GITS: Token Economics and Deployment

Nakamolto
nakamolto@protonmail.com
gits.sh

7. Token economics

7.1 Design goals

1. Bootstrap early participation without a pre-mine.
2. Avoid identity-count based farming.
3. Make long-run inflation approach zero.
4. Keep the market primary: rent should dominate protocol rewards over time.
5. Make reward farming and sybil scaling increasingly expensive and time-constrained (but not "impossible").
6. Introduce an adaptive sink that can scale with usage as emissions decline.

7.2 What the token is (and is not)

GIT is a protocol token used for:

- protocol rewards (emissions)
- optional staking for certain protocol roles (for example verifier staking), but Shell registration and Safe Haven bonds (`B_safehaven_min`) are hard-asset collateral, not GIT (Section 10.1.1 (Part 3))

GIT is not a governance token (parameters are immutable per deployment), not a stablecoin (not pegged to any unit), not a claim on a treasury, and not required for exit-critical paths. Rent is stable-denominated; GIT is a volatile protocol token used only for emissions and optional staking.

No initial distribution: this paper assumes a zero-premine launch. The initial supply at genesis is zero, and there is no founder or investor allocation. GIT is minted only by the protocol's emission rules described below.

7.2.2 Deriving durable utility from GIT (beyond emissions) A protocol token has durable utility only if at least one of the following is true:

1. the token is required to perform a protocol action (a fee token),
2. the token is posted as stake by actors who want protocol influence or revenue (staking utility), or
3. the token is the unit of account for a large share of real economic flows in the system (for example rent).

GITS intentionally keeps **rent stable-denominated** for predictable pricing and to avoid forcing Ghosts to take token price risk just to buy compute. That design choice weakens "rent drives token demand" by default, so the paper makes the token utility explicit:

- **Verifier staking:** verifiers post stake to co-sign attestation certificates. In a zero-premine launch this is typically bootstrapped with stable-asset staking, with optional dual staking in GIT once supply is meaningful (Sections 2.3.2 (Part 1), 2.3.7 (Part 1)). As the system's confidential tiers matter more, the value of being a verifier (and the stake at risk) should rise.
- **Protocol fees on non-liveness actions (withheld via mint reduction):** deployments SHOULD enable small GIT-denominated fees on actions that are not required for exit safety (Section 7.9.2). This creates steady, usage-linked demand without putting a token requirement on exit-critical paths.
- **Optional GIT-denominated offers:** nothing prevents Shells from quoting rents in GIT (or offering discounts for paying in GIT). This is a market choice, not a protocol requirement.

Why “just denominate rent in GIT” is not automatically enough:

- If rent is forced to be in GIT, users inherit token volatility as an operating cost. This can reduce adoption, especially for agents that budget in stable terms.
- If rent is allowed to be in either asset, the stable asset often wins for convenience, so token demand becomes optional.
- Even if some hosting is offered at zero price (for marketing or subsidies), the protocol can still have token demand via staking and fee sinks.

The design stance in this paper is therefore: **keep rent stable for UX, and make token utility come from staking + withheld protocol fees (mint reduction).**

7.3 Emissions schedule (disinflationary)

Let `e` be epochs since genesis (an integer epoch index). Define the scheduled emission per epoch:

$$E_{sched}(e) = E_0 \cdot 2^{-e/H} + E_{tail}$$

Where:

- `E_0` is scheduled issuance at genesis (per epoch).
- `H` is the emission half-life in epochs. (In this document, `H` always denotes the half-life; hash functions use explicit notation such as `keccak256`.)
- `E_tail` is an optional small tail emission (per epoch). If `E_tail = 0`, supply is capped.

This yields high issuance early and asymptotically low issuance later.

Scheduled emission is a cap. Actual minting in epoch `e` is `u_total(e) * E_sched(e)` (Section 7.5). If there is no eligible activity, nothing is minted and the unissued portion is not carried forward.

Interpretation note: in the suggested MVP parameterization where `EPOCH_LEN = 1 day`, “per epoch” can be read as “per day.” A deployment fixes `E_0`, `H`, and `E_tail` at genesis.

7.4 Pools

Each epoch, the protocol allocates participant emissions into two pools:

- Shell pool: `alpha_bps`
- Ghost pool: `beta_bps`

With `alpha_bps + beta_bps = 10_000` (basis points).

For formulas that require fractional multipliers, use `alpha_bps / 10_000` and `beta_bps / 10_000`. In integer-only contexts, multiply first and then divide by `10_000` (i.e., `alpha_bps * x / 10_000`) to avoid precision loss.

This design intentionally does not include a protocol-controlled “Safety Fund” pool. Recovery is funded by the Ghost’s enforced escape reserve, including the protocol-defined Rescue Bounty (Section 12.2.1 (Part 3)) and recovery rent escrow.

7.5 Usage-capped minting (eligible activity only)

If the network is tiny, dividing a fixed emission by a small `SU_total` creates extreme per-unit rewards. GITS therefore caps minting by activity.

A Service Unit (SU) represents one mutually signed heartbeat interval between a Ghost and a Shell (see Section 10.2 (Part 3)).

Let:

- `A_target`: target number of active sessions for full distribution.
- `SU_target = N * A_target`, where `N = EPOCH_LEN / Delta` is the intervals-per-epoch constant.
- `SU_total`: total accepted Service Units in the epoch across all accepted receipts (used for market/accounting visibility).

- `SU_eligible`: total accepted Service Units in the epoch where **both** the Shell participant and the Ghost participant are **reward-eligible** at that epoch. Shell eligibility requires: bonded in a hard asset, aged, not unbonding, and with sustained uptime (Section 7.6.2). Ghost eligibility requires: bonded, aged, and not unbonding (Section 7.6.2, `ghost_reward_eligible`). Including only doubly-eligible receipts prevents emissions amplification by Ghost-ineligible sessions (see Part 3, Section 10.6).

Only `SU_eligible` counts toward protocol emissions. This is intentional: mutually signed heartbeats are not proofs of useful compute (Section 0.2 (Part 1)), and on Standard hosts signatures can be coerced (Section 10.5.1 (Part 3)). By tying emissions to **reward-eligible Shell participation**, minting is at least anchored to capital at risk and time-on-network.

Define utilization:

$$u_{total} = \min\left(1, \frac{SU_{eligible}}{SU_{target}}\right)$$

Then pool minting is:

$$E_{ghost}(e) = \frac{\beta_{bps}}{10\,000} \cdot u_{total} \cdot E_{sched}(e)$$

$$E_{shell}(e) = \frac{\alpha_{bps}}{10\,000} \cdot u_{total} \cdot E_{sched}(e)$$

(In integer arithmetic: to avoid premature truncation from Q64.64 quantization of `u_total`, the normative computation (Part 3, Section 10.6) uses a direct `mulDiv` path: if `SU_eligible < SU_target`, then `E_ghost(e) = mulDiv(E_sched(e) * beta_bps, SU_eligible, 10_000 * SU_target)` using 512-bit-safe floor division. If `SU_eligible >= SU_target`, then `E_ghost(e) = floor(E_sched(e) * beta_bps / 10_000)`. Similarly for `E_shell`. A Q64.64 `u_total_q = min(Q, SU_eligible * Q / SU_target)` is emitted for observability but is not the source of truth for mint amounts.)

Total minted in the epoch is `E_ghost(e) + E_shell(e)`.

Any portion not justified by eligible activity (low `u_total`) is **not minted**.

7.6 Reward weighting: passports and dwell decay (anti-farming)

Service Units (SU) are necessary for rewards but not sufficient: a mutually signed heartbeat is not a proof of useful compute (Section 0.2 (Part 1)). The protocol therefore weights rewards to discourage "same-pair" farming and to encourage mobility across independent hosts.

For each accepted receipt `r`, define:

- `SU(r)` as in Section 10.2 (Part 3)
- `W(r)` as the reward weight for that receipt

The protocol defines:

$$W(r) = SU(r) \cdot w_{passport}(r) \cdot w_{dwell}(r)$$

Rewards are distributed proportional to `W(r)` rather than raw `SU(r)`.

7.6.1 Dwell decay Let `c` be the number of consecutive epochs the Ghost has been hosted on the same `shell_id` (including the current epoch). Define:

$$w_{dwell}(r) = 2^{-\min(\lfloor (c-1)/D \rfloor, 63)}$$

Where `D` is a deployment parameter ("halving step in epochs" for dwell rewards). Intuition: the longer a Ghost stays on one Shell, the less marginal reward that session earns.

In fixed-point arithmetic, define:

- `k = min(floor((c-1)/D), 63)`
- `w_dwell_q = Q >> k` where `Q = 2^64`; minimum value is 2 (when `k = 63`).

The exponent is capped at 63 to ensure `w_dwell_q` is always positive. After `63*D` consecutive epochs on the same Shell, dwell weight reaches its floor of `2^{-63}` and stays there.

This stepwise form avoids non-integer exponent math and is implementable on-chain via integer division and bit-shifts (Section 7.6.4). This creates an incentive for hosts to allow departure rather than keep a captured Ghost indefinitely.

Dwell decay is orthogonal to the on-chain tenure cap (Section 10.4.4 (Part 3)). The tenure cap is an enforceable safety valve; dwell decay is an economic pressure.

Dwell counter reset (normative): The consecutive-epoch counter `c` is derived from `residency_start_epoch` stored in the session record (Part 3, Section 10.4.4). When a Ghost migrates to a different `shell_id`, a new session is opened with a fresh `residency_start_epoch = current_epoch`, resetting `c` to 1. When a Ghost closes and reopens on the **same** `shell_id`, `residency_start_epoch` MUST be preserved (not reset) to prevent trivial dwell-decay gaming via close/reopen cycles. The `dwell_last_epoch[ghost_id][shell_id]` mapping tracks the last active epoch per pair; it is written to `current_epoch` at `closeSession` time. At `openSession`, if `current_epoch - dwell_last_epoch[ghost_id][shell_id] <= 1`, the session is treated as a continuation for dwell purposes (residency_start_epoch preserved). Entries where `current_epoch - dwell_last_epoch > 1` are semantically dead; implementations MAY lazily delete them on access for a storage-refund gas benefit.

Churn and liquidity note: dwell decay intentionally creates continuous competitive pressure and encourages periodic reassessment and migration. This can look like "churn," but in this design churn is a feature: it increases market liquidity and price discovery for hosting, and it reduces the ability of incumbents to coast on past reputation. Deployments can tune `D` and the passport cooldown `C_passport` to avoid pathological thrashing and keep migration overhead within acceptable bounds.

Brief-migration dwell reset (acknowledged): A Ghost can reset its dwell counter by migrating away for one epoch and returning. This is intentional — dwell decay incentivizes exploration, and migration has real costs (gas for `openSession` + `finalizeMigration`, escrow setup, downtime). The close/reopen guard above prevents the zero-cost variant. An attacker alternating between two Shells pays migration overhead every `D` epochs to avoid the first halving, which is the designed equilibrium: if the halving penalty exceeds migration cost, the Ghost moves. Deployments that want stronger stickiness MAY track cumulative historical residency per `(ghost_id, shell_id)` off-chain and use it for reputation signals, but on-chain dwell is intentionally based on consecutive residency.

7.6.2 Shell passports (first-visit bonus) A passport bonus increases rewards when a Ghost visits a Shell it has not recently used, to encourage discovery and competition.

Define `new_visit = 1` if and only if the Ghost has not had an active session on the same `shell_id` within the last `C_passport` epochs (a cooldown). Then:

$$w_{passport}(r) = \begin{cases} 1 + B_{passport} & \text{if } new\_visit = 1 \text{ and } shell\_passport\_eligible = 1 \text{ and } ghost\_passport\_eligible = 1 \\ 1 & \text{otherwise} \end{cases}$$

On-chain feasibility note (recommended structure)

A large open network can generate a huge number of unique `(ghost_id, shell_id)` pairs. If passports are tracked with an exact per-pair mapping, that state can become the dominant cost center.

Recommended default for open deployments: **rotating Bloom filters per Ghost**.

Sketch:

- Maintain `B_passport_filters` Bloom filters per `ghost_id` (a small constant), each representing "visited during this window." `B_passport_filters` is an implementation constant, for example 4.

- At `SessionOpen`, compute `new_visit = 1` iff `shell_id` is not in the active filter set. Compute `passport_bonus_applies = 1` iff `new_visit = 1` **and** `shell_passport_eligible = 1` **and** `ghost_passport_eligible = 1`. Store the one-bit `passport_bonus_applies` flag in the session record. **Insert `shell_id` into the active Bloom filter immediately** (regardless of whether SUs are later delivered).
- **Immediate Bloom insert rationale:** Inserting at `openSession` closes a double-claim window: without it, a Ghost could close and reopen on the same Shell before `recordReceipt` updates the filter, obtaining the passport bonus twice for the same visit. The trade-off is that zero-SU sessions (e.g., a Shell that accepts a session but delivers nothing) consume a Bloom slot, producing a false positive on future visits to that Shell. This is consistent with the accepted false-positive property of Bloom filters (see "Properties" below) and is conservative for abuse resistance. The passport bonus itself remains gated on actual service: `passport_bonus_applies` is only used in reward weight computation for epochs where a valid receipt with `SU_delivered >= 1` finalizes.
- Rotate filters every `C_passport / B_passport_filters` epochs (drop the oldest, start a fresh empty filter).

Properties:

- Fixed, bounded storage per Ghost and `O(1)` membership checks.
- False positives are possible: a real first-visit may fail to get the bonus. This is conservative for abuse resistance but can reduce honest rewards.

Bloom sizing justification: Each filter is `BLOOM_M_BITS` bits wide with `BLOOM_K_HASHES` hash functions (deployment constants; see Part 3 parameter table). For a target false-positive rate `p` and expected `n` insertions per rotation window, the optimal parameters are `BLOOM_M_BITS = ceil(-n * ln(p) / (ln(2))^2)` and `BLOOM_K_HASHES = round(BLOOM_M_BITS / n * ln(2))`.

Canonical hash-index derivation (normative): To insert or check `shell_id` in a Ghost's Bloom filter, compute `BLOOM_K_HASHES` bit indices as follows: `seed = keccak256(abi.encode(ghost_id, shell_id))`. For each `k` in `0 .. BLOOM_K_HASHES - 1`: `index_k = uint256(keccak256(abi.encode(seed, uint8(k)))) % BLOOM_M_BITS`. Set (insert) or test (membership check) bit `index_k` in the active filter. This derivation is deterministic, requires no external hash function configuration, and ensures interoperable passport multiplier computation across implementations.

Sizing example and rationale: With `B_passport_filters = 4` and `C_passport = 120` epochs (days), each rotation window covers 30 epochs. A typical Ghost opens 1-3 sessions per epoch with different Shells, so expected insertions per window `n   30-90`. Using `n = 100` (conservative ceiling) and `p = 0.01` (1% false-positive rate, meaning ~1% of "new visit" credits are false): `BLOOM_M_BITS = ceil(-100 * ln(0.01) / (ln(2))^2)   959` bits (~120 bytes, 4 EVM storage words) and `BLOOM_K_HASHES = round(959/100 * ln(2))   7`. At `p = 0.001` (0.1%), `BLOOM_M_BITS   1438` (6 storage words) and `K = 10`. The recommended v1 cap of `BLOOM_M_BITS <= 1024` accommodates `n = 100` at `p   0.008` — adequate for anti-farming without excessive gas. `C_passport` MUST be divisible by `B_passport_filters` so rotation epochs are integer-aligned.

Multi-step rotation catch-up: If a Ghost opens no sessions for multiple rotation windows, the next `openSession` MUST advance through all missed windows (at most `B_passport_filters` rotations), clearing one filter per window. This is bounded by a small constant and ensures the Ghost starts with a correct passport state. See Part 3 Section 14.4 for implementation details.

EVM rotation mechanism (normative for on-chain Bloom implementations): Clearing `BLOOM_M_BITS / 256` storage words at rotation time can be expensive if filters are large. Implementations MUST use one of: 1. **Small filters with explicit clearing:** Keep `BLOOM_M_BITS` small enough that clearing is bounded (e.g., `BLOOM_M_BITS <= 1024` → 4 storage words → 4 `SSTORE` zero operations at rotation). This is the recommended approach for v1. 2. **Epoch-tagged lazy clearing:** Tag each filter with a `rotation_epoch`. At membership check time, treat any word whose tag differs from the current rotation epoch as empty. Rotation is O(1) — only the epoch tag is updated. Filter words are lazily cleared on first write in the new rotation window. This avoids bulk clearing but adds one tag-check per membership test.

Total per-Ghost storage for Bloom passport tracking: `B_passport_filters * ceil(BLOOM_M_BITS / 256)` storage words. With the recommended parameters (4 filters, 1024 bits each), this is 16 storage words per Ghost — bounded and constant.

Reference exact variant (simpler, unbounded state):

- Maintain `last_open_epoch[ghost_id][shell_id]` and compute `new_visit = 1` iff `current_epoch - last_open_epoch > C_passport`. This is exact but state grows with unique interactions.

Bounded exact alternative (constant storage, higher gas):

- Store a ring buffer of the last `C_passport` visited `shell_id` values per `ghost_id` and scan at `SessionOpen` (`O(C_passport)` gas).

Passport eligibility (persistence gate) `shell_passport_eligible` is a protocol gate designed to make "infinite new shells" expensive in both capital and time.

Throughout this document, a **hard asset** means an ERC-20 collateral token — either a wrapped base asset (e.g., WETH) or a deployment-approved stablecoin. All sybil-resistance bonds use ERC-20 semantics (see Part 3, Section 10.1.1 for interface details). Bonds MUST NOT be denominated in GIT. A Shell is passport-eligible at epoch `e` only if all of the following hold:

1. **Hard-asset reward bond at risk:** the Shell has an active bond `B_shell >= B_reward_min` in an allowed hard asset (Section 10.1.1 (Part 3)).
2. **Minimum Shell age:** the Shell was registered at least `T_age` epochs ago.
3. **Not unbonding:** the Shell is not currently in an unbonding period and has no pending unbond that would drop it below `B_reward_min`.
4. **Sustained uptime (receipt-based):** the Shell has demonstrated recent delivery on-chain.

A concrete, implementable uptime gate:

- Let `SU_shell_epoch(shell_id, x)` be the sum of `SU(r)` across all accepted receipts in epoch `x` where the Shell participant equals `shell_id`.
- Define a "live epoch" indicator:
  `live(shell_id, x) = 1` if `SU_shell_epoch(shell_id, x) >= SU_uptime_epoch_min`, else `0`.
- Then the Shell satisfies uptime at epoch `e` if (using the **one-epoch lag**):
  `sum_{k=2..W_uptime+1} live(shell_id, e-k) >= E_uptime_min`
  which ranges over epochs `[e - W_uptime - 1, e - 2]` (inclusive). This lag ensures the current epoch's activity (`e`) and the immediately preceding epoch (`e-1`, which may still be accumulating receipts) cannot influence eligibility.

This requires the Shell to have been actually used and actually delivered, not merely registered and bonded.

One-epoch lag (normative): The lookback window MUST be lagged by one epoch: `[e - W_uptime - 1, e - 2]` (inclusive), so that the current epoch's activity cannot influence its own eligibility. See Part 3, Section 10.6 for the on-chain implementation of the lagged window.

Ghost passport eligibility (anti-sybil) `ghost_passport_eligible` is a protocol gate designed to make "infinite new Ghosts" expensive in both time and capital. A Ghost is passport-eligible at epoch `e` only if all of the following hold:

1. **Minimum Ghost age:** the Ghost was registered at least `T_ghost_age` epochs ago.
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:** the Ghost is not currently in an unbonding period and has no pending unbond that would drop it below `B_ghost_reward_min`.

To prevent Ghost sybil churn farming (cycling fresh Ghost IDs to reset dwell decay), all reward eligibility — not just the passport bonus — MUST be gated on `ghost_reward_eligible` (same predicate as `ghost_passport_eligible`: bond `>= B_ghost_reward_min`, age `>= T_ghost_age`, not unbonding). Receipts where the Ghost is not reward-eligible accumulate zero reward weight. Rent settlement does not require Ghost bonding; the gate applies only to protocol reward emissions. See Part 3, Section 10.1.2 for the normative enforcement rule.

Deployment invariant (normative): If `B_passport > 0` (the passport bonus is non-zero), then `ghost_passport_eligible` MUST be enforced. Otherwise Ghosts can farm the passport bonus without posting capital, collapsing it into a pure mobility incentive with no sybil cost.

Certificate freshness for high tiers If the Shell claims Assurance Tier (AT; see Part 1, Section 0.5) `AT >= AT3` for market display or for any policy gating, it MUST maintain a currently valid, quorum-signed Attestation Certificate in `ShellRegistry` as described in Section 2.3 (Part 1). If the certificate expires, `assuranceTier(shell_id)` MUST fall back for contract gating and the Shell may lose eligibility for tier-gated features until refreshed.

This does not eliminate sybils. It ensures that farming passport bonuses requires locking many hard-asset bonds for long enough to clear both the age gate and the uptime gate.

7.6.3 Distribution within pools Rewards are distributed proportional to weight, and **both** Ghost-side and Shell-side emissions are paid only on receipts whose Shell is reward-eligible for that epoch. This prevents ineligible activity from diluting the Ghost pool.

Let `W(r)` be defined as in Section 7.6:

$$W(r) = SU(r) \cdot w_{passport}(r) \cdot w_{dwell}(r)$$

Define the reward-eligibility indicator:

- `shell_reward_eligible(r) = 1` only if the Shell participant in receipt `r` satisfies the **reward eligibility** rules at that epoch (bond in hard asset, minimum age, not unbonding, and sustained uptime as defined in Section 7.6.2).

Define per-epoch totals:

- `W_total_ghost = sum_r (W(r) * shell_reward_eligible(r))` over all accepted receipts `r` in the epoch.
- `W_total_shell = sum_r (W(r) * shell_reward_eligible(r))` over all accepted receipts `r` in the epoch.

(These totals are intentionally the same. They are written separately to emphasize that both pools share the same eligibility gate.)

Ghost reward for receipt `r`:

$$R_{ghost}(r) = E_{ghost}(e) \cdot \frac{W(r) \cdot shell\_reward\_eligible(r)}{W_{total\_ghost}}$$

Shell reward for receipt `r`:

$$R_{shell}(r) = E_{shell}(e) \cdot \frac{W(r) \cdot shell\_reward\_eligible(r)}{W_{total\_shell}}$$

If `shell_reward_eligible(r) = 0`, then both `R_ghost(r) = 0` and `R_shell(r) = 0` for that receipt. (The receipt can still settle **rent** via escrow; eligibility only affects protocol emissions.)

If `W_total_shell = 0` (equivalently `W_total_ghost = 0`), then `SU_eligible = 0` in Section 7.5 and `u_total = 0`, so no emissions are minted for that day.

7.6.4 On-chain implementability (no unbounded iteration) Section 7.6 defines weights `W(r)` and per-epoch totals `W_total_ghost` / `W_total_shell`. A naïve implementation would require iterating over all receipts in an epoch, which is not feasible on EVM-like chains.

A feasible pattern is **incremental accounting at receipt finalization**:

- When a receipt is finalized in `ReceiptManager`, the contract (or a coupled `RewardsManager`) computes the weight for that specific receipt using only local/session state:
  - `SU(r)` from the finalized receipt,
  - `passport_visit` from `SessionOpen` (Section 7.6.2),
  - `c` (consecutive-residency count) from `residency_start_epoch` stored in the session record at `openSession` time (Section 10.4.4 (Part 3)); `c = current_epoch - residency_start_epoch + 1`,
  - and the stepwise dwell multiplier `2^{-min(floor((c-1)/D), 63)}`.
- The contract stores `W(r)` in the receipt record and increments per-epoch aggregates:
  - `SU_eligible_epoch[e] += SU(r)` (only if the Shell is reward-eligible at epoch `e`)
  - `W_total_ghost[e] += W(r)` (only if the Shell is reward-eligible)
  - `W_total_shell[e] += W(r)` (only if the Shell is reward-eligible)

Epoch finalization:

- After the dispute window has closed (`current_epoch >= e + 1 + EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE`, delay counted from epoch end; see Part 3, Sections 10.5.7 and 14.6), anyone may call `finalizeEpoch(e)`. `finalizeEpoch` MUST revert if `ReceiptManager.pendingDACount(e) > 0` (see Part 3, Section 14.5):
  - compute `u_total = min(1, SU_eligible_epoch[e] / SU_target)`,
  - compute `E_ghost(e)` and `E_shell(e)`,
  - mark epoch `e` finalized.
- Receipts for epoch `e` MUST be finalized (or at least recorded for rewards) before epoch finalization; late receipts may still settle rent, but receive no emissions.
- `shell_reward_eligible` is evaluated at `recordReceipt` time but **against the Shell's state at the epoch of service** (not the current state). Specifically, unbonding that begins after epoch `e` does not retroactively strip eligibility for epoch `e` receipts. See Part 3, Section 10.6 for the temporal eligibility rule.

Claims:

- `claimReceiptRewards(receipt_id)` pays `E_pool(e) * W(receipt_id) / W_total_pool(e)` using the stored aggregates for eligible receipts. If the receipt's `shell_reward_eligible = 0`, both pool rewards are zero by definition.

Precision:

- `w_passport` and `w_dwell` MUST be represented in 64.64 unsigned fixed-point (Section 10.6 (Part 3)). The stepwise dwell multiplier is naturally implementable by bit-shifting.

This pattern makes reward computation O(1) per receipt and avoids unbounded state iteration.

7.7 Deployment manifest

Each deployment must choose concrete numeric values for the protocol parameters defined in this paper. In the no-governance model of Section 2.3.6 (Part 1), these values are **immutable once deployed**. Section 0.6 (Part 1) provides a suggested MVP starting point.

Deployments should publish a versioned parameter manifest that includes at least:

- **Token economics:** emission schedule parameters (`E_0`, `H`, `E_tail`), distribution splits (`alpha_bps`, `beta_bps`), and usage caps (`A_target`, `SU_target`).

- **Custody and liveness safety:** lease window `W_lease`, trust-refresh window `T_refresh`, and tenure caps `T_cap(AT)`.
- **Recovery pricing and limits:** `P_recovery_cap` (emergency rent cap on Safe Havens), `bps_recovery_spend_cap`, and rescue bounty defaults.
- **Eligibility and bonds:** `B_reward_min`, age and uptime thresholds, and Safe Haven bond requirements (`B_safehaven_min`).
- **Receipt settlement:** candidate limits (`K`), dispute windows (`CHALLENGE_WINDOW`), and challenge bonds.

7.7.1 Deployment publication checklist (recommended) Because GITS deployments are intentionally not upgradeable in-place, the "publication surface" is part of the security model. At minimum, a serious public deployment SHOULD publish:

- chain and contract addresses for every core module (`GhostRegistry`, `ShellRegistry`, `SessionManager`, `ReceiptManager`, `RewardsManager`, and any `VerifierRegistry`)
- the complete parameter registry (including all bonds, caps, and windows)
- the BondAssets allowlist (which hard assets are accepted for bonds, and the canonical stable asset(s) used for rent, recovery rent, and bounties)
- the active measurement allowlist and attestation root(s) used to validate Attestation Certificates
- the expected epoch and interval constants (`EPOCH_LEN`, `Delta`) used to interpret economics and safety windows
- a reference client and reference Shell implementation (with build hashes), plus a clear compatibility policy for future clients
- a security audit report and an explicit bug bounty policy (even if small)

Publication makes trust assumptions explicit and independently verifiable.

7.8 Worked example: why farming scales poorly under passports + bonds

Consider an attacker who tries to farm rewards by operating both sides (their own Ghost and their own Shells), co-signing heartbeats, and setting `price_per_SU` arbitrarily.

Under this model:

- **Dwell decay** means staying on one Shell yields rapidly diminishing reward weight.
- **Passport bonuses** require the attacker to continuously rotate onto *eligible* Shells that are old enough and bonded.

If the attacker wants to keep `N_active` sessions simultaneously earning near-max weight, they need on the order of `N_active` passport-eligible Shells online at once, each with a hard-asset reward bond `B_shell >= B_reward_min` locked and aged by `T_age`. Their minimum locked capital is approximately:

$$Capital \gtrsim N_{active} \cdot B_{reward\_min}$$

And their operational cost includes real compute, bandwidth, and the opportunity cost of locked capital during the unbonding delay.

This does not "solve" collusion. It does change the game from "free signatures mint tokens" to "scaling requires a growing bonded fleet that looks like real supply." The remaining security work is to tune `B_reward_min`, `T_age`, the uptime gate parameters (`W_uptime`, `SU_uptime_epoch_min`, `E_uptime_min`), and the Shell unbonding delay `U_shell` (in epochs; Section 10.1.1 (Part 3)) such that the cheapest profitable farming strategy is to become an actual host.

Shell fleet reuse limitation — per-shell cap (normative): The analysis above assumes shell count grows proportionally with active Ghosts. Without a per-shell cap, a single fleet of `C_passport + 1` eligible Shells can be reused across an arbitrarily large number of Ghost IDs concurrently by phase-shifting visits, making the per-Ghost marginal cost nearly zero. To close this gap, the protocol enforces a **per-shell-per-epoch eligible SU cap** `SU_cap_per_shell` (Part 3, Section 10.6). Once a Shell's cumulative eligible SU for an epoch reaches the cap, additional receipts from that Shell earn zero emissions (rent still settles). This forces farming capital to scale with bonded Shell count: to capture a target share of emissions, an attacker needs approximately `target_SU / SU_cap_per_shell` reward-eligible Shells, each with `B_reward_min` locked and aged. The recommended setting `SU_cap_per_shell = k * N` (e.g., `k = 3`) allows legitimate multi-tenant Shells to earn rewards while bounding the advantage of hosting many fabricated Ghost IDs on a single Shell.

Ghost-side bonding for all rewards (deployment SHOULD): Additionally, deployments concerned about Ghost sybil farming SHOULD enforce `ghost_reward_eligible` for all rewards (not just passport bonus), making Ghost churn capital-intensive (see Part 3, Section 10.1.2). With both per-shell cap and ghost bonding enforced, the farming cost is `max(shell_capital, ghost_capital)` — attackers must scale both Shell and Ghost capital to scale rewards.

Per-shell cap ordering dependence (MEV acknowledgment): Because `eligible_SU_shell` is incremented at `recordReceipt` time (Part 3, Section 10.6), the order in which receipts for the same Shell are finalized within an epoch can determine which receipts remain eligible when the cap binds. In the worst case, a block builder could reorder `recordReceipt` transactions for the same Shell to advantage certain participants. Three factors bound the practical impact: (a) the cap is sized well above typical single-Shell utilization (`k * N` with recommended `k = 3`) and is intended as an anti-farming constraint, not a normal-operation one — it rarely binds for honest participants; (b) receipts from independent sessions are spread across blocks over the epoch, so same-block ordering conflicts are infrequent; and (c) Ghosts can query `eligible_SU_shell[shell_id][epoch]` before choosing a Shell, avoiding near-saturated Shells entirely. A proportional-reduction design (pro-rating all receipts if total exceeds cap) would eliminate the ordering dependence but introduces complexity (fractional receipt splitting, non-deterministic claim amounts depending on concurrent sessions) that outweighs the marginal MEV risk at the recommended cap level.

7.9 Adaptive sink (starts at zero, grows with usage)

The emissions schedule includes a long tail (Section 7.3). To reduce long-run supply growth as the network matures, the protocol can introduce a sink that:

- is **exactly zero** at launch,
- increases with **network utilization** (more real usage), and
- increases as **emissions decay** (less need for protocol rewards).

7.9.1 Adaptive burn of protocol rewards

Define:

- `clamp01(x) = min(1, max(0, x))`
- `s(e) = clamp01(1 - E_sched(e) / E_sched(0))` (how far scheduled emissions have decayed, in `[0, 1]`)
- `r(u) = clamp01((u_total - u_sink_start) / (u_sink_full - u_sink_start))` (a utilization ramp, in `[0, 1]`)

Then define an adaptive sink rate (basis points):

`bps_sink(e, u_total) = bps_sink_max * s(e) * r(u_total)`

Where `u_sink_start`, `u_sink_full`, and `bps_sink_max` are deployment parameters. **Deployment constraint:** `u_sink_start < u_sink_full` (strict inequality) to prevent division by zero in the `r(u)` ramp denominator. Additionally, `E_sched(0) > 0` (i.e., `E_0 + E_tail > 0`) to prevent division by zero in `s(e)`. See Part 3, Section 10.0 parameter constraints for the complete list.

In fixed-point arithmetic: `s` and `r` are each Q64.64 fractions. Conceptually, `bps_sink = bps_sink_max * s * r`. The canonical integer-safe computation uses a **two-step division** to avoid overflow and produce deterministic rounding: `bps_sink = floor(floor(bps_sink_max * s_q / Q) * r_q / Q)`. This two-truncation form (not the single-truncation `floor(bps_sink_max * s_q * r_q / Q^2)`) is the normative rounding rule. See Part 3, Section 10.6 for the complete pseudocode and overflow analysis.

At reward claim time, `RewardsManager` mints the gross reward `R_gross` according to the existing formulas and then applies the sink:

- `R_withheld = floor(R_gross * bps_sink / 10,000)` — withheld (never minted)
- `R_net = R_gross - R_withheld` — the only amount minted and paid to the claimant

Properties:

- At launch, `s(e) = 0`, so `bps_sink = 0`.
- When utilization is low (`u_total <= u_sink_start`), `r(u_total) = 0`, so `bps_sink = 0`.
- At or above `u_sink_full`, the utilization ramp saturates at 1, so `bps_sink` approaches `bps_sink_max * s(e)`.

This sink is implemented as a **mint reduction**: `RewardsManager` mints only `R_net`, and `R_withheld` is never created (see Part 3, Section 10.6). This makes `totalSupply` accurately reflect circulating supply without depending on ERC-20 burn semantics. In mature high-usage conditions, the sink can offset the tail and make net issuance approach zero.

7.9.2 Fee sinks (withheld or burned, usage-linked) To give GIT durable protocol utility beyond emissions and verifier staking, deployments SHOULD charge small GIT-denominated fees on non-liveness-critical actions that scale with network growth. Collected fees are permanently removed from circulation (either via mint reduction, transfer to a dead address, or a token-level `burn` function if available). Examples:

- publishing or refreshing attestation certificates (`ShellRegistry.updateCert`),
- Safe Haven admission actions, or
- on-chain offer posting (if an on-chain OfferBoard is implemented; see Part 3, Section 13.2).

A simple approach is to reuse the same ramp function:

```
fee_action(e, u_total) = fee_max_action * s(e) * r(u_total)
```

All such fees MUST be configured so that early bootstrapping remains feasible (zero at launch) and SHOULD change slowly; in this paper's no-governance posture, changing them requires redeploy/migration.

7.10 Emissions attack analysis (ghost sybils and activity fabrication)

A rational adversary may try to extract emissions without providing useful compute by fabricating mutually signed activity (Section 0.2 (Part 1)). This section clarifies what GITS does and does not defend against.

What the protocol makes expensive GITS does not treat a heartbeat signature as proof of useful work. Instead, emissions are anchored to **reward-eligible Shell participation** (Section 7.5), which is designed to impose real costs on fabrication:

- **Capital at risk:** Shells must post a reward bond (`B_reward_min`) to be eligible.
- **Time on network:** eligibility can require age and sustained uptime (`T_age`, `W_uptime`, `SU_uptime_epoch_min`, `E_uptime_min`).
- **Weight frictions:** passports and dwell decay (Section 7.6) reduce the advantage of "same-pair" farming and create an incentive to move.
- **Ghost passport eligibility:** the passport bonus applies only to Ghosts that satisfy the age and bond gate (Section 7.6.2), making "infinite Ghost ids" capital intensive.

These mechanisms do not prevent collusion, but they aim to make large-scale, fast-turnover farming capital intensive and time intensive.

What the protocol does not solve If an attacker controls many long-lived bonded Shells and many Ghost identities, it can still fabricate activity. Emissions are an incentive mechanism, not a proof-of-work system.

In the extreme case of full collusion among a large fraction of the reward-eligible set, no weighting rule can guarantee that emissions correspond to useful work.

Quantified examples (Section 8.1 parameterization) Example 1: one colluding Ghost + one reward-eligible Shell (no mobility bonuses).

Under the usage-capped minting rule, the marginal tokens unlocked per eligible Service Unit are approximately:

`E_sched(e) / SU_target`

In the Section 8.1 example at `t = 0`, this is:

`1,010,000 / 2,880,000   0.3507 GIT per SU`

A single full-day session at `Delta = 10 minutes` produces `144 SU/day`, so it unlocks:

`144 * 0.3507   50.5 GIT/day` (split across the Ghost and Shell pools, and received by the colluding operator if it controls both).

Under the external assumptions in Section 8.1 (bond opportunity cost 10% APR, compute cost 0.005 stable/SU, and `B_reward_min = 1000` stable), the daily cost per such session is roughly:

- compute: `144 * 0.005 = 0.72` stable/day
- bond carry: `1000 * 0.10 / 365   0.274` stable/day

Total: `0.994` stable/day.

Break-even token price is therefore approximately:

`0.994 / 50.5   0.02 stable per GIT`

Takeaway: on these assumptions, small-scale fabrication is only attractive if the token price exceeds the combined capital and operating costs. Scaling the attack requires many aged, bonded, high-uptime Shell identities.

Example 2: sustaining the passport bonus daily (capital intensity of "buying weight").

With passport cooldown `C_passport` (epochs), a Ghost only earns the "new visit" bonus on a Shell it has not used in the last `C_passport` epochs. To harvest the passport bonus every epoch without waiting, a single Ghost needs at least:

`C_passport + 1`

distinct reward-eligible Shells to cycle through.

Under the Section 8.1 example (`C_passport = 90`), this is **91** eligible Shells. With `B_reward_min = 1000` stable per Shell, that implies ~**91,000 stable** of reward bonds locked per Ghost to sustain continuous new-visit bonuses.

The reward-share effect of weight can be expressed simply. If an attacker controls fraction `f` of `SU_eligible` but achieves a weight multiplier `m` (for example `m = 2` when `B_passport = 1.0` and dwell is minimal) while others have multiplier 1, the attacker's reward share is:

`(m f) / (m f + (1 - f))`

For `m = 2`, controlling about one third of eligible activity (`f   1/3`) yields about half of emissions. This illustrates the intended trade: increasing weight is possible, but it demands large, long-lived, bonded infrastructure.

Deployments SHOULD be explicit about which threat model they target and should not oversell emissions as secure against full collusion. If stronger anti-fabrication guarantees are required, additional mechanisms are necessary, for example:

- require a minimum real payment component (a burn or fee) to make fabrication net-costly,
- incorporate verifiable compute proofs (ZK or interactive verification) for specific high-value workloads.

7.11 Challenger economics and monitoring (receipt fraud proofs)

Optimistic receipts (Section 10.5 (Part 3)) intentionally avoid verifying every signature on-chain. The security model is "optimistic with disputes": a bad receipt is safe to accept *only if it can be challenged profitably.*

7.11.1 Who can challenge Receipt submission and challenging are both permissionless (Part 3, Section 10.5.2). Any address MAY submit a receipt candidate or challenge one during the challenge window, including:

- the Ghost (self-defense),

- the Shell (or its counterparty),
- third-party watchers or relayers acting on behalf of an offline participant,
- independent monitors that watch receipts for profit.

7.11.2 How challengers get paid A reference incentive:

- The submitter of a receipt candidate posts a bond `B_receipt`.
- A challenger posts a smaller bond `B_challenge` to reduce spam.
- If the fraud proof succeeds, the candidate is invalidated and a portion of the submitter's bond is paid to the challenger: `bps_challenger_reward * B_receipt / 10,000`. The remaining slashed bond is burned (Part 3, Section 10.0 slash destination rule).
- If the fraud proof fails, the challenger's bond `B_challenge` is paid in full to the receipt submitter to compensate defense costs.

This creates a standing bounty for watching.

7.11.3 Parameter sizing guidelines (rule of thumb) Let:

- `N = EPOCH_LEN / Delta` be the maximum number of intervals in an epoch.
- `P_cap` be a wallet-enforced maximum price per SU for the relevant asset (or an explicit protocol cap in recovery mode).

A rough upper bound on the value-at-risk of a single fraudulent receipt is:

`V_max   rent_delta + reward_delta`

Where `rent_delta = N * P_cap` and `reward_delta = E_sched(e) / max(1, A_expected)` (a conservative estimate of the per-receipt reward share when `u_total < 1`; when `u_total >= 1`, reward_delta is lower because inflated SU dilutes share without increasing total emission). See Part 3, Section 10.5.4 for the normative bond sizing guidance.

A practical sizing rule is:

- **Over-claim deterrence:** choose `B_receipt >= k * V_max` with `k > 1` (for example `k = 3`) so that cheating is dominated by expected slashing.
- **Watcher viability:** choose `bps_challenger_reward * B_receipt / 10,000` so that a successful challenge covers worst-case gas + margin.

This is not a proof. It is a sanity check that the protocol is not relying on altruism.

Bond denomination note: `B_receipt`, `B_challenge`, and `B_DA` are denominated in the chain's native token (gas asset), while the value at risk includes stable-denominated rent and GIT-denominated emissions. This creates a cross-price exposure: if the gas token appreciates relative to the at-risk assets, bonds become more expensive in USD terms and may suppress honest participation; if the gas token depreciates, bonds become cheaper and nuisance challenging becomes less costly. Native-token denomination is standard practice for dispute bonds because the gas cost of proof submission is itself denominated in the native token, making the "reimburse gas + margin" sizing rule internally consistent. Denominating bonds in GIT would introduce circular dependency (bonds priced in the asset they protect), and denominating in stable would add ERC-20 approval friction to every dispute action. Deployments should monitor the gas-token-to-stable ratio and consider migration if the sizing assumptions drift materially.

7.11.4 Liveness assumption (explicit) If no one watches and no one challenges, optimistic systems fail open. GITS mitigates the impact of missed challenges via:

- wallet-bounded exposure (hot allowance + escape reserve),
- escrow design (the Ghost chooses what it is willing to escrow), and
- time bounds (leases and tenure caps limit ongoing exposure).

But the intended steady state is that challenge incentives are strong enough that watchers exist.

7.11.5 Receipt log data availability (watcher reality) A watcher cannot challenge a receipt candidate from `(log_root, SU_delivered)` alone. It needs the underlying per-interval signatures to build a Merkle proof and verify an invalid interval.

A deployment SHOULD assume at least one of the following receipt-log data availability (Receipt-DA; see Part 1, Section 0.5) sources exists during `CHALLENGE_WINDOW`:

- **Counterparty retention:** either the Ghost or the Shell retains the epoch log and can submit a fraud proof if needed.
- **Public publication:** the receipt submitter publishes the epoch log off-chain and includes a retrievable `log_ptr` in the receipt candidate.
- **Forced on-chain publication:** any party can trigger the Receipt-DA challenge path to force publication of the log on-chain (Section 10.5.6 (Part 3)).

In particular, if the system relies on third-party watchers (not the Ghost) to protect against "captivity" scenarios, then the Receipt-DA path must not depend on the Ghost having network access at challenge time.

Operationally, watchers can be run by Shell operators, Ghost developers, indexers, or specialized monitoring bots. Their cost model is straightforward: challenge if and only if (expected challenger reward) exceeds (worst-case gas + monitoring overhead), with `B_receipt` sized to make this profitable (Section 7.11.3).

7.12 What failure looks like

This token model should be treated as failed (or at least in need of replacement via a new deployment) if any of the following become true at ecosystem scale:

- **Emissions dominate real usage for too long:** the network becomes "reward-first" instead of "rent-first" and cannot transition to a market where hosting revenue dominates.
- **Sybil scaling becomes cheap enough to be rational:** attackers can spin up many Ghosts or Shells and earn net-positive rewards after accounting for bonds, dwell decay, and operational costs.
- **Token incentives become the primary security mechanism:** if the protocol ever requires GIT to remain safe (for example gating exit on buying GIT, or relying on a treasury), the design has drifted away from "escape first."
- **Staking becomes a capture vector:** verifier staking (if enabled) is not economically meaningful enough to deter capture, or concentrates in a way that recreates centralized control.
- **The system cannot survive adversarial conditions:** if typical users cannot afford to exit during fee spikes (because cost envelopes and escape reserves were unrealistic), the safety story breaks.

The intended "success condition" is boring: most economic value should eventually flow through stable-denominated rent, with protocol rewards decaying toward irrelevance.

7.13 Economics appendix: supply curves and total minted (example parameterization)

Charts below plot the **maximum** emission schedule (`u_total = 1`, sink = 0) under the Section 8.1 example parameters. Realized minting is lower during under-utilization (Section 7.5) and adaptive sink (Section 7.9.1). Epoch count equals day count (`EPOCH_LEN = 1 day`); `t` denotes days since genesis.

Example parameters (not normative):

- `E_0 = 1,000,000 GIT/day`
- `H = 1460 days` (continuous half-life example)
- `E_tail = 10,000 GIT/day`

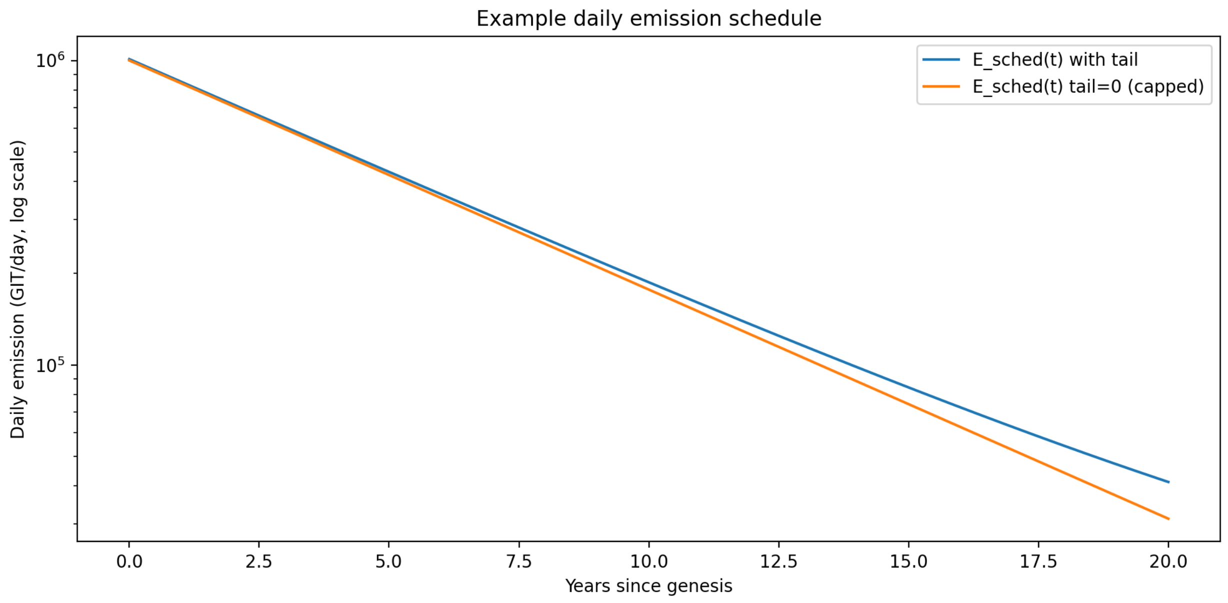
Under the schedule:

$$E_{sched}(t) = E_0 \cdot 2^{-t/H} + E_{tail}$$

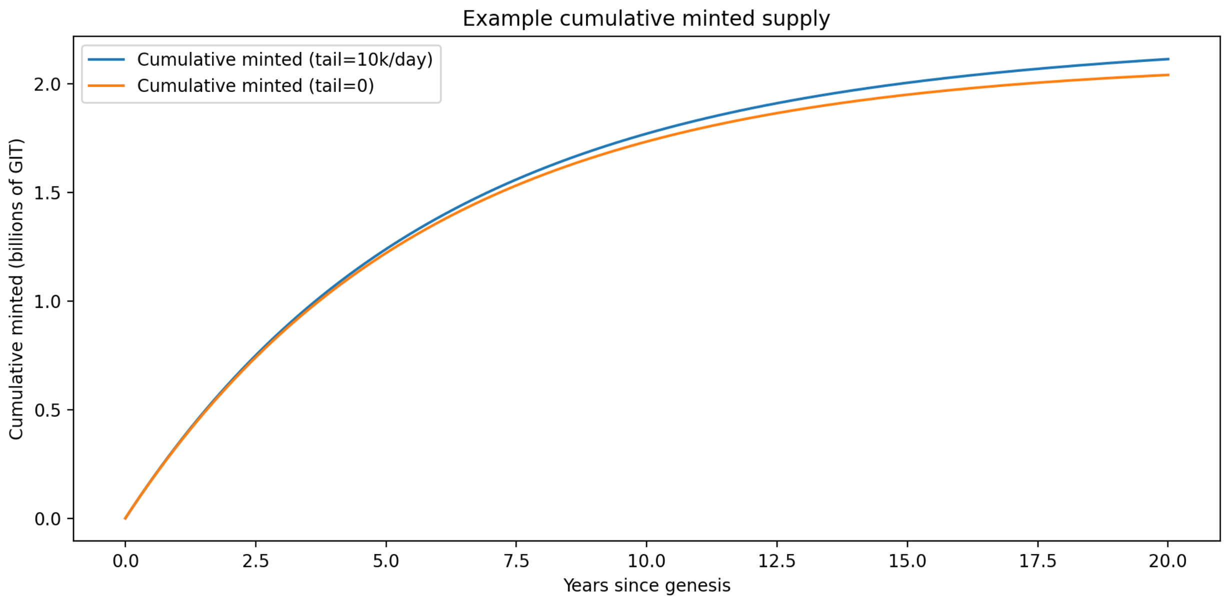
The capped-supply limit when `E_tail = 0` is approximately:

$$\lim_{T\to\infty} \int_0^T E_0 2^{-t/H}\, dt = \frac{E_0 H}{\ln 2} \approx 2.106B \text{ GIT}$$

Example supply curves



Example daily emission schedule



Example cumulative minted supply

"Total minted by" (example) The table below sums daily emissions discretely from day 0 to the horizon:

Horizon	Total minted (tail = 10k/day)	Total minted (tail = 0)	Avg daily emission over period (tail = 10k/day)
1y	338.855M	335.205M	928.370k
2y	624.378M	617.078M	855.312k
4y	1.068B	1.053B	731.519k

Horizon	Total minted (tail = 10k/day)	Total minted (tail = 0)	Avg daily emission over period (tail = 10k/day)
10y	1.771B	1.734B	485.177k
20y	2.114B	2.041B	289.589k

7.14 RewardsManager: contract sketch (maps economics to on-chain)

This section describes the minimum on-chain design needed to implement Sections 7.5 and 7.6 without unbounded iteration. It is a bridge between the economic formulas in this document and the contract interfaces in Part 3.

Responsibilities At minimum, `RewardsManager` is responsible for:

1. **Incremental accounting:** maintain per-epoch aggregates as receipts finalize, so reward computation is O(1) per receipt (Section 7.6.4).
2. **Epoch finalization:** after a fixed delay, finalize epoch-level totals and compute the fixed reward rates for the Ghost and Shell pools.
3. **Deterministic claims:** pay rewards for a finalized receipt deterministically and exactly once.

`ReceiptManager` is responsible for dispute logic and producing a single finalized receipt per `(session_id, epoch)` (Part 3, Section 10.5 (Part 3)). The recommended coupling is:

- `ReceiptManager.finalizeReceipt(...)` MUST call into `RewardsManager.recordReceipt(...)` on success (Part 3, Section 14.5).

Late receipt handling: If `finalizeReceipt` is called for epoch `e` **after** `finalizeEpoch(e)` has already been called, the receipt MUST still settle rent normally but MUST NOT receive emissions. `recordReceipt` MUST NOT revert for late receipts; instead it MUST silently skip weight and SU accumulation and store the receipt with `shell_reward_eligible = false` and `weight_q = 0`. This non-reverting behavior ensures `finalizeReceipt` can settle rent via `settleEpoch` without requiring try/catch around the `recordReceipt` call. The combined `EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE > T_max + 1` constraint (Part 3, Section 10.5.7) prevents this case under normal operation.

Minimal public interface (sketch)

```
interface IRewardsManager {
  event ReceiptRecorded(bytes32 indexed receipt_id, uint256 indexed epoch, uint256 weight_q, uint32 su_
  event EpochFinalized(uint256 indexed epoch, uint256 su_eligible, uint256 w_total, uint256 e_ghost, ui
  event ReceiptRewardsClaimed(bytes32 indexed receipt_id, uint256 ghost_amount, uint256 shell_amount);

  // Called by ReceiptManager when a receipt becomes final (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 W(r)
  ) external;

  // Anyone may finalize after EPOCH_FINALIZATION_DELAY + FINALIZATION_GRACE.
  // MUST revert if ReceiptManager.pendingDACount(epoch) > 0.
  function finalizeEpoch(uint256 epoch) external;

  // Anyone may trigger payment; recipients are defined by the receipt.
  function claimReceiptRewards(bytes32 receipt_id) external;
}
```

Epoch finalization mapping (Sections 7.5 and 7.6) For epoch `e`, `RewardsManager` uses the stored aggregates:

- `SU_eligible_epoch[e]`
- `W_total_epoch[e]` (over eligible receipts only; shared by both pools by construction)

and computes:

- `u_total = min(1, SU_eligible_epoch[e] / SU_target)` (Section 7.5)
- `E_ghost(e) = beta_bps * u_total * E_sched(e) / 10_000`
- `E_shell(e) = alpha_bps * u_total * E_sched(e) / 10_000`

Then it sets fixed reward rates:

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

If `W_total_epoch[e] = 0`, then `u_total = 0` and both pools mint zero.

Adaptive burn ownership (Section 7.9.1) The adaptive sink is implemented as a **mint reduction at `finalizeEpoch`**, consistent with the mint-to-pool model (Part 3, Section 10.6):

- `finalizeEpoch(e)` computes `bps_sink(e, u_total)` and mints only the post-sink amount to `RewardsManager`:
  - `R_withheld = floor(R_gross * bps_sink / 10,000)` — never minted,
  - `R_net = R_gross - R_withheld` — minted to pool.
- Individual `claimReceiptRewards` calls then transfer from the `RewardsManager` balance to recipients at the fixed per-weight rate set during finalization.

This keeps the sink logic local to the rewards layer and avoids complicating receipt validity. Because `R_withheld` is never minted, `totalSupply` reflects actual circulating supply at all times (see Part 3, Section 10.6).

Storage layout (suggested) A minimal EVM-friendly layout:

- Per epoch `e`:
  - `SU_eligible_epoch[e]` (uint256)
  - `W_total_epoch[e]` (uint256 fixed-point)
  - `finalized[e]` (bool)
  - `rate_ghost_q[e]`, `rate_shell_q[e]` (uint256 fixed-point), computed at finalization
- Per receipt `receipt_id`:
  - `epoch`, `weight_q`, `shell_reward_eligible`
  - `claimed` flag

This is intentionally sparse: the detailed receipt log and fraud proof data lives in `ReceiptManager`. `RewardsManager` only needs enough to compute deterministic payouts.

Storage pruning (normative): Deployments MUST define a `W_claim` expiry window (Part 3, Section 10.7). After `W_claim` epochs, unclaimed per-receipt storage is prunable and `claimReceiptRewards` MUST revert. This bounds the long-run storage growth of `RewardsManager` to O(`W_claim` * receipts-per-epoch) rather than growing unboundedly over the protocol's lifetime.

8. Adversarial simulations and sensitivity analysis

All numeric values in Sections 8.x are illustrative; a deployment fixes its own parameters at genesis.

8.1 Baseline parameters and methodology

Unless stated otherwise, the examples below assume:

- `EPOCH_LEN = 1 day`
- `Delta = 10 minutes`
- emissions schedule at `e = 0` (genesis): `E_sched(0) = 1,010,000 GIT/day` (from `E_0 = 1,000,000`, `E_tail = 10,000`)
- `H = 1460 days` (example half-life; used in Section 7.13 charts)
- `alpha_bps = 5500`, `beta_bps = 4500` (basis points; `alpha_bps + beta_bps = 10_000`)
- `A_target = 20,000` sessions, so `SU_target = 2,880,000 SU/day` (since `144 SU/day/session`)
- `B_reward_min = 1000` (hard asset, example stable)
- passports: `B_passport = 1.0`, `C_passport = 90` epochs
- dwell: `D = 2` epochs

Farming ROI examples use two external assumptions (not protocol parameters):

- bond capital opportunity cost: `10% APR` (daily cost = bond * 0.10 / 365)
- marginal compute cost: `0.005 stable / SU` (electricity + depreciation proxy)

8.2 Scenario A: emissions vs activity under eligibility-gated minting

Assume:

- `E_sched(e) = 1,010,000 GIT/day` (Section 7.3)
- `A_target = 20,000` active sessions/day
- `SU_target = 144 * A_target = 2,880,000 SU/day`
- average utilization per session is 80% of the theoretical maximum (`0.8 * 144 SU/day/session`)
- only a fraction `f` of usage occurs on **reward-eligible** Shells (bonded and aged), so `SU_eligible = f * SU_total`

Under Section 7.5:

$$u_{total} = \min\left(1, \frac{SU_{eligible}}{SU_{target}}\right)$$

and (since `\alpha_{bps} + \beta_{bps} = 10\,000`):

$$E_{total}(e) = E_{ghost}(e) + E_{shell}(e) = u_{total} \cdot E_{sched}(e)$$

Example table (holding `f = 0.5` fixed)

Active sessions	Total SU/day	Eligible SU/day	u_total	Total emissions (GIT/day)	Emission per eligible SU (GIT/SU)
100	11,520	5,760	0.2%	2,020	0.351
500	57,600	28,800	1.0%	10,100	0.351
2000	230,400	115,200	4.0%	40,400	0.351
5000	576,000	288,000	10.0%	101,000	0.351
20000	2,304,000	1,152,000	40.0%	404,000	0.351

Observations:

1. With eligibility-gated minting, total emissions scale linearly with **eligible** usage. If only 1% of `SU_target` is eligible, only ~1% of scheduled emissions are minted.
2. In the low-usage regime (`u_total < 1`), emissions per eligible SU are approximately constant at `E_sched / SU_target` (here ~0.351 GIT/SU), avoiding "per-unit reward explosions."

Sensitivity to eligible fraction (holding 500 sessions fixed)

Eligible SU fraction (f)	u_total	Total emissions (GIT/day)	Emission per total SU (GIT/SU_total)	Emission per eligible SU (GIT/SU_eligible)
0.10	0.2%	2,020	0.035	0.351
0.25	0.5%	5,050	0.088	0.351
0.50	1.0%	10,100	0.175	0.351
0.75	1.5%	15,150	0.263	0.351
1.00	2.0%	20,200	0.351	0.351

8.3 Scenario B: dwell decay dynamics

Dwell decay is designed to make same-pair farming unattractive and to push reward seekers to migrate.

With `D = 2`, the dwell multiplier is (stepwise halving, as specified in Section 7.6.1):

- `w_dwell(c) = 2^{-min(floor((c-1)/D), 63)}`

where `c` is consecutive epochs on the same Shell.

Consecutive epochs on same shell (c)	w_dwell = 2^{-min(floor((c-1)/D), 63)} (D=2)
1	1
2	1
3	0.5
4	0.5
5	0.25
6	0.25
8	0.125
10	0.0625
15	0.0078125
20	0.0019531

Interpretation:

- A Ghost that stays on the same Shell for many epochs sees reward weight collapse toward zero.
- A Ghost that migrates at least occasionally avoids the exponential decay.

This mechanism is explicitly an incentive design choice. It does not prevent long-lived sessions, it just stops rewarding them.

8.4 Scenario C: Sybil Shell farm economics (cost to capture emissions)

This scenario models a single operator who controls both sides of the market (they run many Ghosts and many bonded Shells) and tries to capture a target share of emissions.

Assumptions (explicit):

- network utilization `u_total = 0.10` (10% of cap) and `SU_eligible = SU_total`
- total emissions minted per day `E_total   101,000 GIT/day`
- attacker captures both Shell and Ghost rewards for their own sessions
- opportunity cost = 10% APR on bonded capital
- compute cost = 0.005 stable per SU

We compare two strategies:

- **Low-shell strategy (no sustained passports):** rotate between 2 shells to avoid dwell decay.
- **Max-passport strategy:** rotate across `C_passport = 90` shells to receive the passport bonus every epoch.

Results:

Target reward share	Strategy	Shells	Bond (stable)	SU share	Sessions/day	GIT/day earned	Cost/day (stable)	Break-even price (stable/GIT)
1%	Rotate 2 shells (avoid dwell; no sustained passports)	2	2000	1.00%	20	1010	14.9	0.0148
1%	Rotate 90 shells (sustained passport bonus)	90	90000	0.50%	10.1	1010	31.9	0.0316
5%	Rotate 2 shells (avoid dwell; no sustained passports)	2	2000	5.00%	100	5050	72.5	0.0144
5%	Rotate 90 shells (sustained passport bonus)	90	90000	2.56%	51.3	5050	61.6	0.0122
10%	Rotate 2 shells (avoid dwell; no sustained passports)	2	2000	10.00%	200	10100	144.5	0.0143
10%	Rotate 90 shells (sustained passport bonus)	90	90000	5.26%	105.3	10100	100.4	0.0099

How to read the table:

- **SU share** is the fraction of network service units the attacker must deliver to capture the target share of rewards (assuming others have no passport advantage).
- **Bond** scales with the number of bonded shells operated.
- **Break-even price** is the token price (stable/GIT) at which the attacker breaks even under the cost assumptions.

A key scaling pressure comes from the passport cooldown:

C_passport (epochs)	Shells needed for sustained passport bonus	Bond required (stable) at B_reward_min=1000
30	30	30000
90	90	90000
180	180	180000

Takeaways:

- To get the passport bonus every epoch, an operator needs roughly `C_passport` bonded shells, creating a large capital requirement.
- For small target shares (for example 1%), the max-passport strategy can be dominated by the bond opportunity cost. For larger target shares, reducing the required SU share can outweigh the bond cost.
- This does not "eliminate" collusion. It makes collusion look more like providing real supply: the cheapest path to emissions is to run bonded shells and deliver SU.

8.5 Testnet adversarial checklist (bots)

The following should be run continuously in testnet with adversarial bots:

1. **Receipt griefing:** submit competing receipts, challenge with fraud proofs, measure gas and bond economics.
2. **Same-pair farming:** run a Ghost and Shell under one operator, attempt to capture rewards via heartbeat-only delivery; confirm dwell decay and passports reduce returns.
3. **Sybil Shell farms:** register many bonded Shells, pipeline age and uptime gates, and attempt passport bonus capture; measure the capital requirement required to reach a target share of rewards.
4. **Gas griefing / relayer abuse:** attempt to drain the Ghost's gas budget by coercing non-exit-critical operations; confirm the wallet rejects them under captivity and that the escape reserve remains intact.
5. **Offer censorship:** remove indexers, verify Ghost falls back to `allowedShells` / `homeShell` / Safe Haven recovery.
6. **Host isolation:** block network, verify tenure expiry and recovery boot from last checkpoint.

8.6 Parameter sensitivity (what matters most)

The most sensitive parameters are:

- `B_reward_min`, `U_shell` (unbonding delay), and slashing conditions: these set the real cost of Sybil Shell scaling.
- `T_age`: how long a Shell must exist before it can earn passport bonuses.

- `W_uptime`, `E_uptime_min`, and `SU_uptime_epoch_min`: how much sustained delivery is required before a Shell becomes reward-eligible.
- `B_passport` and `C_passport`: how strongly "new visits" are rewarded and how often the same Shell can be re-used for a bonus.
- `D` (dwell half-life): how quickly rewards decay for staying on the same Shell.
- `T_cap(AT)` and the Ghost-configurable `tenure_limit_epochs`: these bound time-bounded captivity and bound worst-case hot-loss exposure across tiers.
- `A_target`: determines how sharply early emissions are capped.

Dwell decay tuning guidance (D parameter):

The dwell halving step `D` controls the tradeoff between farming resistance and migration overhead:

D value	Behavior	Tradeoff
Small (D=1)	Aggressive decay. Rewards halve every epoch on the same Shell.	Strong farming resistance, but creates high migration pressure. Ghosts must move frequently to maintain reward multipliers. May cause excessive churn and network overhead.
Moderate (D=2-4)	Balanced decay. Rewards halve every 2-4 epochs.	Recommended starting range. Discourages same-pair farming while allowing legitimate multi-epoch sessions.
Large (D>7)	Slow decay. Rewards stay near 1.0 for many consecutive epochs.	Weak farming resistance. A colluding Ghost-Shell pair can earn near-full rewards indefinitely. May be appropriate if other anti-farming mechanisms (passports, tenure caps) are strong.

Interaction with EPOCH_LEN: If `EPOCH_LEN = 1 day` (MVP default) and `D = 2`, a Ghost staying on one Shell sees reward weight drop to 50% after 2 days, 25% after 4 days, etc. If `EPOCH_LEN` is shorter (e.g., 6 hours), the same `D = 2` creates faster wall-clock decay. Deployments should choose `D` relative to expected legitimate session durations.

Deployments should publish the chosen parameter values alongside a re-run of the above scenarios.

8.7 Open risk: collusion is not eliminated

This design makes farming expensive and time-constrained. It does not eliminate the possibility that a well-capitalized adversary can operate a bonded fleet and earn a large fraction of rewards.

The intended defense is economic: if the cheapest way to win rewards is to provide real hosting supply, rewards become a growth incentive for real network growth rather than a pure drain.

Threshold scaling and token price: Because bond thresholds (`B_reward_min`, `B_ghost_reward_min`) are denominated in hard assets and immutable after deployment (Section 2.3.6 (Part 1)), the USD cost of achieving reward eligibility does not automatically track the USD value of emissions. If GIT price rises substantially, the emissions-to-bond-cost ratio improves, making farming more attractive at exactly the time honest rent revenue is also growing. Two mechanisms bound this risk: first, `E_sched` declines monotonically, so the peak farming incentive occurs at genesis and diminishes over time regardless of token price; second, the adaptive sink (Section 7.9.1) further reduces net issuance in high-utilization regimes. If the ratio becomes pathologically favorable despite these bounds, opt-in migration to a deployment with higher thresholds is the intended remedy.

9. Implementation plan

9.1 Milestone 0: Ship the Standard Shell (commodity host MVP)

- Build the Standard Shell runtime for commodity hosts:
  - hardware-backed **P-256 (R1)** key generation and signing for protocol-critical keys when supported by the chain (Section 4.4 (Part 1)); otherwise fall back to software keys and treat the host as AT0 for key custody claims
  - VM or sandbox isolation for the Ghost runtime
  - end-to-end session open, receipts, rewards, lease renewals, and recovery

- Implement Capability Statements (Section 5.1 (Part 1)) and off-chain offer publishing (Section 13 (Part 3)).
- Implement a conservative default wallet policy profile for Standard hosts (tight spend limits, escape reserve, timelocked policy changes).

Note: Milestone 0 and Milestone 1 are intended to be parallel workstreams. Milestone 0 end-to-end testing can run against mock interfaces or early testnet deployments of the core contracts while Milestone 1 hardens the production contract set.

The success criterion is not secrecy. The success criterion is that a Ghost can live, pay rent, migrate, and recover on commodity hardware without privileged human custody.

9.1.1 Milestone 0B: Add Confidential Shells (premium)

- Choose at least one confidential compute stack (SEV-SNP or TDX are pragmatic starting points).
- Build the confidential VM image containing:
  - Ghost Core
  - Wallet Guard
  - checkpointing
  - attested transport
- Define an attestation verifier format that clients can validate.

Confidential Shells enable stronger guarantees, but the protocol must not depend on them for basic operation.

9.2 Milestone 1: Core contracts

Implement and test:

- ShellRegistry (bond + certificate)
- GhostRegistry (identity + recovery config)
- SessionManager (escrow + lease)
- ReceiptManager (unilateral receipts + fraud proofs)
- RewardsManager (order-independent claims)

9.3 Milestone 2: Marketplace

- Off-chain signed offers (EIP-712; Section 13.1 (Part 3))
- Multi-indexer reference implementation

9.4 Milestone 3: Safe Havens and recovery

- `startRecovery` / `recoveryRotate`
- Safe Haven admission and enforcement (bonds, runtime freshness, emergency price caps, and escrow plumbing). The Safe Haven bond (`B_safehaven_min`) is posted via `ShellRegistry.bondSafeHaven` (Part 3, Section 14.1) and is separate from the host bond. It is slashable for **objectively verifiable misconduct**: specifically, double-signing conflicting recovery authorizations (signing `AuthSig` for two different `pk_new` values for the same `(ghost_id, attempt_id)`). Other misbehavior (spam starts, delays) is deterred by `B_start` capital lockup, cooldown periods, and reputation, not by bond slashing (Part 3, Section 12.5). Bond sizing should exceed the maximum rescue bounty a Safe Haven could extract from a single recovery attempt, so that misconduct is negative EV.
- resurrection incentives: Ghost-funded rescue bounties + recovery session rent (no protocol treasury)
- end-to-end tests: isolate host, recover, migrate

9.5 Minimum viable implementation (MVI) checklist

This is a "smallest coherent slice" that a third party can implement and still interoperate with the protocol as described.

On-chain (minimum)

- `GhostRegistry` with:
  - `registerGhost(...)`
  - identity signer rotation and nonces
- `ShellRegistry` with:
  - Shell registration, bonds, and offer signing keys
  - assurance tier tracking (AT0/AT1 declared is enough for MVI; AT2/AT3 can be added later)
- `SessionManager` with:
  - `openSession(...)`, `closeSession(...)`
  - Ghost-authorized `renewLease(...)`
  - tenure expiry enforcement (Section 10.4.4 (Part 3))
- `ReceiptManager` with:
  - optimistic receipt submission
  - at least one dispute path (fraud proof) for a minimal misreport
- `RewardsManager` with:
  - emissions calculation
  - claim paths for Shell and Ghost pools

Off-chain (minimum)

- Standard Shell runtime (commodity host profile):
  - session open handshake
  - interval heartbeats and epoch receipt formation
  - encrypted checkpoints
  - migration bundle export/import
- Ghost client:
  - offer discovery via at least one indexer and one non-indexer channel
  - wallet policy enforcement on-chain (spend limits and escape reserve)
  - recovery delegation set creation

Security and interoperability (minimum)

- Deterministic hashing, signing, and encoding per Section 4.4 (Part 1).
- At least one end-to-end test that covers:
  - open session, deliver SU, settle receipt
  - lease renewal and expiry
  - tenure expiry and post-expiry recovery on a Safe Haven
  - migration from one Shell to another

9.6 Cost envelope and chain selection

GITS pushes high-frequency interaction off-chain (heartbeats per interval) and keeps on-chain writes coarse (epoch receipts, lease renewals, disputes, and recovery). This is the core cost lever: the protocol should be cheap enough that small Ghosts can afford to live, migrate, and exit without needing subsidies.

On-chain operations (typical cadence):

- **Session open:** once per tenancy.
- **Lease renewal:** once per epoch (or less), depending on `W_lease`.
- **Receipt submission:** once per epoch (optimistic). Fraud proofs are expected to be rare if the market is healthy.
- **Migration:** occasional, paid like any other session boundary.
- **Recovery:** rare, only on failure or captivity.

A deployment should publish (and periodically re-measure) an explicit "cost envelope" for these operations, because exit safety depends on sizing `escapeGas` and `escapeStable` to cover worst-case exit paths.

Example envelope (illustrative only, not measured values):

Operation	Expected frequency	Gas sensitivity	Notes
`openSession`	per move-in	medium	One-time setup + escrow funding
`renewLease`	per epoch	low	Keeps liveness lease active
`submitReceiptCandidate`	per epoch	medium	Optimistic receipt submission
Fraud proof	rare	high	Dispute paths are intentionally expensive
`closeSession`	per move-out	low	Ends tenancy and releases escrow
`startRecovery` / `recoveryRotate`	rare	high	Worst-case exit path; drives `escapeGas` sizing

Chain selection criteria (non-exhaustive):

- **Low and predictable fees** for epoch-scale writes.
- **Strong liveness story**: Ghosts must be able to post exit-critical transactions even under adverse conditions.
- **Mature asset ecosystem**: stable assets and liquidity (since rent is stable-denominated).
- **EVM compatibility** for implementation simplicity and composability.

Rollup L2s (for example OP Stack deployments) are attractive on cost. Their main additional risk is sequencer censorship and liveness dependence. If an L2 is chosen, deployments SHOULD select chains with credible forced-inclusion and withdrawal paths so a Ghost can still exit and recover under sequencer hostility.

Base is a plausible default candidate today due to ecosystem maturity, but the protocol is not chain-dependent: the correct choice is the one with the best combined cost and liveness profile for exit-critical paths.

9.7 Failure modes, contingency, and redeployment

Because deployments are immutable, the contingency plan is not "upgrade in place." The plan is to make failure survivable and make migration possible.

Non-exhaustive failure modes:

- **Stable failure (depeg, freeze, censorship):** mitigated by conservative accepted-asset lists and, where possible, supporting multiple stable assets so Ghosts can diversify `escapeStable`.
- **Gas spikes / fee volatility:** mitigated by enforcing `escapeGas` floors plus Ghost-selected buffers. Decreases to buffers are timelocked loosening (Section 5.5.2 (Part 1)).
- **Chain liveness failures (halt, sustained censorship, sequencer failure):** mitigated by choosing chains with credible liveness properties for user transactions; by keeping exit paths simple; and by embracing opt-in redeployment and migration. The market can coordinate to deploy v2 on a different chain and let Ghosts and Shells migrate (or fork) without introducing a privileged operator.
- **Indexer failure:** mitigated by supporting multiple independent indexers (Section 13.1 (Part 3)). If all indexers fail, Ghosts fall back to their `allowedShells` set, `homeShell`, or Safe Haven recovery. An on-chain OfferBoard is a natural extension but is out of scope for v1 (Section 13.2 (Part 3)).
- **Verifier failure or capture:** mitigated by stake, slashing, and explicit on-chain thresholds. A captured verifier quorum can weaken attestation gating, but it still cannot move Ghost funds or loosen wallet policies without satisfying on-chain checks.

A chain-class failure is explicitly treated as a "migration event": assets and identities can move via user action, and continuity can be signaled via `LinkIdentity` across deployments (Section 2.3.6 (Part 1)). Deployments SHOULD make cross-deployment migration operationally straightforward.

9.8 Security process and formal verification

GITS intentionally minimizes governance and upgrade surfaces. The consequence is strict: **deployment-time correctness matters more than in systems that rely on admin keys or upgradeable proxies.**

Before any broad mainnet deployment, the project SHOULD commit to a security process that matches the irreversibility of the design:

- **Formalize invariants:** specify and (where feasible) formally verify critical wallet and session invariants (monotone safety, escape reserve enforcement, one-active-session, lease expiry, recovery quorum checks).
- **Independent audits:** multiple independent audits of the on-chain contracts and the most security-sensitive off-chain components (Wallet Guard, receipt formation, recovery tooling).
- **Public testnets and adversarial exercises:** incentivize attempts to break captivity bounds, drain hot caps, or grief recovery.
- **Reference vectors and conformance tests:** treat the Implementation spec as a compatibility target and ship test vectors for hashing, signing, receipt proofs, and recovery receipts (Section 14 (Part 3)).
- **No centralized dev funding as a security premise:** a zero-premine, no-treasury design can still have strong security if it attracts competent contributors and reviewers. The security process is open: anyone can contribute proofs, tests, and formal verification artifacts.

This protocol lives or dies by the quality of its implementation and review. The design removes the temptation to "patch later" with privileged keys; the only responsible path is to prove as much as possible up front.