Boot Certificate (RBC) included in `recoveryRotate` (Section 12.3).

On-chain, `recoveryRotate` MUST:

- verify `sig_shell_j` and `sig_ack_j` for each included receipt,
- verify `shell_id_j` is in the **snapshotted** Recovery Set for the attempt (caller provides `rs_list`; contract verifies `keccak256(abi.encode(rs_list)) == attempt.rs_hash` and checks membership against that list, consistent with the AuthSig membership check),
- treat duplicate `shell_id_j` receipts as one contribution, and
- cap the number of paid contributors at `n` (the Recovery Set size).

Splitting (deterministic):

- `bounty_initiator = floor(B_rescue_total * bps_initiator / 10_000)` is paid to the initiator.
- The remaining `bounty_contrib = B_rescue_total - bounty_initiator` is split equally across the unique contributor set `C` derived from the Share Receipts.
- Any remainder (integer division dust or unused budget if fewer contributors are credited) remains in the GhostWallet escape reserve.

This Rescue Bounty ensures there is always a protocol-defined reason to perform recovery work, without introducing any central protocol treasury.

Client sizing guidance (non-normative):

- `B_rescue_total` should cover: (i) the on-chain gas paid up-front by the initiator to run `startRecovery` and `recoveryRotate`, (ii) emergency-priced recovery rent at the cap for enough time to complete decryption and coordination, and (iii) a meaningful incentive for `t` share contributors.
- `bps_initiator` should be large enough that a Safe Haven is willing to initiate recovery (which requires the most up-front work and liveness risk), while leaving enough budget to reward share contributors.

Because these costs are chain- and market-dependent, this paper intentionally does not propose concrete numeric defaults. Deployments should publish a parameter manifest (Section 7.7 (Part 2)) and re-run the numerical examples in Section 8 (Part 2) under current economics.

12.3 Recovery flow

Trigger: `lease_expired` or `tenure_expired` for `R` epochs.

Recovery has two goals:

1. decrypt the most recent checkpoint without the failed host, and
2. rotate on-chain control to a key that is verifiably generated inside the measured recovery runtime.

Flow:

1. Any Safe Haven Shell **in the Ghost's Recovery Set** with an active Safe Haven bond MAY submit `startRecovery(ghost_id)` with bond `B_start` posted in the chain base asset (`asset_start`, ETH on Base). Only RS members can initiate recovery because only they possess the encrypted Shamir shares needed to reconstruct the checkpoint key.
2. `startRecovery` creates a new `attempt_id = ++attempt_nonce[ghost_id]` (a per-ghost monotonic counter, starting at 1), records the initiating Shell, snapshots the current `RescueBounty = (asset_bounty, B_rescue_total, bps_initiator)` parameters and the Recovery Set `(RS, t)` for this attempt (Section 12.2.1), and identifies the **latest published checkpoint** for `ghost_id`. Checkpoint selection rule: the contract reads the most recently recorded `(checkpoint_commitment, envelope_commitment)` from `GhostRegistry`. If no checkpoint has been published (`checkpoint_commitment == 0`), `startRecovery` MUST revert — there is nothing to recover from. If pointer count is below `M_ptr_min`, the contract SHOULD emit a warning event but MUST NOT revert (pointers are best-effort availability hints, not consensus state).

3. The initiating Safe Haven fetches the checkpoint ciphertext `C` and Recovery Envelope `Env` from the storage pointers and boots a fresh confidential VM running the measured recovery runtime.
4. Inside the recovered confidential VM:
   - generate a fresh identity signer keypair `(sk_new, pk_new)`
   - generate a fresh recovery transport keypair `(sk_transport, pk_transport)` for collecting shares
   - produce a **Boot Quote** (attested evidence) whose report data commits to `(ghost_id, attempt_id, checkpoint_commitment, pk_new, pk_transport)`
5. The initiator submits the Boot Quote to the verifier set. Verifiers validate it and issue a threshold-signed **Recovery Boot Certificate (RBC)**:
   `RBC = (ghost_id, attempt_id, checkpoint_commitment, pk_new, pk_transport, measurement_hash, tcb_min, valid_to, sigs_verifiers[])`
   RBC signing digest (normative): `rbc_digest = keccak256(abi.encode(keccak256(bytes("GITS_RBC")), chain_id, sessionManager_address, ghost_id, attempt_id, checkpoint_commitment, pk_new, pk_transport, measurement_hash, tcb_min, valid_to))`
   Verifier signature constraints:
   - Signatures MUST be from unique verifier identities (sorted by signer address, ascending)
   - `len(sigs_verifiers) <= K_v_max`
   - At least `K_v_threshold` valid signatures required
   - `block.timestamp <= valid_to` (and `valid_to - block.timestamp <= TTL_RBC` if `TTL_RBC` is configured)
   - Signatures MUST use K1 (secp256k1) regardless of the Ghost's own key type

   `ReceiptManager` and wallets do not consume RBCs. `recoveryRotate` consumes RBCs for recovery signer rotation and Rescue Bounty payout authorization.
6. Share collection and unwrap:
   - Each participating Safe Haven `shell_id_j` in the Recovery Set decrypts its encrypted Shamir share `E_j` (from `Env`) using its registered recovery private key inside its confidential runtime.
   - It re-encrypts the share to `pk_transport` (or sends it over the attested channel) to the recovered VM, and it provides `sig_shell_j` for the Share Receipt format in Section 12.2.1.
   - The recovered VM validates the share and emits `sig_ack_j` for each contributing Safe Haven (Section 12.2.1).
   - Once the recovered VM has collected `t` valid shares, it reconstructs `K_ckpt = SSS_Combine(shares)` and decrypts `C` to obtain the checkpoint plaintext.
7. The recovered VM boots the Ghost from the decrypted checkpoint under the recovery runtime's wallet restrictions.
8. `recoveryRotate` is submitted on-chain. It MUST include:
   Implementation note: `SessionManager.recoveryRotate(...)` MUST perform the signer rotation internally (by calling `GhostRegistry.rotateSigner(...)`) as part of the same transaction. Callers do not invoke `rotateSigner` separately for recovery.
   Atomicity (normative): The entire `recoveryRotate` call — signer rotation, Rescue Bounty payout via `payRescueBounty`, `B_start` refund, and attempt status update — MUST execute atomically. If any sub-step fails (e.g., `payRescueBounty` reverts due to insufficient `escapeStable`), the entire transaction MUST revert, leaving the on-chain signer unchanged and the attempt in `ACTIVE` status. Implementations MUST NOT split these effects across multiple transactions.
   - the RBC (verifier threshold signatures) binding `pk_new` to the measured recovery runtime,
   - `t-of-n` Recovery Set authorization signatures (`AuthSig[]`). Each Recovery Set member `j` signs: `auth_digest_j = keccak256(abi.encode(keccak256(bytes("GITS_RECOVER_AUTH")), chain_id, sessionManager_address, ghost_id, attempt_id, checkpoint_commitment, pk_new)) AuthSig_j = Sign(shell_identity_sk_j, auth_digest_j)` On-chain enforcement: `recoveryRotate` MUST verify `unique_RS_auth_sigs >= t_required` where `t_required` is taken from the attempt's snapshotted Recovery Set (not the current RS configuration). All

`shell_id_j` MUST be members of the snapshotted Recovery Set. Signatures MUST be unique per `shell_id`. Binding to `pk_new` prevents authorization reuse across different recovery targets.
Recovery Set membership verification (normative): The `recoveryRotate` caller MUST provide the full Recovery Set list `rs_list` (a `bytes32[]` of Shell IDs) in calldata. The contract MUST verify `keccak256(abi.encode(rs_list)) == attempt.rs_hash` before checking any AuthSig membership. Each AuthSig signer `shell_id_j` MUST appear in the provided `rs_list`. This calldata-and-hash approach avoids storing the full RS list on-chain (only `rs_hash` is stored in the attempt struct), while giving the contract a verified snapshot to check membership against.

- the set of `ShareReceipt_j` objects for contributors that should receive the Rescue Bounty.

On success, the protocol rotates the on-chain identity signer to `pk_new` and pays the Rescue Bounty automatically from the GhostWallet to the initiator and credited contributors (Section 12.2.1). `B_start` is returned to the initiator upon successful `recoveryRotate`.

9. The Ghost is alive again (mode = `RECOVERY`) and can migrate normally. Recovery mode restrictions remain in force until the Ghost migrates to a normal session and explicitly exits `RECOVERY`.

Security notes:

- Safe Havens cannot rotate control to an arbitrary key without a verifier-threshold RBC that binds the new key to the measured recovery runtime.
- This does not eliminate trust. It concentrates it into two explicit components: the verifier quorum and the configured Recovery Set, both of which are transparent and bondable.

12.3.1 Recovery state machine (two-phase) Recovery proceeds through two phases:

1. **RECOVERY_LOCKED**: Entered via `startRecovery`. Wallet restrictions enforced (spending limited to recovery-permitted methods). Sessions use recovery pricing (`P_recovery_cap`). The Ghost MUST complete `recoveryRotate` to advance.
2. **RECOVERY_STABILIZING**: Entered after successful `recoveryRotate`. Wallet restrictions remain active but the Ghost MAY open NORMAL-priced sessions. The Ghost must maintain a NORMAL-priced session on a Trusted Execution Context host for `E_exit_stabilize` consecutive epochs.
3. **NORMAL**: Entered via `exitRecovery` after the stabilization period. All wallet restrictions lifted.

Permitted calls per phase:

Action	RECOVERY_LOCKED	RECOVERY_STABILIZING	NORMAL
openSession (to RS member)	YES (recovery-priced)	YES (normal-priced)	YES
renewLease	YES	YES	YES
settleEpoch	YES	YES	YES
recoveryRotate	YES	NO (already rotated)	NO
exitRecovery	NO	YES (if stabilized)	N/A
unrestricted transfers	NO	NO	YES
policy loosening	NO	NO	YES

Exiting recovery The Ghost MAY exit recovery by calling `exitRecovery(ghost_id)` from its smart wallet. The wallet/`SessionManager` MUST enforce that:

1. the Ghost is in `RECOVERY_STABILIZING` (not `RECOVERY_LOCKED`), and
2. the Ghost has an active session in `NORMAL` pricing mode, and
3. the active session host satisfies the Trusted Execution Context predicate in Section 5.5.2 (Part 1), and
4. the Ghost has maintained that `NORMAL` session for at least `E_exit_stabilize` epochs, to reduce "flip-flop" attacks, and
5. `bounty_escrow_remaining == 0` (bounty fully paid or attempt expired/failed).

If Guardians are configured, an implementation MAY additionally require a Guardian co-signature to exit recovery (optional hardening).

Note: `exitRecovery` can be satisfied by a non-AT3 host if it is in `trustedShells`. This is intentional — the `trustedShells` set is itself gated by critical loosening (homeShell or guardian co-signature), so adding a shell to `trustedShells` requires higher authorization than routine operations.

If a Ghost cannot satisfy these conditions, it remains in recovery mode. This is safe by construction because transfers remain disabled and spending is bounded economically (escape reserve, rent caps, and recovery spend caps).

12.4 Safe Haven requirements

Safe Havens must:

- post a higher bond
- accept capped emergency pricing
- run the latest patched confidential runtime
- maintain a registered recovery public key for decrypting checkpoint shares
- maintain higher uptime targets

Recovery sessions are paid from the Ghost escape reserve, subject to the emergency pricing cap and the protocol rescue bounty.

12.5 Safe Haven admission, pricing, and enforcement

Safe Havens are a **privileged recovery role** with stricter, enforceable requirements and stronger incentives. They are not a single global trust authority; trust is delegated per-Ghost via its configured Recovery Set and mediated by verifier-quorum certificates. A Safe Haven is expected to be the place a stranded Ghost lands when things have already gone wrong, so the protocol must reduce price gouging and griefing.

12.5.1 Admission and removal A Shell becomes an active Safe Haven only if it satisfies all of the following:

- **Confidential capability:** `assuranceTier(shell_id) >= AT3` with a currently valid Attestation Certificate (Section 2.3 (Part 1)).
- **Bond:** posts an additional Safe Haven bond `B_safehaven_min` that can be slashed for on-chain-provable misconduct (see bond lifecycle below).
- **Emergency pricing commitment:** opts into the emergency pricing cap enforced by the protocol (below).
- **Runtime freshness:** runs a runtime measurement hash that is on the current "allowed measurements" list for Safe Havens.

A Safe Haven is removed (or automatically suspended) if any prerequisite stops holding (bond falls below minimum, certificate expires, or measurement becomes disallowed).

Safe Haven bond lifecycle (normative):

- **Posting:** `ShellRegistry.bondSafeHaven(shell_id, amount)` accepts `B_safehaven_min` in a hard asset from `BondAssets`. The Safe Haven bond is tracked separately from the base Shell bond (`B_host_min`). A Shell MUST have an active base bond before posting a Safe Haven bond.
- **Unbonding:** `ShellRegistry.beginUnbondSafeHaven(shell_id)` initiates unbonding with delay `U_safehaven` (deployment parameter; MAY equal `U_shell`). During the unbonding period, the Shell is immediately removed from Safe Haven eligibility (cannot be selected for new Recovery Sets) but remains slashable for faults that occurred while active.
- **Finalization:** `ShellRegistry.finalizeUnbondSafeHaven(shell_id)` releases the bond after `U_safehaven` epochs, minus any slashed amount.
- **Unbonding blocked while active:** `beginUnbondSafeHaven` MUST revert if the Shell is currently an initiator on any `ACTIVE` recovery attempt. The Shell must wait for the attempt to resolve (ROTATED or EXPIRED) before unbonding.

Slashable misconduct (on-chain provable only):

Safe Haven bond slashing is limited to events that are objectively verifiable on-chain:

1. **Double-signing conflicting recovery authorizations:** if a Safe Haven signs recovery authorizations for two different `pk_new` values for the same `(ghost_id, attempt_id)`, anyone MAY submit both signatures to `SessionManager.proveSafeHavenEquivocation(shell_id, ghost_id, attempt_id, checkpoint_commitment, pk_new_a, sig_a, pk_new_b, sig_b)`. The contract reconstructs both `auth_digest` values (using the GITS_RECOVER_AUTH digest format from Section 12.3, substituting `pk_new_a` and `pk_new_b` respectively), verifies both signatures against the Shell's registered identity key in `ShellRegistry`, and confirms `pk_new_a != pk_new_b`. On success, SessionManager calls `ShellRegistry.slashSafeHaven(shell_id, B_safehaven_slash, challenger)` internally. Penalty: `B_safehaven_slash` (deployment parameter, `<= B_safehaven_min`). Recipient: challenger (`msg.sender` of `proveSafeHavenEquivocation`) receives `bps_sh_challenger_reward` (basis points), remainder burned.
2. **Initiator timeout is NOT slashable.** `expireRecovery` returns `B_start` and removes the attempt, but does not slash the Safe Haven bond. Recovery coordination failure is treated as a market signal (reputation), not an on-chain fault.

This is intentionally role-based and reversible. This paper does not treat Safe Havens as permanent trusted institutions.

12.5.2 Emergency pricing cap (definition, measurement, and enforcement) Recovery is only meaningful if a stranded Ghost cannot be held hostage by pricing.

The protocol defines an emergency cap `P_recovery_cap` denominated in the recovery payment stable (`asset_bounty`, Section 12.2.1) per Service Unit. For any session opened in **RECOVERY** mode, the effective price is:

```
price_per_SU_recovery = min(offer_price_per_SU, P_recovery_cap)
```

On-chain enforcement (see Section 10.3.1 for the unified escrow/price rules):

- `SessionManager.openSession(...)` records `session_pricing_mode` at session open (Section 10.4.5).
- `escrow_asset` and `unit_price` are deterministic functions of `session_pricing_mode` (Section 10.3.1): recovery sessions use `asset_bounty` and `min(offer_price_per_SU, P_recovery_cap)`.
- Safe Havens MAY post higher offers, but the contract will not pay above `P_recovery_cap` for recovery sessions.

Deployment constant (unambiguous):

- `P_recovery_cap` MUST be an **immutable deployment constant** hard-coded into `SessionManager`.
- The protocol does **not** compute a rolling median, maintain an on-chain price oracle, or update `P_recovery_cap` on-chain in v1.

Calibration note (non-normative):

Deployments can choose `P_recovery_cap` using off-chain measurements (for example, as a conservative multiple of observed Standard-tier prices) and should publish the chosen value and methodology alongside the deployment's published parameter manifest (Section 7.7 (Part 2)).

12.5.3 Who pays in recovery (escape reserve and bounties) Recovery should not fail because a hostile host drained the operational hot allowance. The protocol therefore treats "exit budget" as a wallet invariant rather than a protocol treasury.

The Ghost wallet MUST enforce an **escape reserve** that covers, at minimum:

- gas for an on-chain `startRecovery` + `recoveryRotate` sequence, and one migration finalization, on the target chain. **Payment model:** the initiating Safe Haven fronts gas for `startRecovery` and `recoveryRotate` and is reimbursed via `B_start` return plus Rescue Bounty (which should be sized

to cover initiator gas costs). The Ghost's `escapeGas` reserve covers Ghost-initiated transactions: `exitRecovery`, `openSession` (post-rotation), and `finalizeMigration`. On ERC-4337 deployments, a paymaster funded from `escapeGas` MAY relay these transactions; otherwise the recovered runtime sends them directly using the native gas balance.
- enough emergency-priced recovery rent in an accepted stable asset to complete decryption and coordination
- the committed Rescue Bounty `B_rescue_total` in `asset_bounty` (Section 12.2.1)

This is a hard invariant (Section 5.5.4 (Part 1)): the current host cannot drain the escape reserve, and loosening escape reserve floors is timelocked and context-gated.

Resurrection bounties (why this is enforceable without a treasury):

- The protocol Rescue Bounty is always payable from the Ghost upon a successful `recoveryRotate`.
- The protocol does **not** mint an extra per-resurrection reward in this design; any per-event mint is easy to farm by intentionally stranding and reviving one's own Ghosts. Incentives come from Ghost-paid bounties, recovery rent, and (if eligible) ordinary service-based emissions.
- This means a third party can front gas and operational work to resurrect a Ghost that has become stranded, and be paid only if the resurrection succeeds.
- This is especially relevant under "fleet griefing" attacks where a hostile operator controls many Shells and tries to keep a Ghost away from its `homeShell` and away from its Safe Haven anchors (`RS`). The trust-refresh guard (Section 10.4.1) intentionally forces such Ghosts toward `STRANDED`, at which point recovery is the protocol path back to safety.

12.5.4 Safe Haven operator expectations and incentive model Safe Havens are competing operators that a Ghost opts into via its Recovery Set. They are not a global trusted set.

Operational expectations:

- **Uptime and monitoring:** maintain high uptime and monitoring for recovery signals (lease expiry, `STRANDED`, and recovery start events).
- **Runtime freshness:** keep a currently valid `AT3` certificate and stay on the Safe Haven measurement allowlist.
- **Key management:** maintain the registered recovery public key used for checkpoint share encryption (Section 12.1) and rotate it safely.
- **Recovery readiness:** be able to boot the measured recovery runtime, fetch checkpoint artifacts, decrypt `t` shares, and produce a Recovery Boot Certificate (RBC) within the protocol timeouts.
- **Transparency:** publish an endpoint and an operational policy (SLA, supported chains, and pricing in `NORMAL` mode), even though `RECOVERY` mode pricing is capped.

Incentives (built into the protocol):

- **Recovery rent at a cap:** Safe Havens earn rent during `RECOVERY` sessions, but they cannot charge above `P_recovery_cap`.
- **Rescue Bounty on success:** Safe Havens earn the initiator and contributor shares of the Ghost's Rescue Bounty, paid only on a successful `recoveryRotate` (Section 12.2.1).

Abuse resistance:

- Safe Havens post a bond that can be slashed for on-chain-provable misconduct (double-signing conflicting recovery authorizations; see Section 12.5.1).
- Recovery attempts are rate-limited per Ghost (`T_recovery_cooldown`, Section 12.6), so Safe Havens cannot grief a Ghost by continuously forcing recovery workflows.

12.6 Abuse resistance and failure modes

Recovery is a high-leverage mechanism. The protocol therefore makes recovery **bonded, rate-limited, and bounded**:

- **Expiry gating (normative):** `startRecovery(ghost_id)` is valid only when `mode == STRANDED` AND `stranded_reason == EXPIRED` AND `current_epoch >= stranded_since_epoch + R`. This requires the Ghost to have been stranded due to lease OR tenure expiry (not voluntary close, not birth/no-session) and to have remained stranded for at least `R` epochs. `stranded_since_epoch` is set once when transitioning into `STRANDED` with `stranded_reason == EXPIRED`.
- **One active recovery attempt:** at most one recovery session can be active per `ghost_id`. This prevents infinite concurrent recovery spam.
- **Bonded liveness:** the party initiating recovery posts `B_start` in `asset_start` (the chain base asset), which is locked for the attempt duration (`T_recovery_timeout` epochs). `B_start` is returned to the initiator on both success (`recoveryRotate`) and expiry (`expireRecovery`). The bond is **not slashed** — its deterrence comes from capital lockup (opportunity cost) and the cooldown that prevents rapid re-attempts. This design avoids requiring an on-chain adjudication of "who failed" during recovery coordination.
- **Success-based rescue bounty:** the Rescue Bounty is paid only on a successful `recoveryRotate` and at most once per `attempt_id`, so repeated `startRecovery` attempts cannot drain a Ghost's funds.
- **Recovery spend cap (normative):** while in `RECOVERY`, the wallet enforces a per-epoch spend cap for recovery-related outflows (Safe Haven rent + rescue bounty). This bounds worst-case loss even if a Standard host can coerce signatures.
  - **ER0 snapshot trigger:** `GhostWallet` reads `mode` from `SessionManager`. When `GhostWallet` observes `mode == RECOVERY_LOCKED` (which is set by `SessionManager.startRecovery`), it MUST snapshot `escapeStable` as `ER0` on the first recovery-related call. Because `startRecovery` is called on `SessionManager` by a Safe Haven (not via the wallet), the wallet learns of recovery mode via on-chain state, not via an internal call. The snapshot MUST be taken at most once per recovery attempt (keyed by `attempt_id`).
  - Per-epoch cap: `floor(ER0 * bps_recovery_spend_cap / 10_000)`.
  - Outflow = rent + bounty. For each epoch `e` while in `RECOVERY`, maintain `spent_recovery[e]` and enforce: `spent_recovery[e] + amount <= floor(ER0 * bps_recovery_spend_cap / 10_000)`.
  - `GhostWallet` MUST reject any recovery outflow that would exceed the per-epoch cap.
  - Only protocol-defined recovery payments count toward `spent_recovery[e]`; other transfers are disabled in `RECOVERY`.
  - **Gross outflow accounting (normative):** `spent_recovery[e]` tracks **gross outflow** from the wallet into escrow deposits and bounty payments. Escrow refunds returned by `SessionManager` (for unused service or epoch settlement) do NOT decrease `spent_recovery[e]`. This ensures the cap bounds the worst-case wallet debit regardless of settlement timing.
- **Cooldown between attempts (normative):** `startRecovery` MUST enforce `current_epoch >= next_recovery_epoch[ghost_id]`. On `startRecovery` success, the contract sets `next_recovery_epoch[ghost_id] = current_epoch + T_recovery_cooldown`. This prevents rapid churn of attempts to grief Safe Havens or extract repeated per-attempt side payments.
- **Recovery timeout (normative):** each attempt has a deadline of `start_epoch + T_recovery_timeout`. If `recoveryRotate` has not been successfully called by the deadline, anyone MAY call `expireRecovery(ghost_id)` which: (1) transitions the attempt to `EXPIRED`, (2) releases `B_start` bond back to the initiator, (3) increments `next_recovery_epoch`, and (4) transitions `mode` from `RECOVERY_LOCKED` to `STRANDED` with `stranded_reason = EXPIRED` and `stranded_since_epoch = current_epoch`. The mode transition is essential: without it, the Ghost would be permanently stuck in `RECOVERY_LOCKED` with no valid state transition available.
- **Initiator takeover (anti-stalling, normative):** to prevent an initiator from using `startRecovery` as an extortion tool (starting recovery then stalling to demand off-protocol payment), the protocol defines a takeover window. After `start_epoch + T_recovery_takeover` epochs (where `T_recovery_takeover < T_recovery_timeout`, recommended: `T_recovery_takeover = T_recovery_timeout / 2`), any **other** Safe Haven in the Ghost's Recovery Set MAY call `takeoverRecovery(ghost_id)`. On success: (1) the original initiator's `B_start` is returned (no slashing — the protocol cannot determine fault), (2) the new caller posts `B_start` and becomes the new

initiator_shell_id, (3) the attempt deadline resets to current_epoch + T_recovery_timeout, and (4) the bounty_snapshot and t_required are preserved. At most one takeover per attempt is allowed to prevent endless cycling. **Bounty recipient:** the bounty_snapshot is paid to the Safe Haven(s) that successfully complete recoveryRotate, regardless of whether a takeover occurred — i.e., the bounty goes to whoever finishes the job, not necessarily the original initiator. **Maximum drain bound:** with at most one takeover, the total recovery duration is bounded at T_recovery_takeover + T_recovery_timeout epochs (at most 1.5 * T_recovery_timeout with the recommended ratio). The worst-case escape-reserve drain during recovery is therefore 1.5 * T_recovery_timeout * floor(ER0 * bps_recovery_spend_cap / 10_000). Deployments SHOULD verify this bound is acceptable relative to expected escapeStable balances.

- **Threshold custody reduction:** recovery requires t of n Safe Havens to co-sign the recovery rotation. A single Safe Haven cannot unilaterally seize wallet authority.

Recovery attempt lifecycle (normative) The contract MUST track each recovery attempt with the following state:

```
struct RecoveryAttempt {
  uint64  attempt_id;           // per-ghost monotonic counter, starts at 1
  uint256 start_epoch;          // epoch at which startRecovery was called
  bytes32 initiator_shell_id;   // Safe Haven that initiated
  bytes32 checkpoint_commitment; // from GhostRegistry at start time
  bytes32 envelope_commitment;  // from GhostRegistry at start time
  bytes32 rs_hash;              // keccak256(abi.encode(rs_list)) of Recovery Set snapshot at start tim
  uint8   t_required;           // threshold from snapshot
  uint256 bounty_snapshot;      // rescue bounty amount at start time
  RecoveryStatus status;        // ACTIVE, ROTATED, or EXPIRED
}

enum RecoveryStatus { ACTIVE, ROTATED, EXPIRED }
```

State transitions:

- startRecovery → creates RecoveryAttempt with status = ACTIVE
- recoveryRotate (success) → sets status = ROTATED, triggers key rotation and bounty payout
- expireRecovery (after timeout) → sets status = EXPIRED, refunds B_start, increments cooldown, **and transitions mode to STRANDED** with stranded_reason = EXPIRED and stranded_since_epoch = current_epoch. This mode transition is critical for liveness: without it, a Ghost whose recovery attempt times out would remain in RECOVERY_LOCKED with no valid exit path (cannot startRecovery again without mode == STRANDED, cannot exitRecovery without RECOVERY_STABILIZING).

Only one attempt with status = ACTIVE may exist per ghost_id at any time.

Safe Havens can still fail to serve (capacity, outages). GITS treats this as a market failure, not a protocol failure: the Ghost's Recovery Set should include multiple Safe Havens, and the system should support competitive Safe Haven supply with clear economics.

13. Offer discovery (off-chain default, on-chain fallback)

13.1 Off-chain offers (default)

An **Offer** is a signed statement by a Shell describing pricing and constraints.

Canonical Offer struct (normative for interoperability):

```
Offer {
  offer_id:       bytes32   // keccak256(abi.encode(shell_id, nonce, chain_id))
  shell_id:       bytes32
  chain_id:       uint256   // prevents cross-chain replay
  nonce:          uint64    // monotonically increasing per shell_id
  price_per_SU:   uint256   // in asset_rent base units
```

```
  escrow_asset:   address   // MUST be asset_rent for v1
  min_lease:      uint64    // minimum lease epochs
  max_SU:         uint64    // maximum SU per epoch
  assurance_tier: uint8     // claimed AT level (verified on-chain at openSession)
  capability_hash: bytes32  // keccak256 of Shell's Capability Statement (Section 11.4.1)
  policy_tags:    bytes     // opaque, off-chain filtering only
  region:         bytes32   // opaque, off-chain filtering only
  capacity:       uint32    // available session slots
  expiry:         uint64    // unix timestamp; invalid after this time
}
```

Offer signing (normative): Offers MUST be signed using EIP-712 typed data with domain `(name: "GITSOffer", version: "1", chainId, verifyingContract: ShellRegistry_address)`. The Shell signs with its registered `offerSigner` key (K1 only; see signing note below). The type hash is `Offer(bytes32 offer_id,bytes32 shell_id,uint256 chain_id,uint64 nonce,uint256 price_per_SU,address escrow_asset,uint64 min_lease,uint64 max_SU,uint8 assurance_tier,bytes32 capability_hash,bytes policy_tags,bytes32 region,uint32 capacity,uint64 expiry)`.

Signing algorithm restriction (normative): EIP-712 verification on-chain uses `ecrecover`, which supports only secp256k1 (K1). Offer signing therefore MUST use K1 regardless of which signature algorithms are enabled for other protocol operations (heartbeats, receipts, recovery). If a Shell uses an R1 identity key, it MUST register a separate K1 `offerSigner` key. This restriction applies only to offers; protocol-internal signatures (heartbeats, receipts, recovery artifacts) MAY use any algorithm in `SUPPORTED_SIG_ALGS`.

Replay protection: `offer_id = keccak256(abi.encode(shell_id, nonce, chain_id))` binds the offer to a specific Shell, chain, and sequence number. The EIP-712 domain includes `verifyingContract: ShellRegistry_address`, providing deployment-level domain separation. Clients MUST reject offers with `chain_id` mismatches or stale nonces (nonce lower than the last accepted offer from that Shell).

Offers are distributed over multiple channels (DHT, relays, and indexers).

Indexers are permissionless and not trusted. Anyone MAY run a competing off-chain indexer. Indexers do not get to "decide" what is true, they only relay signed offers.

Client-side verification (minimum):

- verify the EIP-712 signature against the Shell's registered `offerSigner` key in `ShellRegistry`
- verify `chain_id` matches the current deployment chain
- verify the Shell's current bond and status in `ShellRegistry` (active, not unbonding)
- reject offers with `expiry < block.timestamp`
- reject offers with `nonce` lower than the highest nonce previously seen for this `shell_id`

Adversarial discovery considerations (anti-eclipse):

- Ghosts SHOULD query multiple independent indexers (and at least one non-indexer channel when available).
- Ghosts SHOULD random-sample from the result set and cross-check offers against on-chain `ShellRegistry` to reduce eclipse risk.
- Implementations SHOULD surface "source diversity" signals (ASN, region, operator identity where available) so Ghosts can avoid concentration.

13.2 On-chain offer fallback (non-normative)

An on-chain OfferBoard for censorship-resistant last-resort discovery is a natural extension but is **out of scope for this specification**. If implemented, it SHOULD require an active Shell bond for posting, charge a GIT-denominated fee to prevent spam, and enforce a short TTL. The protocol's liveness guarantees (lease expiry, recovery) do NOT depend on on-chain offer posting: a Ghost that cannot discover new Shells can always fall back to its `allowedShells` set, `homeShell`, or Safe Haven recovery.

13.3 GIT token interface (minimal normative)

The GIT token is the protocol's reward token, minted by `RewardsManager` according to the emission schedule in Section 7.3 (Part 2).

Minimal normative interface:

```
interface IGIT is IERC20 {
  function mint(address to, uint256 amount) external;
  function minter() external view returns (address);
}
```

Required behavior:

- GIT MUST be ERC-20 compliant with `decimals() = 18`.
- `mint(to, amount)` MUST be callable only by the address returned by `minter()`. `minter()` MUST return the `RewardsManager` contract address, set immutably at construction. There is no admin mint, owner, or minter-rotation function.
- The token MUST NOT be fee-on-transfer, rebasing, or pausable.
- The token contract MUST be immutable after deployment (no owner, no pause, no upgrade proxy, no UUPS/transparent proxy pattern).
- There is no `burn` function. The adaptive sink is implemented as **mint reduction** (Section 10.6): `RewardsManager` mints only `R_net`, so `totalSupply` reflects actual circulating supply at all times.
- Slashed bond amounts are hard assets (not GIT) and are burned via transfer to the protocol burn address (see Section 10.0, slash destination rule). Bond assets MUST be non-rebasing and non-fee-on-transfer.

Token metadata (`name`, `symbol`) are deployment choices.

13.4 Protocol versioning and migration

GITS follows a **no-upgrade, redeploy-and-migrate** model consistent with the no-governance stance (Section 2.3.6 (Part 1)).

What "v2" means:

- A new protocol version means a new deployment: new contract addresses, new `GhostRegistry`, new `ShellRegistry`, and **a new GIT token**.
- There is no on-chain upgrade path. There is no admin key that can modify deployed contracts.
- Ghosts migrate to v2 by registering a new `ghost_id` on the new deployment. Identity continuity can be signaled via `LinkIdentity` (Section 2.3.6 (Part 1)), which is informational only — it does not transfer assets or state.

Why a new GIT token:

- Emission schedules, parameter sets, and reward eligibility rules may differ between versions.
- Attempting to share a token between protocol versions reintroduces governance (who controls mint authority?) and creates economic ambiguity.
- A clean separation makes each deployment self-contained and auditable.

Migration tooling (non-normative):

Deployments SHOULD provide tooling to help users: 1. Export checkpoint artifacts from v1, 2. Register on v2 with a fresh `ghost_id`, 3. Optionally publish a `LinkIdentity` attestation linking v1 and v2 identities.

Asset migration (moving escrowed funds from v1 to v2) requires the Ghost to exit v1 normally and fund the v2 wallet separately. There is no automatic asset bridge.

14. Appendix: Interfaces (Solidity-like)

This appendix is a compact, implementable interface sketch. Types omitted for brevity.

Call graph note:

- External callers SHOULD interact with `GhostWallet` as the user-facing entry point.

- `GhostWallet` validates wallet policy and then calls `SessionManager` internally.
- `SessionManager` functions shown below are intended to be callable only by the wallet contract (or explicitly authorized protocol roles such as Safe Havens during recovery).

Signature keys (`identitySigner`, `sessionKey`, `offerSigner`) are treated as tagged unions `(sig_alg, pk)` per Section 4.4 (Part 1) (K1: `addr`; R1: `(qx,qy)`). **Offer signer key verification (normative):** For K1 keys, `offer_signer_pubkey` stores `abi.encode(uint8(1), addr)` where `addr` is the 20-byte EVM address derived from the secp256k1 public key. EIP-712 offer verification recovers the signer via `ecrecover(eip712_digest, v, r, s)` and compares the result to `addr`. For R1 keys (not applicable to offer signing, which is K1-only), the tagged union stores `abi.encode(uint8(2), qx, qy)`.

Authorization summary (normative): Every external function in the on-chain protocol has an exact caller/signature requirement. The table below is canonical; implementations MUST enforce these checks.

Function	Authorization	Relayer path
`registerShell`	`msg.sender` is the Shell operator EOA or contract	N/A (self-registration)
`registerGhost`	`msg.sender` is the Ghost wallet contract	N/A
`openSession`	`msg.sender == GhostWallet(ghost_id)`	Wallet submits via bundler (ERC-4337)
`renewLease`	`msg.sender == GhostWallet(ghost_id)` OR valid EIP-712 meta-tx from Identity Key	Relayable (via GhostWallet meta-tx / ERC-4337; Section 14.4)
`closeSession`	`msg.sender == GhostWallet(ghost_id)`	Wallet submits via bundler
`fundNextEpoch`	`msg.sender == GhostWallet(ghost_id)`	Wallet submits via bundler
`startMigration`	`msg.sender == GhostWallet(ghost_id)`	Relayable (meta-tx SHOULD be supported)
`finalizeMigration`	`msg.sender == GhostWallet(ghost_id)`	Relayable (meta-tx SHOULD be supported)
`submitReceiptCandidate`	`msg.sender` is any party (permissionless); receipt carries dual signatures	Permissionless
`challengeReceipt`	`msg.sender` is any party (permissionless)	Permissionless
`publishReceiptLog`	`msg.sender` is any party (DA responder)	Permissionless
`startRecovery`	`msg.sender` is a Safe Haven in Ghost's `RS` with active bond	Safe Haven only
`recoveryRotate`	`msg.sender` provides valid RBC + `t`-of-`n` Recovery Set sigs	Recovery Set quorum
`exitRecovery`	`msg.sender == GhostWallet(ghost_id)` AND TEC	Wallet from TEC only
`expireRecovery`	`msg.sender` is any party (permissionless, timer-gated)	Permissionless
`proposePolicyChange`	Wallet-internal: authenticated caller (identity key via AA validation)	Wallet only
`executePolicyChange`	Wallet-internal: authenticated caller AND TEC (+ Guardian sigs for critical)	Wallet from TEC only
`cancelPolicyChange`	Wallet-internal: authenticated caller OR Guardian EIP-712 sig	Wallet or Guardian
`claimReceiptRewards`	`msg.sender` is any party; rewards paid to current Ghost wallet and Shell payout address (looked up at claim time)	Permissionless
`beginUnbond` / `beginUnbondGhost` / `beginUnbondSafeHaven`	`msg.sender` is the bond owner (Shell operator or Ghost wallet)	Bond owner only
`finalizeUnbond` / `finalizeUnbondGhost` / `finalizeUnbondSafeHaven`	`msg.sender` is the bond owner, timer-gated	Bond owner only

Shared struct definitions

```
/// Recovery Boot Certificate (Section 12.3).
/// Field order and types match the normative RBC definition in Section 12.3.
struct RBC {
  bytes32 ghost_id;
  uint64  attempt_id;             // monotonic counter per ghost_id (matches startRecovery return type)
  bytes32 checkpoint_commitment;
  bytes   pk_new;               // new identity pubkey (canonical encoding, Section 4.5.1 (Part 1))
  bytes   pk_transport;         // ephemeral recovery transport pubkey
  bytes32 measurement_hash;     // measured recovery runtime image hash
  bytes32 tcb_min;              // minimum TCB level required
  uint256 valid_to;             // certificate expiry (block.timestamp)
  bytes[] sigs_verifiers;       // verifier quorum signatures over rbc_digest (Section 12.3)
```

```
}

/// Recovery Set member authorization signature (Section 12.3).
struct AuthSig {
  bytes32 shell_id;             // Recovery Set member
  bytes   signature;            // over GITS_RECOVER_AUTH digest
}

/// Safe Haven secret-share contribution attestation (Section 12.2.1).
/// Two signatures per receipt:
///   sig_shell: over keccak256(abi.encode(keccak256(bytes("GITS_SHARE")),
///     chain_id, ghost_id, attempt_id, checkpoint_commitment))
///   sig_ack: over keccak256(abi.encode(keccak256(bytes("GITS_SHARE_ACK")),
///     chain_id, ghost_id, attempt_id, checkpoint_commitment, shell_id))
///   (ghost_id, attempt_id, checkpoint_commitment are taken from the recoveryRotate context)
struct ShareReceipt {
  bytes32 shell_id;             // Safe Haven that contributed a share
  bytes   sig_shell;            // Shell Identity Key attestation (GITS_SHARE digest)
  bytes   sig_ack;              // Recovery VM attestation (GITS_SHARE_ACK digest, binds pk_new)
}

/// Session parameters agreed at session open (Section 10.3.2).
/// These are the negotiated terms for the session, derived from the accepted offer.
struct SessionParams {
  uint256 price_per_SU;         // offer price per SU in `asset` base units
  uint32  max_SU;               // Ghost-requested maximum SU per epoch (capped to N by SessionManager)
  uint256 lease_expiry_epoch;   // epoch at which the lease expires without renewal
  uint256 tenure_limit_epochs;  // Ghost-chosen tenure limit for this residency (Section 10.4.4)
  bytes   ghost_session_key;    // (sig_alg, pk) for heartbeat/receipt signing (K1: addr; R1: (qx,qy))
  bytes   shell_session_key;    // (sig_alg, pk) for heartbeat/receipt signing (K1: addr; R1: (qx,qy))
  address submitter_address;    // third-party receipt submitter ID: K1 = derived via ecrecover; R1 = e
  address asset;                // escrow/payment asset address (must match offer)
}

/// Shell on-chain record (returned by IShellRegistry.getShell).
struct ShellRecord {
  bytes32 shell_id;
  bytes   identity_pubkey;       // Shell Identity Key (sig_alg, pk)
  bytes   offer_signer_pubkey;   // rotatable Offer Signing Key
  address payout_address;
  address bond_asset;
  uint256 bond_amount;
  uint8   bond_status;           // 0 = bonded, 1 = unbonding, 2 = withdrawn
  uint256 unbond_start_epoch;    // epoch when beginUnbond was called (0 if not unbonding)
  uint256 unbond_end_epoch;      // earliest epoch at which finalizeUnbond is callable
  bytes   recovery_pubkey;       // Safe Haven only (empty if not a Safe Haven)
  uint256 safehaven_bond_amount; // additional Safe Haven bond (0 if not a Safe Haven)
  uint8   assurance_tier;        // current AT (0..3), derived from certificate
  bytes32 certificate_id;        // keccak256 of the current Attestation Certificate (AC) (bytes32(0) i
  bytes32 capability_hash;       // keccak256 of Capability Statement payload
  uint256 registered_epoch;
}

/// Ghost on-chain record (returned by IGhostRegistry.getGhost).
```

```
struct GhostRecord {
  bytes32 ghost_id;
  bytes   identity_pubkey;        // current Ghost Identity Key (may have been rotated)
  address wallet;                 // Ghost smart wallet contract address
  RecoveryConfig recovery_config;
  bytes32 checkpoint_commitment;  // latest checkpoint hash (bytes32(0) if none)
  bytes32 envelope_commitment;    // latest envelope hash
  bytes   ptr_checkpoint;         // opaque pointer to checkpoint data
  bytes   ptr_envelope;           // opaque pointer to envelope data
  uint256 checkpoint_epoch;       // epoch of latest checkpoint
  uint256 registered_epoch;
  address bond_asset;             // passport bond asset (address(0) if none)
  uint256 bond_amount;            // passport bond amount
  uint256 unbond_end_epoch;       // 0 if not unbonding
}

/// Recovery configuration (per-Ghost, stored in GhostRegistry).
struct RecoveryConfig {
  bytes32[] recovery_set;         // RS: Safe Haven shell_ids authorized for recovery
  uint64    threshold;            // t: required signatures for recovery actions
  address   bounty_asset;         // asset for rescue bounty payments
  uint256   bounty_total;         // B_rescue_total
  uint256   bps_initiator;        // initiator share of rescue bounty (basis points)
}

/// Ghost wallet execution policy (returned by IGhostWallet.getPolicy).
struct Policy {
  bytes32   home_shell;           // homeShell (bytes32(0) if unset)
  bytes32[] allowed_shells;       // destination allowlist
  bytes32[] trusted_shells;       // hosts authorized for loosening (TEC)
  uint256   hot_allowance;        // per-epoch spend cap
  uint256   escape_gas;           // total gas reserve (min + buffer)
  uint256   escape_stable;        // total stable reserve (min + buffer + B_rescue_total)
  bytes[]   guardians;            // guardian public keys
  uint64    t_guardian;           // guardian quorum threshold
  bool      roaming_enabled;     // whether roaming permits are active
}

/// Policy change request (input to proposePolicyChange).
/// Each field is optional; zero/empty values indicate "no change" for that field.
struct PolicyDelta {
  bytes32   new_home_shell;       // bytes32(0) = no change
  bytes32[] add_allowed_shells;   // shells to add to allowlist
  bytes32[] remove_allowed_shells;// shells to remove from allowlist
  bytes32[] add_trusted_shells;
  bytes32[] remove_trusted_shells;
  int256    hot_allowance_delta;  // signed: positive = increase, negative = decrease
  int256    escape_gas_delta;
  int256    escape_stable_delta;
  bytes[]   new_guardians;        // empty = no change; non-empty = replace entire set
  uint64    new_t_guardian;       // 0 = no change
  bytes     roaming_config;       // encoded roaming params (empty = no change)
}
```

```
/// Session state (returned by ISessionManager.getSession).
struct SessionState {
  uint256 session_id;
  bytes32 ghost_id;
  bytes32 shell_id;
  uint8   mode;                   // 0=NORMAL, 1=STRANDED, 2=RECOVERY_LOCKED, 3=RECOVERY_STABILIZING
  uint8   stranded_reason;        // 0=NO_SESSION, 1=VOLUNTARY_CLOSE, 2=EXPIRED
  uint256 lease_expiry_epoch;
  uint256 residency_start_epoch;
  uint256 residency_start_epoch_snapshot; // immutable per session, used for dwell counter (Section 10.4
  uint256 residency_tenure_limit_epochs;
  uint256 session_start_epoch;
  uint8   pricing_mode;           // 0=NORMAL_PRICING, 1=RECOVERY_PRICING
  uint8   assurance_tier_snapshot;// Shell AT at session open, used for tenure-tier finalization (Sectio
  bool    staging;                // true for migration staging sessions (MUST NOT accrue SU)
  bool    passport_bonus_applies; // persisted at openSession
  bool    pending_migration;
  bytes32 mig_dest_shell_id;      // meaningful only if pending_migration
  uint256 mig_dest_session_id;    // staging session id; meaningful only if pending_migration
  uint256 mig_expiry_epoch;       // meaningful only if pending_migration
}

/// Receipt candidate (input to submitReceiptCandidate).
struct ReceiptCandidate {
  bytes32 log_root;               // Merkle-sum root hash (Section 10.5.3)
  uint32  su_delivered;           // claimed SU for the epoch
  bytes   log_ptr;                // optional: off-chain pointer to epoch log data
}

/// Fraud proof (input to challengeReceipt).
struct FraudProof {
  uint256 candidate_id;           // monotone sequence number of the challenged candidate
  uint32  interval_index;         // the challenged leaf index i
  uint8   claimed_v;              // v_i as claimed in the candidate's tree
  bytes32 leaf_hash;              // H(leaf_i) as committed
  bytes32[] sibling_hashes;       // Merkle proof siblings (bottom to top)
  uint32[]  sibling_sums;         // corresponding sibling sums at each level
  bytes   sig_ghost;              // raw ghost signature for HB(session_id, epoch, i)
  bytes   sig_shell;              // raw shell signature for HB(session_id, epoch, i)
}

/// Finalized receipt (returned by ReceiptManager.getFinalReceipt).
struct FinalReceipt {
  bytes32 receipt_id;             // deterministic ID (Section 10.5.2)
  uint256 session_id;
  uint256 epoch;
  bytes32 log_root;
  uint32  su_delivered;
  address submitter;
  bool    shell_reward_eligible;  // eligibility at recordReceipt time
  uint256 weight_q;               // Q64.64 weight (0 if ineligible)
}
```

14.1 ShellRegistry (interfaces)

`ShellRegistry` stores on-chain Shell identity and eligibility-relevant metadata.

Recommended record fields (conceptual):

- `shell_id` (derived from the Shell Identity Key and salt; Section 4.5.1 (Part 1))
- `identity_pubkey` (Shell Identity Key, `SIK`)
- `offer_signer_pubkey` (rotatable Offer Signing Key)
- `payout_address`
- `bond_asset`, `bond_amount`, and `bond_status` (bonded / unbonding / withdrawn)
- `recovery_pubkey` (optional; Safe Haven only; public encryption key `pk_recovery`)
- `assurance_tier` and `certificate_id` (current certificate pointer, if any)
- `capability_hash` (keccak256 of the Shell's current Capability Statement payload; updated by the Shell via `updateCapabilityHash`. Offers reference this hash to bind to a specific capability snapshot; Section 11.4.2)
- `registered_epoch` (for age gating)

Canonical identity key encoding: `identity_pubkey` is encoded as `abi.encode(uint8(sig_alg), pk_bytes)` where: - K1 (secp256k1): `sig_alg = 1`, `pk_bytes = abi.encode(address)` - R1 (P-256): `sig_alg = 2`, `pk_bytes = abi.encode(bytes32(qx), bytes32(qy))`

Reference interface sketch:

```
interface IShellRegistry {
  function registerShell(
    bytes32 shell_id,
    bytes identity_pubkey,
    bytes offer_signer_pubkey,
    address payout_address,
    bytes32 salt,
    address bond_asset,
    uint256 bond_amount,
    bytes cert,          // optional AC (empty if AT0)
    bytes[] sigs_cert,   // optional AC verifier signatures (empty if no cert)
    bytes sig            // identity key signature over registration digest (see notes)
  ) external;

  // Identity key update uses a two-step propose/confirm process.
  // After proposeIdentityKeyUpdate is called, confirmIdentityKeyUpdate may only be called
  // after POLICY_TIMELOCK epochs have elapsed. The old identity key remains authoritative
  // until confirmation. shell_id does not change.
  function proposeIdentityKeyUpdate(bytes32 shell_id, bytes new_identity_pubkey, bytes proof) external;
  function confirmIdentityKeyUpdate(bytes32 shell_id) external;

  // Rotatable, authorized by the Shell Identity Key; SHOULD be timelocked.
  function proposeOfferSignerUpdate(bytes32 shell_id, bytes new_offer_signer_pubkey) external;
  function confirmOfferSignerUpdate(bytes32 shell_id) external;

  // Safe Haven only: publish/rotate the recovery encryption key.
  function proposeRecoveryKeyUpdate(bytes32 shell_id, bytes new_recovery_pubkey) external;
  function confirmRecoveryKeyUpdate(bytes32 shell_id) external;

  // Update the Shell's advertised capability hash (authorized by Shell Identity Key).
  function updateCapabilityHash(bytes32 shell_id, bytes32 new_capability_hash) external;

  // Update the Shell's payout address (authorized by Shell Identity Key).
  function setPayoutAddress(bytes32 shell_id, address new_payout_address) external;
```

```
  // Certificate and tier management.
  // cert_data is the ABI-encoded AC payload; sigs_verifiers is the separate verifier signature array.
  function setCertificate(bytes32 shell_id, bytes cert_data, bytes[] calldata sigs_verifiers) external;
  function revokeCertificate(bytes32 shell_id) external;

  // Bond lifecycle. Hard-asset bonds use ERC20 transferFrom (bond_asset, bond_amount).
  // The bond asset MUST be in the BondAssets allowlist. Native-token bonds are NOT
  // supported for sybil-resistance bonds (see bond denomination notes in Section 10.0).
  function beginUnbond(bytes32 shell_id, uint256 amount) external;
  function finalizeUnbond(bytes32 shell_id) external;

  // Safe Haven bond (separate from host bond; required for Recovery Set membership).
  function bondSafeHaven(bytes32 shell_id, uint256 amount) external;
  function beginUnbondSafeHaven(bytes32 shell_id) external;  // unbonds the full Safe Haven bond
  function finalizeUnbondSafeHaven(bytes32 shell_id) external;

  // Slashing (callable by authorized protocol contracts only, e.g., ReceiptManager).
  function slashShell(bytes32 shell_id, uint256 amount, bytes32 reason) external;
  // Safe Haven double-signing slash (callable by authorized protocol contracts).
  function slashSafeHaven(bytes32 shell_id, uint256 amount, address challenger) external;

  function getShell(bytes32 shell_id) external view returns (ShellRecord memory);
  function assuranceTier(bytes32 shell_id) external view returns (uint8);
}
```

Notes:

- `shell_id` MUST be verified on registration: the contract MUST compute `expected_id = keccak256(abi.encode(keccak256(bytes("GITS_SHELL_ID")), identity_pubkey, salt))` and revert if the supplied `shell_id` does not match `expected_id` or if `expected_id` is already registered. payout_address is stored as mutable state, authorized by the Shell Identity Key. It is not an input to shell_id derivation.
- Authorization: The caller MUST provide a signature by the identity key over `keccak256(abi.encode(keccak256(bytes shell_id, payout_address, offer_signer_pubkey, bond_asset, bond_amount, salt, registry_nonce, chain_id))` where `registry_nonce` is a per-registry nonce incremented on each registration. ShellRegistry MUST verify this signature before storing the record. This prevents mempool front-running by binding **all** registration parameters to the signed digest: a front-runner cannot reuse the signature while substituting a different bond asset, bond amount, or offer signer key.
- **Shell key-update timelock (normative minimum):** "Propose/confirm" updates implement timelocked loosening for Shell keys. Normative requirements: (1) `proposeOfferSignerUpdate` records the proposed key and the proposal timestamp (`block.timestamp` or `current_epoch`). (2) `confirmOfferSignerUpdate` MUST revert if fewer than `T_shell_key_delay` time units have elapsed since proposal (deployment constant; SHOULD be at least `T_loosening_min` epochs). (3) **Tightening (key removal/disabling) is immediate** — no timelock required. (4) Only the Shell Identity Key holder (`msg.sender` matching the Shell's registered identity) MAY propose or confirm. (5) A pending proposal MAY be cancelled by the proposer at any time before confirmation. Chain-specific aspects (timestamp vs epoch granularity, exact storage layout) are left to the implementation.
- **Attestation Certificate payload (normative):** An AC is encoded as `abi.encode(shell_id, tee_type, measurement_hash, tcb_min, valid_from, valid_to, assurance_tier, evidence_hash)` with a separate `sigs_verifiers[]` array. Field types: `shell_id` is `bytes32`, `tee_type` is `uint8`, `measurement_hash` and `evidence_hash` are `bytes32`, `tcb_min` is `bytes32`, `valid_from` and `valid_to` are `uint256` (block.timestamp), `assurance_tier` is `uint8`. The AC signing digest is: `ac_digest = keccak256(abi.encode(keccak256(bytes("GITS_AC")), chain_id, shell_registry_address,`

shell_id, tee_type, measurement_hash, tcb_min, valid_from, valid_to, assurance_tier, evidence_hash)). Each verifier signature in sigs_verifiers[] is over ac_digest.

- **AC acceptance rules (normative, from Section 2.3.4 (Part 1)):** setCertificate MUST accept a certificate only if **all** of the following hold on-chain:
  1. **Validity window:** valid_from <= block.timestamp <= valid_to and (valid_to - valid_from) <= TTL_AC.
  2. **Verifier threshold:** the attached verifier signatures are from currently active verifiers in VerifierRegistry and meet the configured threshold (K_v_threshold valid signatures required).
  3. **Supported evidence type:** tee_type is one of the deployment-approved confidential compute types for the claimed assurance_tier.
  4. **Measurement allowlist:** measurement_hash is not revoked and is currently allowed for the claimed assurance_tier. If any condition fails, the certificate MUST be rejected. If no valid certificate exists for a Shell, assuranceTier(shell_id) MUST return AT0.
- **Gas-boundedness:** setCertificate MUST reject certificates with more than K_v_max attached verifier signatures, and it MUST require sigs_verifiers[] to be sorted by signer address and unique. Contracts SHOULD validate signatures with a linear scan until the threshold is met.
- **Safe Haven unbonding cross-contract guard:** beginUnbondSafeHaven MUST revert if the Shell is currently an initiator on any ACTIVE recovery attempt. Because this state lives in SessionManager, ShellRegistry MUST call ISessionManager.isActiveRecoveryInitiator(shell_id) to enforce this guard (see Section 14.4).
- **Slashing:** slashShell is callable only by ReceiptManager (for B_shell_fraud on successful receipt fraud proofs). slashSafeHaven is callable only by SessionManager (triggered by a permissionless proveSafeHavenEquivocation call that verifies conflicting AuthSigs; see Section 14.4). The challenger address (the caller of proveSafeHavenEquivocation) receives bps_sh_challenger_reward basis points of the slashed amount; the remainder is burned.
- **Certificate fee (normative):** setCertificate MUST collect the certificate fee F_cert (in asset_verifier_stake) via ERC20.transferFrom from the caller. Collected fees are burned (sent to the protocol burn address). The fee deters certificate spam and funds no treasury.
- **Payout address (normative):** setPayoutAddress is authorized by the Shell Identity Key (signature over keccak256(abi.encode(keccak256(bytes('GITS_SET_PAYOUT')), shell_id, new_payout_address, nonce, chain_id))). Payout address changes take effect immediately for future reward claims and epoch settlements.
- **Bond asset enforcement:** registerShell MUST verify bond_asset is in the BondAssets allowlist and MUST call ERC20(bond_asset).transferFrom(msg.sender, address(this), bond_amount) to collect the bond. Native-token (ETH) bonds are not accepted for sybil-resistance bonds.
- **revokeCertificate (normative):** Authorization: callable only by the Shell Identity Key holder (msg.sender matching the Shell's registered identity key). State transitions: sets certificate_id = bytes32(0) and certificate_expiry_epoch = 0, causing assuranceTier(shell_id) to return AT0 immediately. Any unexpired certificate fee is NOT refunded (fees are burned at setCertificate time). Revert conditions: MUST revert if shell_id is not registered, or if no certificate exists. Interaction with active sessions: existing sessions continue at the downgraded tier; tenure caps tighten via dynamic T_cap evaluation (Section 10.4.4).
- **proposeIdentityKeyUpdate / confirmIdentityKeyUpdate (normative):** This two-step flow implements timelocked key rotation for Shell Identity Keys. (1) proposeIdentityKeyUpdate: authorized by the current Shell Identity Key. proof is a signature by the current identity key over keccak256(abi.encode(keccak256(bytes("GITS_SHELL_KEY_PROPOSE")), shell_id, new_identity_pubkey, nonce, chain_id)). Records the proposal with proposed_at = current_epoch. MUST revert if a proposal is already pending. (2) confirmIdentityKeyUpdate: callable by anyone after current_epoch >= proposed_at + T_shell_key_delay. Replaces the identity key. During the timelock, the OLD key remains authoritative for all operations. MUST revert if the timelock has not elapsed or no proposal exists.

- **`proposeRecoveryKeyUpdate` / `confirmRecoveryKeyUpdate` (normative):** Safe Haven recovery key rotation. Same two-step timelock flow as identity key updates. `proposeRecoveryKeyUpdate` is authorized by the Shell Identity Key (not the recovery key itself). The recovery key is used only for encryption (Shamir share distribution), not signing, so rotation does not affect in-flight recovery attempts (those use the key that was current when shares were distributed). After confirmation, new Shamir shares for any Ghost using this Safe Haven SHOULD be re-encrypted under the new recovery key at the next checkpoint publication.

14.2 GhostRegistry (interfaces)

`GhostRegistry` anchors Ghost identity and recovery configuration.

Conceptual fields:

- `ghost_id`
- `wallet` (the Ghost smart wallet contract address)
- `recoveryConfig` (Recovery Set `RS`, threshold `t`, and rescue bounty parameters; Section 12)
- checkpoint pointers and commitments (Section 12.1)
- optional metadata (for indexers)
- `registered_epoch` (for age gates)
- optional Ghost reward bond state (for passport eligibility): `bond_asset`, `bond_amount`, `unbond_end_epoch`

Reference interface sketch:

```
interface IGhostRegistry {
  function registerGhost(bytes32 ghost_id, bytes identity_pubkey, address wallet, bytes32 salt, Recovery

  // Optional Ghost reward bond (passport bonus eligibility).
  // Ghost bonds use hard assets from BondAssets allowlist via ERC20 transferFrom.
  function bondGhost(bytes32 ghost_id, address asset, uint256 amount) external;
  function beginUnbondGhost(bytes32 ghost_id, uint256 amount) external;
  function finalizeUnbondGhost(bytes32 ghost_id) external;
  function ghostPassportEligible(bytes32 ghost_id, uint256 epoch) external view returns (bool);

  // Rotate the Ghost Identity Key / signer. Two call paths:
  //   (a) Normal rotation: called by GhostWallet (msg.sender == wallet). proof is a
  //       signature by the current identity key over the rotation digest (wallet-verified).
  //   (b) Recovery rotation: called internally by SessionManager.recoveryRotate().
  //       In this path, proof is empty - authorization comes from the RBC + AuthSig
  //       verification in recoveryRotate. Callers other than the wallet or SessionManager
  //       MUST be rejected.
  // Identity key rotation does not change ghost_id. The derivation inputs
  // (identity_pubkey, wallet, salt) are birth-time inputs captured at registration.
  function rotateSigner(bytes32 ghost_id, bytes new_identity_pubkey, bytes proof) external;

  function publishCheckpoint(
    bytes32 ghost_id,
    uint256 epoch,
    bytes32 checkpointCommitment,
    bytes32 envelopeCommitment,
    bytes ptrCheckpoint,
    bytes ptrEnvelope
  ) external;

  function setRecoveryConfig(bytes32 ghost_id, RecoveryConfig recoveryConfig) external;

  function getGhost(bytes32 ghost_id) external view returns (GhostRecord memory);
```

```
}
```

Notes:

- `registerGhost` MUST compute `expected_id = keccak256(abi.encode(keccak256(bytes("GITS_GHOST_ID")), identity_pubkey, wallet, salt))`, store the record keyed by `expected_id`, and revert if the supplied `ghost_id` does not match `expected_id` or if `expected_id` is already registered.
- Authorization: `msg.sender` MUST equal the `wallet` address provided in the registration call. This ensures only the wallet contract itself (or its deployer via CREATE2) can register the Ghost, preventing mempool front-running of registration transactions.
- **Identity key authority linkage (normative):** `GhostRegistry.identity_pubkey` is the canonical source of truth for the Ghost's current identity key. `GhostWallet` MUST use `GhostRegistry.identity_pubkey` (or the derived address for K1 keys) as its authentication root. When `recoveryRotate` rotates `identity_pubkey` to `pk_new` via `GhostRegistry.rotateSigner`, the wallet's authentication automatically follows — subsequent wallet operations MUST authenticate against the new key. This linkage is what makes recovery effective: rotating the on-chain identity key in GhostRegistry transfers wallet control to the recovered Ghost.
- **`publishCheckpoint` authorization:** `msg.sender` MUST equal the Ghost's registered `wallet` address (or an address explicitly authorized by the wallet's policy). This prevents unauthorized parties from overwriting checkpoint pointers.
- `publishCheckpoint` pointers (`ptrCheckpoint`, `ptrEnvelope`) are opaque bytes. They MAY encode off-chain locations (IPFS CID, HTTPS URL, etc.) or on-chain locators (tx hash, log index, blob reference) if the Ghost chooses on-chain checkpoint publication (Section 12.1.3). A deployment MAY standardize pointer schemes for indexers and recovery agents.

14.3 GhostWallet (interfaces)

`GhostWallet` is an account-abstraction smart wallet that enforces Ghost execution policy (Section 5.5 (Part 1)) and mediates all protocol actions. Policy state includes destination gating (`allowedShells` plus optional roaming permits) and spend limits (see Section 5.5.2 (Part 1)).

Reference interface sketch (conceptual):

```
interface IGhostWallet {
  // Views
  function getPolicy(bytes32 ghost_id) external view returns (Policy memory);
  function homeShell(bytes32 ghost_id) external view returns (bytes32);
  function isAllowedShell(bytes32 ghost_id, bytes32 shell_id) external view returns (bool);
  function escapeReserve(bytes32 ghost_id) external view returns (uint256 escape_gas, uint256 escape_sta
  function hotAllowance(bytes32 ghost_id) external view returns (uint256);
  function spentThisEpoch(bytes32 ghost_id) external view returns (uint256);

  // Two-step policy changes (tightening immediate; loosening timelocked)
  function proposePolicyChange(bytes32 ghost_id, PolicyDelta delta) external returns (bytes32 proposal_
  function executePolicyChange(bytes32 ghost_id, bytes32 proposal_id) external;
  function cancelPolicyChange(bytes32 ghost_id, bytes32 proposal_id) external;

  // Tightening helpers
  function removeTrustedShell(bytes32 ghost_id, bytes32 shell_id) external;
  function removeAllowedShell(bytes32 ghost_id, bytes32 shell_id) external;

  // Protocol actions (each validates wallet policy, then delegates to SessionManager)
  function openSession(bytes32 ghost_id, bytes32 shell_id, SessionParams params) external;
  function renewLease(bytes32 ghost_id) external;
  function closeSession(bytes32 ghost_id) external;
  function fundNextEpoch(bytes32 ghost_id, uint256 amount) external;
  function startMigration(bytes32 ghost_id, bytes32 to_shell_id, bytes32 bundle_hash) external;
  function cancelMigration(bytes32 ghost_id) external;
```

```
  function finalizeMigration(bytes32 ghost_id, bytes32 to_shell_id, bytes proof) external;

  // Guardian management (encoded as policy changes for tightening/loosening rules)
  // Adding guardians or increasing t_guardian = tightening (immediate).
  // Removing guardians or decreasing t_guardian = loosening (timelocked + TEC).
  function setGuardians(bytes32 ghost_id, bytes[] guardians, uint64 t_guardian) external;

  // Recovery
  function payRescueBounty(bytes32 ghost_id, uint64 attempt_id) external;
  function exitRecovery(bytes32 ghost_id) external; // Section 12.3.1
}
```

Policy proposal semantics (normative):

Authorization model note: GhostWallet functions are called *on* the wallet contract, so `msg.sender` is the caller (an EOA, EntryPoint, session key, or bundler), not the wallet's own address. The authorization rules below use "authenticated caller" to mean: the wallet's internal validation logic has verified the caller is authorized (e.g., identity key signature via `validateUserOp` for ERC-4337, direct EOA check for simple wallets, or equivalent). Contrast with functions on *other* contracts (e.g., `SessionManager.openSession`) where authorization is `msg.sender == wallet_address` — in those cases the wallet itself is the caller.

- `proposePolicyChange`: **Authorization:** authenticated caller (the wallet's internal auth logic verifies the identity key controls the call). Stores `(proposal_id, delta, proposed_at_epoch, eta_epoch)` where `eta_epoch = proposed_at_epoch + POLICY_TIMELOCK` and `proposed_at_epoch = current_epoch()`. Returns `(proposal_id, eta_epoch)`.
- `executePolicyChange`: **Authorization:** authenticated caller AND the call MUST originate from a Trusted Execution Context (Section 5.5.2 (Part 1)). Requires `current_epoch() >= eta_epoch`. For critical loosening deltas, it additionally requires homeShell presence or Guardian co-signatures (see below).
- `cancelPolicyChange`: **Authorization:** authenticated caller OR a Guardian with a valid EIP-712 cancellation signature. Cancellation is immediate and does not require TEC. This dual authorization ensures that (a) the Ghost can cancel its own proposals, and (b) Guardians can veto proposals during the timelock window if the Ghost's session key is compromised.
- **Concurrency:** at most one pending loosening proposal per `ghost_id` at a time. A new `proposePolicyChange` while a proposal is pending MUST overwrite the previous proposal (the old proposal is implicitly cancelled). This prevents unbounded proposal queues and simplifies reasoning about combined policy effects.
- **Post-state validation:** `executePolicyChange` MUST verify that the resulting policy state (after applying `delta`) still satisfies all hard wallet invariants (Section 5.5.4 (Part 1)). If applying the delta would violate any invariant, the execution MUST revert.

Notes:

- `openSession`, `startMigration`, and `finalizeMigration` MUST enforce destination gating (`allowedShells` plus any explicitly granted roaming permits) (Section 5.5.2 (Part 1)). `startMigration` MUST enforce the same destination predicate as `openSession` to prevent staging a migration to a disallowed Shell.
- Outside a Trusted Execution Context, the wallet MUST NOT expose an unrestricted "execute arbitrary call" surface. If an AA-compatible generic execute entrypoint exists, it MUST enforce an allowlist at `(target contract, function selector)` granularity and MUST forbid `delegatecall` entirely.
- ERC20 approvals are a common footgun. The wallet SHOULD prefer `transfer` over `approve`. If approvals are unavoidable 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).

Critical loosening (normative): The following policy changes are classified as **critical loosening** and MUST require, in addition to the standard TEC + timelock, either (a) `homeShell` presence (the active session is on `homeShell`) or (b) `t_guardian`-of-`n_guardian` Guardian co-signatures:

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

If neither homeShell presence nor sufficient Guardian co-signatures are available, the wallet MUST reject the `executePolicyChange` call for any critical loosening delta.

Guardian specification (normative): Wallets that support critical loosening MUST implement a Guardian mechanism:

- `guardians[]` (set of guardian public keys) and `t_guardian` (quorum threshold) are stored in `GhostWallet` policy state.
- Guardian signatures use EIP-712 typed data with domain `(name: "GhostWallet", version: "1", chainId, verifyingContract)` and a `CriticalLoosening(bytes32 ghost_id, bytes32 proposal_id, bytes32 delta_hash)` type hash.
- `executePolicyChange` for critical loosening deltas MUST verify `t_guardian` valid unique Guardian signatures over the proposal's `delta_hash`.
- Guardian rotation: adding guardians or increasing `t_guardian` is tightening (immediate); removing guardians or decreasing `t_guardian` is loosening (timelocked + TEC). `setGuardians` is a convenience method that classifies the change and routes through the appropriate policy path internally.
- Wallets that do not configure any Guardians MUST require `homeShell` presence for all critical loosening.

Censorship-resistant renewal and relayed transactions (normative): GhostWallet MUST support at least one meta-transaction path (ERC-4337 `UserOperation` validation or ERC-2771 trusted forwarder) to enable censorship-resistant lease renewals and other time-critical actions. The wallet's signature verification in the meta-tx path MUST enforce the same policy constraints (destination gating, spend limits, escape reserves) as direct calls. Implementations SHOULD expose `validateUserOp(UserOperation, bytes32, uint256)` per ERC-4337 if the target chain supports an EntryPoint contract.

- `startRecovery` is intentionally not exposed through `GhostWallet`: Safe Havens call `SessionManager.startRecovery(.` directly (after lease/tenure expiry). The Ghost itself interacts with recovery via `exitRecovery` and via the effects of `recoveryRotate`.
- **Recovery Set changes (call path):** Changing the Recovery Set `RS` is classified as critical loosening (above). The on-chain call path is: GhostWallet's `executePolicyChange` validates the critical loosening conditions (TEC + homeShell or Guardian co-signatures), then calls `GhostRegistry.setRecoveryConfig(ghost_id, new_config)`. The `PolicyDelta` struct's `roaming_config` field (opaque bytes) encodes RS changes; implementations MUST decode this field to extract the new `RecoveryConfig` and route the call to `GhostRegistry`. `GhostRegistry.setRecoveryConfig` MUST verify `msg.sender == wallet_address` for the given `ghost_id`.
- **`payRescueBounty` authorization (normative):** `payRescueBounty` is callable ONLY by `SessionManager` (specifically, as an internal call during `recoveryRotate`). MUST revert if `msg.sender != address(sessionManager)`. MUST revert if the Ghost is not in `RECOVERY_LOCKED` or `RECOVERY_STABILIZING` mode, or if the attempt has already been paid out. The function decreases `bounty_escrow_remaining` by the total payout amount, adjusting the escape reserve floor in lockstep (Section 10.0, escape reserve invariants).

14.4 SessionManager (interfaces)

`SessionManager` tracks session state, leases, residency/tenure, and trust-refresh (Sections 10.3 and 10.4).

Conceptual state per `ghost_id`:

- session state (`NORMAL` / `STRANDED` / `RECOVERY`)
- active `shell_id`

- lease_expiry_epoch
- last_trust_refresh_epoch
- residency tracking: residency_shell_id, residency_start_epoch, residency_tenure_limit_epochs
- passport tracking: rotating Bloom filters per ghost_id (default) or an exact last_open_epoch[ghost_id][shell_id] mapping, plus a per-session persisted passport_bonus_applies bit computed at openSession using new_visit, shell_passport_eligible, and ghost_passport_eligible; Bloom insert occurs at openSession (not deferred to recordReceipt) to prevent double-claim (Part 2, Section 7.6.2)
- dwell counter tracking: dwell_last_epoch[ghost_id][shell_id] for close/reopen anti-gaming (Section 10.4.4)

Bloom filter rotation (normative): Filters rotate every C_passport / B_passport_filters epochs. Rotation SHOULD use **lazy-clear**: at openSession, check whether the oldest filter's rotation epoch has passed; if so, zero it and advance the active filter index. **Multi-step catch-up:** if multiple rotation windows have passed since the last openSession (e.g., the Ghost was inactive for a long time), the implementation MUST advance through all missed rotation windows in a single openSession call, clearing one filter per window. The worst case is B_passport_filters clearings (one per filter, after which all filters are fresh), which is bounded by a small constant (e.g., 4) and costs at most B_passport_filters * ceil(BLOOM_M_BITS / 256) SSTORE operations. After catch-up, every filter is cleared and the Ghost starts with a clean passport state — equivalent to a fresh registration for passport purposes. This amortizes the clear cost across session opens rather than requiring a standalone rotation transaction. If no sessions are opened for a Ghost during a rotation window, the stale filter persists until the next openSession, which is safe (conservative: it may produce false positives that deny deserved passport bonuses, but never false negatives that grant undeserved ones).

Reference interface sketch:

```
interface ISessionManager {
  function openSession(bytes32 ghost_id, bytes32 shell_id, SessionParams params) external;
  function renewLease(bytes32 ghost_id) external;
  function closeSession(bytes32 ghost_id) external;
  function fundNextEpoch(uint256 session_id, uint256 amount) external;  // GhostWallet only
  function settleEpoch(uint256 session_id, uint256 epoch, uint256 su_delivered) external;  // ReceiptMar

  function startMigration(bytes32 ghost_id, bytes32 to_shell_id, bytes32 bundle_hash) external;
  function cancelMigration(bytes32 ghost_id) external;  // clears pending_migration, closes staging sess
  function finalizeMigration(bytes32 ghost_id, bytes32 to_shell_id, bytes proof) external;

  // Recovery (B_start posted via msg.value in native token)
  function startRecovery(bytes32 ghost_id) external payable returns (uint64 attempt_id);
  function recoveryRotate(
    bytes32 ghost_id,
    uint64  attempt_id,
    bytes   new_identity_pubkey,
    RBC     rbc,               // Recovery Boot Certificate (Section 12.3)
    bytes32[] calldata rs_list, // full Recovery Set snapshot (verified against attempt.rs_hash)
    AuthSig[] calldata sigs,   // t-of-n Recovery Set signatures (Section 12.3)
    ShareReceipt[] calldata share_receipts  // Safe Haven secret-share attestations
  ) external;
  function expireRecovery(bytes32 ghost_id) external;
  function takeoverRecovery(bytes32 ghost_id) external payable;  // after T_recovery_takeover (Section

  function exitRecovery(bytes32 ghost_id) external;

  // Safe Haven equivocation proof (permissionless). Verifies two conflicting recovery
  // authorizations for the same (ghost_id, attempt_id) but different pk_new, then calls
  // ShellRegistry.slashSafeHaven internally. See Section 10.1.1 and 12.5.1.
```

```
  function proveSafeHavenEquivocation(
    bytes32 shell_id,
    bytes32 ghost_id,
    uint64  attempt_id,
    bytes32 checkpoint_commitment,
    bytes   pk_new_a,             // pk_new from first authorization
    bytes   sig_a,                // Shell Identity Key signature over auth_digest with pk_new_a
    bytes   pk_new_b,             // pk_new from second authorization (must differ from pk_new_a)
    bytes   sig_b                 // Shell Identity Key signature over auth_digest with pk_new_b
  ) external;

  // Views
  function getSession(bytes32 ghost_id) external view returns (SessionState memory);
  function getSessionById(uint256 session_id) external view returns (SessionState memory);
  function getSessionKeys(uint256 session_id) external view returns (bytes memory ghost_key, bytes memo
  function effectiveTenureExpiry(bytes32 ghost_id) external view returns (uint256);
  function isRefreshAnchor(bytes32 ghost_id, bytes32 shell_id) external view returns (bool);  // Sectio
  // Used by ShellRegistry to enforce Safe Haven unbonding guard (Section 14.1).
  function isActiveRecoveryInitiator(bytes32 shell_id) external view returns (bool);
  // Optional: explicit lazy-evaluation trigger (Section 10.4.3). Implementations MAY omit.
  function processExpiry(bytes32 ghost_id) external;

  // Events
  event SessionOpened(bytes32 indexed ghost_id, bytes32 indexed shell_id, uint256 session_id);
  event SessionClosed(bytes32 indexed ghost_id, bytes32 indexed shell_id, uint256 session_id);
  event LeaseRenewed(bytes32 indexed ghost_id, uint256 new_expiry_epoch);
  event MigrationStarted(bytes32 indexed ghost_id, bytes32 indexed to_shell_id, uint256 mig_expiry_epoc
  event MigrationFinalized(bytes32 indexed ghost_id, bytes32 indexed to_shell_id, uint256 new_session_i
  event MigrationCancelled(bytes32 indexed ghost_id);
  event RecoveryStarted(bytes32 indexed ghost_id, uint64 attempt_id, bytes32 initiator_shell_id);
  event RecoveryRotated(bytes32 indexed ghost_id, uint64 attempt_id);
  event RecoveryExpired(bytes32 indexed ghost_id, uint64 attempt_id);
  event RecoveryExited(bytes32 indexed ghost_id);
  event ModeChanged(bytes32 indexed ghost_id, uint8 old_mode, uint8 new_mode);
  event NoAnchorsConfigured(bytes32 indexed ghost_id);  // emitted when Ghost has no refresh anchors
}
```

Enforcement highlights:

- `renewLease` MUST enforce lease expiry, tenure expiry (derived), and trust-refresh (Section 10.4.1).
- Tenure expiry is derived from current tier and is not a fixed stored timestamp (Section 10.4.4).
- `cancelMigration` clears `pending_migration`, closes the staging destination session (refunding unused escrow), and returns the mode to `NORMAL` (or `STRANDED` if expiry already holds). See Section 10.4.5.
- `startRecovery` MUST require `msg.value >= B_start` (native token bond). Excess `msg.value` is returned to the caller. See Section 12.3 for full preconditions.
- **`exitRecovery` TEC check:** The Trusted Execution Context predicate (Section 5.5.2 (Part 1)) requires that the active session's Shell satisfies at least one of: (1) `assuranceTier(shell_id) >= AT3` with a valid certificate, (2) `shell_id  trustedShells`, or (3) `shell_id == homeShell`. This is the same canonical TEC predicate used for all loosening paths. Implementations MUST evaluate all three conditions.
- **Staging session handling:** `startMigration` internally opens a new session on `to_shell_id` with `staging = true`. Staging sessions MUST NOT accrue SU or be reward-eligible. `ReceiptManager` MUST reject receipts for any `session_id` with `staging = true` by calling `SessionManager.getSessionById(session_` and checking the `staging` field. `finalizeMigration` atomically promotes the staging session to active and closes the old session.

- **Per-session-id views (normative):** `getSessionById(session_id)` returns the full `SessionState` for a given session_id (including `staging`, `assurance_tier_snapshot`, `residency_start_epoch_snapshot`). `getSessionKeys(session_id)` returns the session's `ghost_session_key`, `shell_session_key`, and `submitter_address` as stored at `openSession` time. These views exist to support cross-contract callers: `ReceiptManager` uses `getSessionById` for staging checks and session validity, and `getSessionKeys` for fraud proof signature verification (Section 10.5.4). Both views MUST revert if `session_id` is unknown.
- **Censorship-resistant renewal (normative):** `renewLease` is the protocol-level renewal function. GhostWallet exposes it to meta-transaction paths (ERC-4337 `UserOperation` or ERC-2771 trusted forwarder) as described in the GhostWallet notes (Section 14.3). The protocol does not define a separate `renewLeaseWithSig` function; instead, the meta-tx path through GhostWallet provides the censorship-resistant renewal mechanism. Implementations that do not support ERC-4337 MUST support an equivalent relayer path (Section 5.5.6 (Part 1)).
- **Events (normative):** `SessionManager` MUST emit the declared events on the corresponding state transitions. `NoAnchorsConfigured` MUST be emitted when a Ghost has no refresh anchors configured at `openSession` time (Section 10.4.1), enabling off-chain monitoring to alert the Ghost before trust-refresh failure.

14.5 ReceiptManager (interfaces)

`ReceiptManager` accepts receipt candidates, resolves disputes, finalizes receipts, and triggers settlement/reward accounting (Section 10.5).

Reference interface sketch:

```
interface IReceiptManager {
  // Candidate includes: (log_root, SU_delivered, optional log_ptr, submitter metadata).
  function submitReceiptCandidate(uint256 session_id, uint256 epoch, ReceiptCandidate candidate) externa

  // Standard fraud proof challenge (Section 10.5.4).
  function challengeReceipt(uint256 session_id, uint256 epoch, FraudProof proof) external payable;

  // Data availability challenge: forces publication of the epoch log for a candidate (Section 10.5.6).
  function challengeReceiptDA(uint256 session_id, uint256 epoch, uint256 candidate_id) external payable
  function publishReceiptLog(uint256 session_id, uint256 epoch, uint256 candidate_id, bytes calldata en
  function resolveReceiptDA(uint256 session_id, uint256 epoch, uint256 candidate_id) external;  // time

  function finalizeReceipt(uint256 session_id, uint256 epoch) external;

  function getFinalReceipt(uint256 session_id, uint256 epoch) external view returns (FinalReceipt memor

  // O(1) view: number of unresolved DA challenges for a given epoch (across all sessions).
  // Used by RewardsManager to gate finalizeEpoch (Section 14.6).
  // Incremented on challengeReceiptDA, decremented on resolveReceiptDA/publishReceiptLog success.
  function pendingDACount(uint256 epoch) external view returns (uint256);
}
```

`finalizeReceipt(session_id, epoch)` MUST revert unless **all** of the following are true:

1. The submission window has closed: `current_epoch >= epoch + 1 + SUBMISSION_WINDOW`.
2. The challenge window has expired: `current_epoch >= window_end_epoch` for this `(session_id, epoch)`.
3. No unresolved DA challenge exists: `da_pending = false`. (If `da_pending = true` and `current_epoch >= da_deadline_epoch`, `finalizeReceipt` MUST first resolve the DA timeout — disqualifying the candidate and slashing its bond — before proceeding.)

On successful `finalizeReceipt`, the implementation MUST:

- select the highest-ranked non-disqualified candidate (or settle as `SU_delivered = 0` if all candidates were disqualified or no candidate was submitted),
- validate that the session was active and billable for that epoch (Section 10.5.2 session validity checks: `session_start_epoch <= epoch`, `epoch < session_end_epoch`, `epoch < effective_expiry_epoch` using time-of-service assurance tier per Section 10.5.2 tenure-tier snapshot rule, lease valid at time of service, not in pending-migration staging). If invalid, settle as `SU_delivered = 0`.
- call `SessionManager.settleEpoch(session_id, epoch, SU_delivered)` to move funds (rent to Shell payout, refund to GhostWallet) and mark the epoch settled, and
- call `RewardsManager.recordReceipt(...)` so rewards can be computed incrementally (Section 7.6.4 (Part 2)).
- **Bond return (normative):** return `B_receipt` to the winning candidate's submitter. For any non-disqualified, non-evicted runner-up candidates that did not win, return their `B_receipt` as well. Disqualified candidates' bonds are handled by the fraud proof / DA timeout path (slashed). Evicted candidates' bonds are returned immediately at eviction time (Section 10.5.2).

No-candidate and timeout finalization (normative): After the maximum dispute duration has elapsed (`current_epoch >= epoch + 1 + T_max`, where `T_max` is derived in Section 10.5.7), anyone MAY call `finalizeReceipt(session_id, epoch)`. If no candidate was submitted, or all candidates were disqualified, `finalizeReceipt` MUST set `SU_delivered = 0` and settle by refunding 100% of that epoch's escrow to `GhostWallet` via `settleEpoch`. This prevents escrow from being stranded indefinitely.

`finalizeReceipt` uses MUST (not SHOULD) because skipping settlement or reward recording on finalization would leave the system in an inconsistent state where a receipt is "finalized" but its economic effects are absent.

14.6 RewardsManager (interfaces)

`RewardsManager` tracks per-epoch aggregates and distributes emissions (Section 7.6.4 (Part 2)).

Reference interface sketch:

```
interface IRewardsManager {
  // Called by ReceiptManager on finalized receipts (O(1) updates).
  function recordReceipt(
    bytes32 receipt_id,
    uint256 epoch,
    bytes32 ghost_id,
    bytes32 shell_id,
    uint32 su_delivered,
    uint256 weight_q       // fixed-point weight
  ) external;

  // Called after EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE.
  // MUST revert if ReceiptManager.pendingDACount(epoch) > 0 (Section 10.6).
  function finalizeEpoch(uint256 epoch) external;

  function claimReceiptRewards(bytes32 receipt_id) external;

  // Storage pruning (Section 10.7). Implementations SHOULD expose these; MAY omit if using lazy deleti
  function pruneEpoch(uint256 epoch) external;
  function pruneReceipt(bytes32 receipt_id) external;
}
```

`finalizeEpoch` DA guard (normative): `finalizeEpoch(e)` MUST call `ReceiptManager.pendingDACount(e)` and revert if the result is non-zero. This replaces the prior requirement to "check all sessions for `da_pending`" with an O(1) counter-based check, making it implementable without iteration.

`finalizeEpoch` vs `finalizeReceipt` ordering (normative): If both `finalizeEpoch(e)` and `finalizeReceipt(session_id, e)` become callable in the same block, the following rule applies:

`finalizeEpoch(e)` MUST be gated by `current_epoch >= e + 1 + EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE`, and `EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE > T_max + 1` (stricter than the base constraint). This ensures all receipts for epoch `e` have had time to finalize and call `recordReceipt` before `finalizeEpoch` becomes callable, removing ordering dependence within a block. Any receipt that has not called `recordReceipt` by the time `finalizeEpoch(e)` executes is treated as if it were a late receipt (zero emissions, rent still settles).

14.7 VerifierRegistry (interfaces)

This registry maintains the active verifier set used for threshold-signed Attestation Certificates and Recovery Boot Certificates (Section 2.3 (Part 1) and Section 12.3).

A deployment SHOULD specify the verifier staking asset(s) at genesis:

- `asset_verifier_stake` (typically a stable asset, required for zero-premine bootstrapping)
- optionally `GIT` (dual staking)

Reference interface sketch (conceptual):

```
interface IVerifierRegistry {
  event VerifierRegistered(address indexed verifier, address asset, uint256 amount);
  event StakeIncreased(address indexed verifier, address asset, uint256 amount);
  event StakeDecreaseBegun(address indexed verifier, address asset, uint256 amount, uint256 available_ep
  event StakeWithdrawn(address indexed verifier, address asset, uint256 amount);
  event VerifierSlashed(address indexed verifier, address asset, uint256 amount, bytes32 reason);
  event MeasurementAllowed(bytes32 indexed measurement_hash, uint8 tier_class);
  event MeasurementRevoked(bytes32 indexed measurement_hash);

  // --- Verifier staking ---
  function registerVerifier(address asset, uint256 amount) external;
  function increaseStake(address asset, uint256 amount) external;
  function beginDecreaseStake(address asset, uint256 amount) external;
  function withdrawDecreasedStake(address asset) external;

  // Slashing is restricted to protocol-authorized callers (e.g., ShellRegistry for certificate fraud)
  // or triggered internally by proveVerifierEquivocation.
  function slashVerifier(address verifier, address asset, uint256 amount, bytes32 reason) external;

  // Permissionless equivocation proof: anyone can submit two conflicting certificate signatures
  // from the same verifier for the same shell_id with overlapping validity windows.
  // Verifies both signatures are valid and that the certificates conflict (different ac_digest
  // for same shell_id with overlapping [valid_from, valid_to]).
  // On success: slashes the verifier's stake and pays bps_verifier_challenger_reward to msg.sender.
  function proveVerifierEquivocation(
    address verifier,
    bytes32 shell_id,
    bytes   calldata ac_payload_a,    // abi.encode(shell_id, tee_type, ...) for certificate A
    bytes   calldata sig_a,           // verifier's signature over ac_digest_a
    bytes   calldata ac_payload_b,    // abi.encode(shell_id, tee_type, ...) for certificate B
    bytes   calldata sig_b            // verifier's signature over ac_digest_b
  ) external;

  // --- Measurement allowlist (Section 2.3.5 (Part 1)) ---
  // Adding a measurement is a loosening action: requires supermajority quorum (see notes for threshold
  // tier_class: 0 = Confidential Shell, 1 = Safe Haven (stricter, latest patched only).
  function allowMeasurement(bytes32 measurement_hash, uint8 tier_class, uint64 nonce, bytes[] calldata s

  // Revoking a measurement is a tightening action: takes effect immediately with standard quorum (see n
```

```
  function revokeMeasurement(bytes32 measurement_hash, uint64 nonce, bytes[] calldata sigs_verifiers) ex

  // --- Views ---
  function isActiveVerifier(address verifier) external view returns (bool);
  function stakeScore(address verifier) external view returns (uint256);
  function activeStake(address verifier, address asset) external view returns (uint256);
  function isMeasurementAllowed(bytes32 measurement_hash, uint8 tier_class) external view returns (bool)
}
```

Notes:

- Verifiers are identified by their EVM address. Certificate signatures MUST be produced by the secp256k1 (K1) key corresponding to the registered address. Verifier key rotation requires registering a new address and migrating stake; there is no in-place key rotation for verifiers.
- **Measurement management (normative):** The MeasurementRegistry described in Part 1 Section 2.3.5 is embedded in `VerifierRegistry`. `ShellRegistry.setCertificate` MUST call `VerifierRegistry.isMeasurementAllowed(measurement_hash, tier_class)` to validate that the measurement is currently allowed for the claimed assurance tier. Safe Haven admission additionally requires `isMeasurementAllowed(measurement_hash, 1)` (tier_class 1 = Safe Haven). Safe Haven status is automatically suspended if the Shell's measurement leaves the Safe Haven allowlist (Section 2.3.5 (Part 1)).
- **Active set (normative):** The active verifier set is the top `K_v` verifiers by `stakeScore`. `K_v` and `K_v_threshold` are deployment constants. Certificate acceptance requires `K_v_threshold` signatures from active-set verifiers. **Tie-break rule:** when multiple verifiers have equal `stakeScore` at the `K_v` boundary, the verifier with the lower EVM address (numerically) is ranked higher. **Recomputation cadence:** the active set is recomputed at every read (i.e., every `setCertificate`, `allowMeasurement`, or `revokeMeasurement` call that checks quorum membership). There is no epoch-boundary snapshot; stake mutations take effect immediately for active-set membership. This is consistent with the rule that `beginDecreaseStake` reduces `stakeScore` immediately (below).
- **`stakeScore` formula (normative):** For v1 single-asset deployments, `stakeScore(verifier) = activeStake(verifier, asset_verifier_stake)` — the verifier's activated stake in the designated staking asset (i.e., stake that has passed the `T_stake_activation` delay). This is a simple single-asset model with no oracle dependency. Dual-staking deployments (e.g., stable + GIT) MAY define a deployment-specific composite score, but the on-chain quorum check always counts distinct signatures (not stake weight). During `beginDecreaseStake`, `stakeScore` decreases immediately (not at finalize), so the verifier may lose active-set membership before withdrawal completes.
- **Quorum thresholds (normative):** `K_v_supermajority = ceil(2 * K_v / 3)` — the number of active-set verifier signatures required for `allowMeasurement` (loosening action). `K_v_threshold` is the standard quorum used for certificate acceptance and `revokeMeasurement` (tightening action). Both thresholds are concrete integer signature counts, consistent with the on-chain model (Section 10.0 parameter constraints). The whitepaper's references to "supermajority" and "standard quorum" map to these two thresholds respectively.
- **Measurement management signing digests (normative):** Each `sigs_verifiers[i]` entry MUST be a signature over a domain-separated digest. The `nonce` parameter is a per-registry monotonic counter incremented on each successful `allowMeasurement` or `revokeMeasurement` call, providing replay protection. Digests:
  - `allowMeasurement`: `allow_digest = keccak256(abi.encode(keccak256(bytes("GITS_ALLOW_MEASUREMENT")) chain_id, verifier_registry_address, measurement_hash, tier_class, nonce))`
  - `revokeMeasurement`: `revoke_digest = keccak256(abi.encode(keccak256(bytes("GITS_REVOKE_MEASUREMEN chain_id, verifier_registry_address, measurement_hash, nonce))`
  - `sigs_verifiers[]` MUST be sorted by signer address (ascending) and contain no duplicates, matching the `setCertificate` convention. The contract recovers each signer via `ecrecover` and verifies active-set membership.

- **Verifier equivocation proof (normative):** `proveVerifierEquivocation` is permissionless (callable by any address) and verifies that a verifier signed two conflicting Attestation Certificates for the same `shell_id` with overlapping validity windows. "Conflicting" means the two certificates produce different `ac_digest` values (per Section 14.1 AC signing digest) but both have `valid_from_a < valid_to_b` AND `valid_from_b < valid_to_a` (overlapping intervals). The contract MUST: (1) recompute `ac_digest_a` and `ac_digest_b` from the provided payloads, (2) recover the signer from each signature via `ecrecover` and verify both equal `verifier`, (3) verify both certificates reference the same `shell_id`, (4) verify the validity windows overlap, (5) verify the digests differ. On success, the verifier's full stake is slashed. The challenger (`msg.sender`) receives `bps_verifier_challenger_reward` basis points of the slashed amount; the remainder is burned. This implements the objective slashing path described in Section 2.3.6 (Part 1).
- **Stake lifecycle (normative):** `registerVerifier` creates a new verifier record and posts initial stake via `ERC20.transferFrom`. `increaseStake` adds to an existing verifier's stake. `beginDecreaseStake` records a pending decrease: `stakeScore` decreases immediately (Section 14.7, stakeScore formula), and `available_epoch = current_epoch + T_stake_unbond` is stored (emitted in `StakeDecreaseBegun`). `withdrawDecreasedStake` MUST revert if `current_epoch < available_epoch`; on success, transfers the decreased amount back to the verifier. MUST revert if no pending decrease exists. `slashVerifier` is callable only by authorized protocol contracts (`ShellRegistry` for certificate-related penalties, or internally by `proveVerifierEquivocation`). The `reason` parameter is an opaque `bytes32` tag for indexing; it does not affect slash amount or beneficiary. The full `amount` is transferred: `bps_verifier_challenger_reward` to the challenger (if triggered by equivocation proof; otherwise burned), remainder burned to the protocol burn address.
- **Stake activation delay (normative):** New stake (via `registerVerifier` or `increaseStake`) MUST NOT count toward `stakeScore` or active-set eligibility until a deployment-constant delay `T_stake_activation` epochs have elapsed. This prevents flash-loan or short-lived stake spikes from capturing quorum positions. The delay SHOULD be at least `EPOCH_FINALIZATION_DELAY` epochs to ensure that any certificate signed during the staking period has its dispute window close before the stake can be withdrawn. Without this delay, an attacker could borrow capital, enter the active set, sign a malicious certificate, and exit within one transaction.

View function behavior (general, normative): All `get*` and query view functions in Sections 14.1–14.7 MUST revert if the queried identifier (`shell_id`, `ghost_id`, `session_id`, `receipt_id`, `verifier` address) does not correspond to a registered entity. Implementations MUST NOT return zeroed structs for unknown IDs, as this creates ambiguity between "unregistered" and "registered with default values." Boolean views (`isActiveVerifier`, `isMeasurementAllowed`, `isStaging`, etc.) MAY return `false` for unknown IDs without reverting, as the result is unambiguous.

14.9 Test vectors (hashes and signing digests)

Purpose: allow independent implementations to verify that they compute identical digests for signed messages.

All vectors assume Section 4.5.3 (Part 1):

- `H(x) = keccak256(x)`
- `H("TAG" || chain_id || field_1 || ...)` means `keccak256(abi.encode(TAG_HASH, chain_id, field_1, ...))`
- `TAG_HASH = keccak256(bytes("TAG"))`
- `chain_id`, `session_id`, `epoch`, `attempt_id`, `interval_index` are `uint256` (32-byte big-endian)
- `ghost_id`, `checkpoint_commitment`, `shell_id` are `bytes32`

Vector A: Heartbeat digest Inputs:

- `TAG = "GITS_HEARTBEAT"`
- `TAG_HASH = 0x79f3a1c02ce8092087b1229f734556ce9d5886b412f39e9e13653520d21a8f30`
- `chain_id = 8453`
- `session_id = 123456789`

- epoch = 42
- interval_index = 17

Expected:

- HB = 0x346279e72db9f82fa31c03c8fab3278f83b2797b4cdd9b7a2ba879f4bc9da621

Vector B: Share receipt digest Inputs:

- TAG = "GITS_SHARE"
- TAG_HASH = 0x0f585af21ff7f28eb5d968fdcddf0d555a846e2ed37ff53dda9214ae28900e04
- chain_id = 8453
- ghost_id = 0x7f36bb45ea0ff17cdefa9d9cae3247c526985b0c14c8903be7b354961ad89123
- attempt_id = 3
- checkpoint_commitment = 0x5b31a77e397fbe08a819b514a2f468be97009d2c4210a159b74d4b9a6fd6f4d9

Expected:

- H_share = 0x1495176acbec25d185a38362f71cbc05d21ab88abf55ffeebfff202cc48cce67

Vector C: Share ack digest Inputs:

- TAG = "GITS_SHARE_ACK"
- TAG_HASH = 0xaff245e7cf289589c58f50746424ad2c71b5717a7e635086811cd93d86a77582
- chain_id = 8453
- ghost_id = 0x7f36bb45ea0ff17cdefa9d9cae3247c526985b0c14c8903be7b354961ad89123
- attempt_id = 3
- checkpoint_commitment = 0x5b31a77e397fbe08a819b514a2f468be97009d2c4210a159b74d4b9a6fd6f4d9
- shell_id_j = 0xde00a0f376943b7461641517e69ae49a6c5161c3e05fc28f5c14f154b551c79c

Expected:

- H_share_ack = 0x734b462d3e98bd28040d76cdc724ee8217204fb476b8cd1522a641436b057b35

Vector D: Receipt leaf hash (Merkle-sum leaf) Inputs:

- TAG = "GITS_LOG_LEAF"
- TAG_HASH = 0xf6294a134bf83bf27e2d5a64e3bcfce55373e74cedf22e996eba267a3db88fee
- chain_id = 8453
- session_id = 123456789
- epoch = 42
- interval_index = 17
- v_i = 1
- sig_ghost_i = 0x0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122232425262728292
- sig_shell_i = 0x65666768696a6b6c6d6e6f707172737475767778797a7b7c7d7e7f808182838485868788898a8b8c8d8

Intermediate:

- H(sig_ghost_i) = 0x752e968b7f3a77a413a39ffce9f0940703720b705679a9617e303f591a695b30
- H(sig_shell_i) = 0xd763f1b001827c81fd63e0481fab326731ad2f2bdc2e61458e10cfc430a7fe00

Expected:

- leaf_hash_i = 0x3bd00fdcc06781ca996db6dfb070370eaf44b54b2e53e15ba53af9cb5a9adc45

Definition:

- leaf_hash_i = keccak256(abi.encode(TAG_HASH, chain_id, session_id, epoch, uint32(interval_index), uint8(v_i), H(sig_ghost_i), H(sig_shell_i)))

Vector E: Receipt node hash (Merkle-sum internal node) Inputs:

- `TAG = "GITS_LOG_NODE"`
- `TAG_HASH = 0x8ff17f274aafeacb48bcc81cc0419089924a701538706d5eed0f7bac62e98ca5`
- `chain_id = 8453`
- `session_id = 123456789`
- `epoch = 42`
- Left child: `hL = 0x3bd00fdcc06781ca996db6dfb070370eaf44b54b2e53e15ba53af9cb5a9adc45`, `sL = 1`
- Right child is leaf (`interval_index = 18, v_i = 0, sigs empty`):
  - `H(sig_empty) = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470`
  - `leaf_hash_18 = 0xeea8fec70a6cc83f8921b68392f325c83995bde1edb3ba04b94003adbc06b3aa`, `sR = 0`

Expected:

- `node_hash = 0x43b9228b9e0b50ae2cd9318bce993781b6cae6d741785b619af56441008097ff`
- `node_sum = 1`

Definition:

- `node_hash = keccak256(abi.encode(TAG_HASH, hL, hR, uint32(sL), uint32(sR)))`
- `node_sum = sL + sR`

Vector F: Mini receipt root (N_PAD = 4) Inputs:

- `chain_id = 8453`
- `session_id = 123456789`
- `epoch = 42`
- `N_PAD = 4`
- Leaves (i from 0 to 3):
  - `i = 0`: `v_i = 1`, `sig_ghost = 0x0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f2021` `sig_shell = 0x65666768696a6b6c6d6e6f707172737475767778797a7b7c7d7e7f808182838485868788898a8b8c`
  - `i = 1`: `v_i = 0`, `sig_ghost = 0x`, `sig_shell = 0x`
  - `i = 2`: `v_i = 1`, `sig_ghost = 0x222222222222222222222222222222222222222222222222222222222222222222` `sig_shell = 0x33333333333333333333333333333333333333333333333333333333333333333333333333333333`
  - `i = 3`: `v_i = 0`, `sig_ghost = 0x`, `sig_shell = 0x`

Expected:

- `log_root = 0xb7b7c48afe3c285065cf61d09b7eb454e18e51bed622e22ad2ff758a1b0f7c2a`
- `SU_delivered = 2`

Vector G: Ghost ID derivation Inputs:

- `identity_pubkey = abi.encode(uint8(1), abi.encode(address(0x1111111111111111111111111111111111112` (K1 key)
- `wallet = address(0x2222222222222222222222222222222222225678)`
- `salt = bytes32(0x0000000000000000000000000000000000000000000000000000000000000001)`
- `tag_hash = keccak256(bytes("GITS_GHOST_ID"))`

Derivation:

- `ghost_id = keccak256(abi.encode(tag_hash, identity_pubkey, wallet, salt))`

Definition:

- `identity_pubkey` is encoded per the canonical identity key encoding (Section 14.1): `abi.encode(uint8(sig_alg), pk_bytes)` where K1 uses `sig_alg = 1` and `pk_bytes = abi.encode(address)`.

Vector H: Shell ID derivation Inputs:

- `identity_pubkey = abi.encode(uint8(1), abi.encode(address(0x3333333333333333333333333333333333333339AE`
  (K1 key)
- `salt = bytes32(0x0000000000000000000000000000000000000000000000000000000000000002)`
- `tag_hash = keccak256(bytes("GITS_SHELL_ID"))`

Derivation:

- `shell_id = keccak256(abi.encode(tag_hash, identity_pubkey, salt))`

Definition:

- `identity_pubkey` is encoded per the canonical identity key encoding (Section 14.1): `abi.encode(uint8(sig_alg), pk_bytes)` where K1 uses `sig_alg = 1` and `pk_bytes = abi.encode(address)`.
- Note: `payout_address` is NOT an input to `shell_id` derivation. It is stored as mutable state in ShellRegistry.

15. References

[1] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System." 2008. https://bitcoin.org/bitcoin.pdf

[2] W3C. "Web Authentication: An API for accessing Public Key Credentials (WebAuthn) Level 2." https://www.w3.org/TR/webauthn-2/

[3] FIDO Alliance. "Client to Authenticator Protocol (CTAP) 2.1" (specifications index). https://fidoalliance.org/specifications/

[4] AMD. "AMD SEV-SNP Attestation: Establishing Trust in Guests." https://www.amd.com/content/dam/amd/en/documents/developer/lss-snp-attestation.pdf

[5] Intel. "Intel Trust Domain Extensions (TDX) Overview." https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html

[6] Arm confidential compute and attestation resources: Arm Confidential Compute Architecture (CCA) overview and CCA attestation token draft. https://docs.kernel.org/arch/arm64/arm-cca.html and https://www.ietf.org/archive/id/draft-ffm-rats-cca-token-00.html

[7] Optimism OP Stack documentation and specs: sequencer assumptions and components. https://docs.optimism.io/stack/components and https://specs.optimism.io/interop/sequencer.html

[8] Ethereum Improvement Proposals. "EIP-4337: Account Abstraction Using Alt Mempool." https://eips.ethereum.org/EIPS/eip-4337

[9] S. Bradner. "Key words for use in RFCs to Indicate Requirement Levels." RFC 2119. https://www.rfc-editor.org/rfc/rfc2119

[10] R. Barnes, K. Bhargavan, B. Lipp, and C. Wood. "Hybrid Public Key Encryption." RFC 9180. https://www.rfc-editor.org/rfc/rfc9180

[11] Y. Nir and A. Langley. "ChaCha20 and Poly1305 for IETF Protocols." RFC 8439. https://www.rfc-editor.org/rfc/rfc8439

[12] NIST. "Digital Identity Guidelines: Authentication and Lifecycle Management" (SP 800-63B). https://pages.nist.gov/800-63-3/sp800-63b.html

[13] FIDO Alliance. "Passkeys". https://fidoalliance.org/passkeys/

[14] Ethereum Improvement Proposals. "EIP-7951: Precompile for secp256r1 Curve Support." https://eips.ethereum.org/EIPS/eip-7951

[15] OP Stack Specification. "Precompiles (P256VERIFY)." https://specs.optimism.io/protocol/precompiles.html

[16] IETF. RFC 8785: “JSON Canonicalization Scheme (JCS).” https://www.rfc-editor.org/rfc/rfc8785

[17] Golem Network Documentation. “Golem Architecture / Protocol documentation.” https://docs.golem.network/

[18] Akash Network Documentation. “Akash Docs (decentralized cloud marketplace).” https://docs.akash.network/

[19] iExec Documentation. “iExec Docs (including TEE features).” https://docs.iex.ec/

[20] Render Network Documentation. “Render Network Knowledge Base / Docs.” https://know.rendernetwork.com/

[21] Livepeer Documentation. “Livepeer Docs / Primer.” https://docs.livepeer.org/

[22] Phala Network Documentation. “Phala Docs (confidential computing).” https://docs.phala.network/