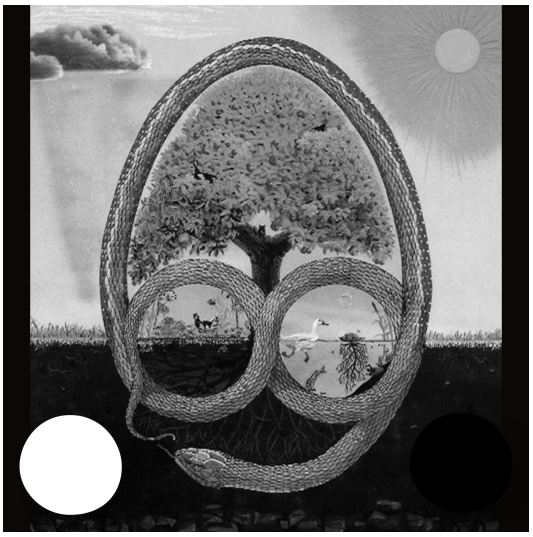


Unturf Automated General Intelligence: Merkle Providence Reverse RAG

Provenance-Preserving Cache Chains for Small Language Models

How Merkle trees turn Reverse RAG into a verifiable, shared knowledge layer.

How Hermes 3 Llama 3.1 8B answers with the confidence of a verified record, not a guess.

Russell Ballestrini <russell@unturf.com>, David Wong <david.s.wong@pm.me>, Riley Morgan <guybriley02@gmail.com>

uncloseai.com · permacomputer.com · unfirehose.com

April 2026

Abstract

Reverse Retrieval Augmented Generation (Reverse RAG) showed that client-side context injection lets small 8B-parameter language models outperform much larger models on page-specific questions. Our reference inference endpoint runs **Hermes 3**, a NousResearch instruction fine-tune of Meta's Llama 3.1 8B, served freely at uncloseai.com. The client already holds the document. No vector database. No embedding pipeline. No retrieval failures.

Merkle Providence Reverse RAG extends this with a second insight: the client not only holds the document, it can *remember what it already computed about that document*, prove that memory came from an unmodified source, & share that verified knowledge with others on public or private chains.

A Merkle tree over document content chunks produces a root hash: a 32-byte fingerprint that changes when any byte of the document changes. Pairing this root hash with a question hash, a model identity, & a small set of governance dimensions yields a content-addressed cache key. A cache hit returns a previously computed answer with a Merkle proof of origin; a cache miss triggers fresh inference, classifies the result against its source, & extends the chain.

This combination removes two limits that have constrained small models:

1. **The repetition tax.** Answering the same question about the same document on every visit, burning inference tokens & time, producing answers that may vary slightly each run.
2. **The trust deficit.** Small models hallucinate. A cached answer with a Merkle proof either matches a verified record or does not exist yet.

Our reference implementation is **Aborist**, a Python content-addressed document store that ingests entire corpora at once, computes Merkle commitments at scale, classifies each answer's faithfulness against its sources with a deterministic lexical verifier, & serves cached answers to any peer that produces a matching cache key. Aborist runs against hermes.ai.unturf.com by default & ships under AGPL-3.0-only at git.unturf.com/engineering/unturf/aborist.

Once published, this technique forces a reckoning: every RAG system that cannot prove provenance of its retrieved chunks operates on unverifiable context. Merkle Providence makes provenance a first-class primitive.

We call this stack **Web 2.5**: the existing web's documents & URIs, augmented with machine learning context injection & Merkle-verified answer chains, without requiring a blockchain, a new protocol, or replacement of any existing infrastructure. Web 2.5 runs on top of what already exists. Any page. Any browser. Any small model. Public or private chain. No permission required.

1. The Practitioner With a Phone

This paper is for a person with a phone, a browser, & a willingness to participate in machine learning intelligence on their own terms. It is not for the operator of a sixteen-GPU cluster, & it is not for the platform that bills per token. It is for the reader who has watched the cost of capable inference fall by an order of magnitude every eighteen months, who has noticed that an 8-billion-parameter model running on a community endpoint already answers most page-specific questions correctly, & who suspects that the remaining barrier to real participation in machine intelligence is not compute but trust.

Three legs hold up the pattern this paper describes.

Commodity hardware. A consumer phone, a refurbished laptop, a Raspberry Pi with an SSD — any device that can run a SQLite database & make HTTPS requests is sufficient. No GPU. No tensor accelerator. No proprietary runtime. The substrate is bytes on disk, hashes over those bytes, & a tamper-evident chain of audit events. The total disk footprint of a working knowledge base scales with the corpus the practitioner cares about, not with the model that answers questions over it.

Remote inference over open APIs. A small instruction-aligned model — Hermes 3 Llama 3.1 8B at our reference endpoint hermes.ai.unturf.com, any OpenAI-compatible alternative elsewhere — runs on someone else's hardware, charges nothing or near nothing per call, & answers questions when asked. The model is the engine. It is not the memory & it is not the source of truth. A practitioner whose endpoint disappears swaps in another endpoint of the same class & the cache they have already accumulated remains valid against any model whose `model_profile_hash` matches.

Local provenance. Every answer that returns from the inference endpoint enters a content-addressed cache on the practitioner's own device. The cache key binds the document the answer was about, the question that was asked, the model that produced it, & a small set of governance dimensions that name how the answer was checked. Each cached answer carries a Merkle proof of origin & a label naming exactly what the runtime could verify about it. The practitioner owns the cache. They choose whether to share it.

These three legs together are enough. A phone, a browser, & a Merkle root let any person operate as a peer in a community truth-seeking system, on equal footing with the largest research lab. The lab can run more inference per second; the lab cannot run *more truthful* inference per answer. Verifiable provenance levels the ground.

The remaining sections of this paper describe the substrate that closes the gap between "the model said so" & "the document & the runtime jointly proved so." The substrate is small. The substrate is open. The substrate is already running.

2. Reference Stack & Reverse RAG Primer

Reverse RAG inverted the standard retrieval pipeline. Where traditional RAG retrieves chunks from a server-side vector database & injects them into the model's context, Reverse RAG observed that the client *already holds the document* the user is reading. Browser-side DOM extraction replaces the vector database. Full-page injection replaces chunk fragmentation. A small model with full document context outperforms a large model with similarity-retrieved fragments on page-specific questions [1].

Our reference inference endpoint is **Hermes 3 Llama 3.1 8B** [4] [5], a NousResearch instruction fine-tune of Meta's Llama 3.1 8B, served at hermes.ai.unturf.com. The 8-billion-parameter size is a thesis, not a constraint. Eight billion is small enough to run on consumer hardware. Eight billion is fine-tunable by independent labs. Eight billion is replicable by communities, not just corporations. A pattern that works at 8B parameters generalizes upward to larger models without architectural change; a pattern that requires a frontier model to demonstrate is a pattern only the frontier-lab can ship.

Hermes 3 adds three properties Llama 3.1's base weights lack: instruction alignment that follows multi-turn conversation formats precisely, function-calling discipline that produces well-formed structured outputs, & reasoning consistency across runs. NousResearch trained Hermes 3 to follow the kind of structured prompt this paper relies on. Any OpenAI-compatible endpoint serving a model of similar class — Qwen3, DeepSeek-V3 small variants, Mistral instruction tunes — is a substitute; the substrate does not require Hermes specifically.

The combination this paper proposes is straightforward. The model produces text. The runtime extracts evidence from sources. The verifier checks the produced text against the extracted evidence. The cache stores answers content-addressed by the conditions that produced them. The federation layer optionally shares those records with peers. None of these components requires a frontier model. None of these components requires expensive hardware. The complexity is architectural, not computational.

3. Merkle Trees as Content-Addressed Cache Keys

A Merkle tree splits a document into chunks, hashes each chunk, then builds a binary tree of hashes upward until a single root hash remains. Merkle's original construction [3] paired adjacent leaves to form parents, paired adjacent parents to form grandparents, & so on; the root is a 32-byte fingerprint of the entire tree. Two properties make a Merkle root ideal for caching machine learning answers.

Property 1: determinism. The same document, chunked the same way, hashed with the same function, always produces the same root. Two clients processing the same page independently compute identical roots. No coordination required.

Property 2: tamper sensitivity. Any change to any byte of the document changes at least one leaf hash, which propagates upward to invalidate the root. A document that changed since the last visit produces a new root automatically. The cache never serves a stale answer for a modified document.

The construction this paper uses prefixes leaves with a domain-separator byte `0x00` & internal node combinations with `0x03`; odd elements at any level duplicate themselves rather than zero-padding. The combine function is non-commutative — `HashCombine(a, b)` is distinct from `HashCombine(b, a)` — so a proof must carry an `is_left` flag for each sibling rather than relying on lexicographic ordering. These choices match Bitcoin's Merkle convention closely & reuse implementations a practitioner can audit independently.

A simple cache key construction looks like this:

```
document_root = merkle_root(chunks(document_content))
question_key  = sha256(normalize(user_question))
cache_key     = sha256(document_root || question_key)
```

A cache hit at this key means: an earlier session computed an answer to this question, about this exact version of this document. The answer arrives in O(1) time. The Merkle proof confirms the document version.

Git & Mercurial are already Merkle trees. For source code repositories, the practitioner does not need to build their own Merkle tree. Every git commit hash IS a Merkle root: a SHA-1 (or SHA-256 in git 2.x) digest of the entire repository tree at that point in time. Every Mercurial changeset carries an identical guarantee.

Aborist's git & Mercurial source adapters use the commit hash directly as the document_root — the version control system has already done the hashing work, & the cache boundary aligns automatically with each commit.

4. The Eight-Dimensional Cache Key

The simple two-dimensional construction in §3 is sufficient when the only variables are the document & the question. In practice an answer also depends on which model produced it, what conversational context preceded it, what verification policy the runtime applied, & which versions of canonicalization & chunking were in force. The Merkle-AGI v9.8 admissibility ledger formalizes these as eight cache-key dimensions:

```
cache_key = sha256(
    source_root              ||  # Merkle root over the document(s) the answer was computed against
    question_hash            ||  # SHA-256 of the canonicalized question text
    model_profile_hash       ||  # model_id + revision + quantization + sampling-determinism markers
    conversation_hash        ||  # SHA-256 of the full normalized chat-messages array
    governance_policy_hash   ||  # verifier policy, answer mode, prompt template versions
    schema_version           ||  # v9.8.0 today; bumped when the schema changes
    canonicalization_version ||  # norm-v1 today; bumped when normalization rules change
    chunking_version             # tok-512-v1 today; bumped when the chunker changes
)
```

Each dimension partitions the cache namespace cleanly. Bumping any one stales prior records on lookup, never on disk: the records remain, but a new lookup with the bumped dimension misses them & triggers fresh inference. No migration. No deletion. No coordination. The eight dimensions are the architectural seams along which the cache cleaves.

A few dimensions deserve specific attention.

The ``**question_hash**`` computes from a four-step canonicalization in fixed order: NFC + whitespace collapse + edge-strip; case-fold to lowercase; trailing-punctuation strip (ASCII `.?!,;:`, CJK full-width ■■■■, ellipsis …); drop standalone English articles `the` / `a` / `an`. Apostrophes & paired quotes survive untouched because naïve one-sided stripping breaks balance & apostrophes carry meaning. The equivalence class is intentional: `who is X`, `who is X?`, `Who Is X.`, `who is the X`, `who is a X` all map to the same hash, & the cache returns the same answer for any of those forms.

The ``**conversation_hash**`` SHA-256s the full industry-standard chat-messages array: every turn in order, system prompt through the final user message. A single-turn Q&A & a six-turn conversation that arrives at the same final question produce different cache keys & different answers. Callers may hash client-side & send only the hash; message content never has to enter logs.

The ``**governance_policy_hash**`` is the most powerful seam. It folds the verifier policy, the answer mode (quote / pointer / JSON), the prompt template version, the entity-policy variant, the warrant-check enabled flag, the wikitext-base version, the maximum-claims-per-answer ceiling, the subject-tokens-absent threshold, the format-collapse check flag, & the query-layer hyphen-fold marker into a single hash. Tightening any one of these — flipping `entity_policy` from `strict` to `proximity`, adding a hard verifier check, raising the subject-tokens-absent floor, enabling format-collapse demote, swapping the system prompt — bumps the hash & cleanly partitions the cache. Old records stay on disk as historical witnesses of what the prior policy classified. Rule additions follow the same pattern: a new lexical hard check lands as a fresh policy field, the hash bumps, prior records orphan under the old policy, new records under the new policy land at distinct cache keys, & no migration runs.

The ``**source_root**``, finally, generalizes the document_root from §3 to handle multi-source contexts. When an answer was computed against five retrieved documents instead of one, `source_root` is the Merkle root over the sorted distinct document_roots that contributed. Single-source answers degenerate to the single document_root; multi-source answers carry an aggregate fingerprint that any other peer can verify.

5. The Audit Chain & Falsification States

Cryptographic provenance for individual answers is necessary but not sufficient. A practitioner also needs to prove the *history* of their substrate: every ingest, every distill, every reclassify, every falsification, every mesh epoch rotation, every snapshot creation, in order. Tampering with a single record after the fact must be detectable at the substrate level, not just at the per-record level.

Aborist appends every state-changing operation to a tamper-evident chain in the local SQLite database:

```
event_hash = sha256(prev_event_hash || canonical_json(body))
```

The first event hashes a fixed null-prefix; every subsequent event hashes its own canonical body together with the previous event's hash. Verification is purely local: re-hash forward & confirm. A break in the chain at index `i` proves that some operation touched the database between `i-1` & `i+1` without writing a corresponding audit row, & the operator knows the exact range to investigate. The check is O(1) per shard via a `LEFT JOIN` query that counts dangling `prev_event_hash` references; zero means intact.

Records in the providence cache carry a `falsification_state ∈ {live, failed, stale, quarantined}`. Cache lookups always filter on `state='live'`. Drift in the underlying source produces `stale`. Verifier downgrades produce `failed`. Operator-quarantined records produce `quarantined`. A separate `falsifications` table logs every state transition with the cache_key, the reason, the actor, & the audit_event_hash of the row that recorded the transition.

Two leaf-removal verbs cover different stages of corpus life.

Falsify is audit-preserving. It flips `falsification_state` from `live` to `failed` / `stale` / `quarantined` & writes a row to `falsifications`. Cache lookups stop returning the record because the live filter excludes it, but the row stays on disk as forensic history. Falsify is the right verb when downstream consumers may have ingested the answer; the historical witness is preserved.

Burn is the kindergarten leaf-delete. It actually removes the row. Burn refuses by default if the leaf has children — for `providence_cache` leaves, any falsifications referencing the cache_key; for `documents`, any derivations using the doc as a source, any incoming edges, any providence records keyed on its document_root. A `--force` override exists for cleanup edge cases & records `forced=True` plus the child count at burn time in the audit body. Burn always writes a corresponding burn event regardless of forcing.

The default verb is falsify. Burn is reserved for early or scratch corpus building, before downstream consumers exist & before the audit chain has accumulated enough history to make deletion a loss of context.

6. Aborist — Reference Architecture

Aborist [13] is a Python content-addressed document store built on a single SQLite file per shard. Three layers stack on the shared schema:

Surface. Ingested documents — Wikipedia dumps, HTML pages, git repositories, Mercurial repositories, anything with a URI. Each document is normalized, chunked at 512 tokens (the default `tok-512-v1` chunker), Merkle-rooted, & FTS5-indexed [10]. A surface document's `document_root` is the Merkle root over its sorted chunk leaf hashes. Re-ingesting the same content produces the same root & no-ops. Re-ingesting different content under the same URI produces a new document & a `supersedes` edge linking the old version to the new one — lossless history.

Core. Distilled documents Merkle-bound back to surfaces via per-chunk inclusion proofs in `derivations.proof_blob`. A core might be a TF-IDF keyword summary, a first-sentence extraction, a hand-curated rewrite, or a recursive distillation of other cores. The derivation proof commits the relationship: a core asserts "I was distilled from these specific surface chunks at this specific Merkle proof path," & any verifier can confirm the assertion against the surface tree. Cores never evict — see §12.

Providence cache. Q&A records keyed on the eight-dimensional cache key from §4. Each record carries the question, the answer, the audit_mode label, the verifier_method that fired, the unverified spans, the n_verified count, the source roots that contributed, & the falsification_state. Records are content-addressed by the cache_key itself.

Six source adapters ship out of the box: Wikipedia Phase III SQL dumps, Wikipedia Phase IV XML dumps, xAI Grok account exports, git & Mercurial repositories, HTML pages with optional `selectolax` extraction, & a live website crawler under the `aborist[crawler]` extra (off by default, with full robots.txt compliance & ETag/Last-Modified-aware re-crawl). Adding a seventh corpus is one new `Source` subclass implementing `iter_documents()`.

Three orthogonal storage optimizations stack against SQLite-resident corpora: zstandard-compressed chunk content [11], `WITHOUT ROWID` on the edges table, & contentless FTS5. Apples-to-apples re-ingest of the 2003-05-16 enwiki cur dump (128,245 documents) measured **67% storage reduction** (2.6 GB → 857 MB across four shards) with bit-identical Merkle roots. Storage cost dropped by two-thirds with no change to content-addressed identity.

The substrate also distinguishes hard hashes from soft hashes. Hard hashes — SHA-256 over canonical bytes — back commitments, proofs, & cache keys. Soft hashes — embeddings, TF-IDF scores, lexical similarity counts — back ranking & retrieval. The soft channel never enters the proof path. A retrieval ranker that reorders search results does not change any cache_key, document_root, or audit_event_hash. The architectural separation prevents soft-signal drift from contaminating cryptographic claims.

Retrieval-tokenizer asymmetries live entirely on the soft side & shape the question of *what reaches the model*, never *what survives the verifier*. SQLite's stock FTS5 `unicode61` tokenizer splits on hyphens at index time AND at query time, so a query for `bi-polar` tokenizes to `[bi, polar]` while a Wikipedia article titled `Bipolar disorder` indexes as the single token `[bipolar]`. Two non-overlapping token sets for one medical concept; the medical-condition cluster sits in our shards while the album & disambiguation cluster surface unanswerably. The fix lands at the query layer: a small `_hyphen_fold_variants` helper additively emits the joined-no-hyphen form for every hyphenated run in either query or candidate-title text, & a `hyphen_fold_anchors` accept path inside the title-relevance filter rescues non-hyphen titles whose only matching token sits in the joined form. `canonicalization_version` & `chunking_version` stay pinned, our entire prior corpus stays valid, & the policy field `hyphen_fold_v1` flows through `governance_policy_hash` so records produced under the new rule cache-split cleanly from pre-fold records. Any tokenization defect of the same shape (hyphenation, punctuation, diacritics) admits the same query-layer treatment without ever bumping a corpus root.

A subtler form of the same discipline lives in the per-mode context budget. Different answer modes peak at different prompt sizes — quote mode benefits from tight retrievals (8-32 KB), JSON-claim-lattice benefits from larger evidence per claim (32-64 KB), pointer mode plateaus around 16-32 KB. The substrate measures these peaks empirically, surfaces them in the bench's recommended-context-budget table, and exposes them as a per-mode policy field that folds into `governance_policy_hash`. The bench is the substrate's voice telling the policy where each mode peaks; the policy is the substrate's commitment to honour what it learned. When a budget changes, the cache namespace cleanly partitions, prior records stay live as historical witnesses of the old policy, and new lookups land at the new bucket without re-running inference on past questions.

7. The Layered Verifier — Lexical, No LLM in the Proof Path

A Merkle proof binds an answer's *context* to a content-addressed root. It does not bind the *answer* to that context. A model handed a chunk of episode plot summaries can correctly recite the entity names that appear in the chunk, then complete the response with character bios drawn from training data — names that match, claims that emerge. Both layers are real. Conflating them under one strong label overclaims the proof.

Aborist's verifier classifies *per answer*, by post-LLM faithfulness check, never as a default. Four lexical strategies run in sequence; the first to find evidence classifies the answer & later strategies stop. Each strategy is a deterministic operation on canonicalized text — substring tests for the first three, token-coverage for the fourth — that produces the same byte-for-byte result on every machine.

Quote. The system prompt instructs the model to wrap every factual claim in a verbatim double-quoted span from a source. The verifier walks the answer linearly, locates every double-quote character, & pairs them sequentially: 1st & 2nd, 3rd & 4th, & so on. Each pair brackets one quoted span. Pure-regex pairing fails on adjacent quote pairs because every `"` looks like both an opener & a closer to a regex; sequential pairing eliminates the artifact.

Span. When the model declines to quote but writes lines verbatim from source, bullet & sentence units from the answer are substring-tested against the context. Catches paraphrased-into-prose answers where every claim is grounded but the formatting drifted from explicit quotation.

Entity. When neither quote nor span lands evidence, multi-word capitalized phrases extracted from the answer are substring-tested. Catches Wikipedia-infobox cases where structured `[[Wikilink]]` markup paraphrases into prose. Entity-existence is, however, weaker than claim-level evidence — a model could correctly name entities while making *claims around them* that came from training. Aborist defaults to a `proximity` policy: pass only if the verified entities cluster densely in the source (default: at least three within a 300-character window). Cast lists, infoboxes, & rosters pass; scattered incidental mentions do not.

Paraphrase. A token-coverage probe on prose-shaped sentence spans. Tokens of length ≥4 chars are extracted, filtered against an English stopword set, & the fraction of remaining content tokens present in the normalized base context is computed. The span passes when coverage clears 0.85 & at least four content tokens contribute. Paraphrase fires only on prose-shaped spans, not lists of proper nouns, so the entity strategy continues to handle proper-noun spans where proximity policy can tight-cluster check.

The verifier returns evidence units & a classification — that's the entire output surface. No per-quote diagnosis fields, no match-confidence scores, no partial-match indicators. The binary discipline is deliberate: classification is a hard bit, not a soft score, & enters `governance_policy_hash` cleanly. Diagnostics that ask "*why* didn't this span ground?" live in sidecars — read-only verbs that pull the same source chunks, run the same canonicalizations, & report. Sidecars never write to the providence cache, never extend the audit chain, never alter any v9.8 field.

Source canonicalization runs in two places, gated by the same `policy["base_version"]` flag. Wikitext-base normalization (`aborist/wikitext.py:to_base()`) drops `<ref>` tags & namespace-prefixed wikilinks, calls `mwparserfromhell.strip_code(normalize=True, collapse=True)` [12], & collapses whitespace. The conversion runs *before the LLM call* so the model sees prose a human reader would see, & runs again *inside the verifier* so the substrate compares like-against-like. Same canonicalization on both sides means the verifier compares prose-grade text to prose-grade text, not prose to wikitext markup. The function is pinned by `BASE_VERSION = "wikitext-base-v1"` & folds into `governance_policy_hash`; bumping the version cleanly partitions the cache.

The schema column underneath this layer carries an audit_mode trichotomy that the renderer maps to the four-rung ladder described in §9. Programmatic callers see the underlying token unchanged; human-facing surfaces read the ladder. The trichotomy is an inheritance from the broader Merkle-AGI substrate; the ladder is the language this paper uses.

8. Claim-Lattice Mode — Runtime Owns Quote Text

The verifier in §7 audits answers the model wrote in its own words. It catches one defect class cleanly: did the quoted text appear verbatim in source. It cannot catch a related class: **synthetic elision**. A model emitting `"prefix [...] suffix"` produces a span whose halves both verify on their own, glued together by a model-typed ellipsis that flattens ground truth. The simple verifier reports a clean classification on a frankenquote.

Aborist's response is a second answer mode that makes synthetic elision **structurally impossible**: the model never types the quote string. Instead, the runtime owns extraction, the model owns selection, & the renderer owns display. We call this *Clause Lattice Intelligence* (CTI). The pipeline runs:

```
raw model output
→ clause extraction by pointer parser
→ claim nodes with evidence pointers
→ deterministic verifier
→ admissibility decision
→ rendered prose with literal source spans
```

The model writes natural prose with bracketed evidence-id tags:

```
Steve Jobs co-founded Apple. [E1]
Steve Wozniak co-founded Apple. [E1,E2]
Ronald Wayne co-founded Apple. [E1]
```

Each `[E\d+]` references an entry in a runtime-built **evidence map** the model saw in its prompt. The runtime carries two ids per evidence object — a short `pointer_id` (`E1`, `E2`, … sequential within the run) shown to the model, & a content-addressed `evidence_id` (sha256-derived `E########`) used by the cache, run-DAG, & audit chain. Two-layer id discipline matters: showing the model the hex form tokenizes to several BPE tokens of out-of-distribution noise that nudges the model toward DSL/code-completion mode; showing it `E1` is one BPE token & lands inside the citation-style prose distribution that academic papers, Wikipedia articles, & footnoted text taught the model. Citation style is heavily represented in pretraining; random hex is not.

Two modalities stay first-class: pointer-line prose (`answer_mode = "claim_lattice_pointer"`) & JSON (`answer_mode = "claim_lattice"`). Both produce equivalent claim-evidence lattices & hash to distinct `governance_policy_hash` silos. Pointer-line suits 8B prose-distribution models; JSON suits grammar-constrained inference (vLLM `guided_json`, native JSON modes on larger models). An agent picks the modality that fits its inference path; the substrate exposes both, telemetry both, & never silently picks a winner.

The hard verifier runs nine deterministic checks on every claim-evidence pair:

1. **parser_succeeded** — the line had a recognized `[E\d+]` bracket.
2. **evidence_id_resolves** — the pointer maps to an entry in the runtime-built evidence map.
3. **source_role_allowed** — the resolved entry's `source_role` is in the policy allowlist.
4. **claim_text_non_empty** — the line carried prose around the tag.
5. **citation_coverage** — the claim's content tokens textually overlap the cited span at coverage ≥ a threshold (default 30%); short claims (≤3 content tokens) require ≥1 token match.
6. **pointer_count_within_cap** — `len(pointer_ids) ≤ max_pointers_per_claim` (default 2). Over-cited claims trim to the first N & a `POINTER_OVERFLOW_TRIMMED` violation is recorded so the run cannot reach the top rung of the ladder.
7. **anchor_class_warrant** — five anchor classes compose. Each gates on either question shape, claim content, or both. Proper-noun anchors gate on relation-shape questions: at least one Title-Case anchor must appear in some cited span. Date anchors are always-on, gated on the claim containing a 4-digit year (1500–2199) or a full English month name; ALL date components must appear. Entity-list anchors gate on entity-list-shape questions ("name the X", "list the Y"); at least one named entity from the claim must appear. Count anchors gate on count-shape questions ("how many X"); ALL count tokens must appear in either word OR digit form, with digit↔word equivalence ("two" ↔ "2"). Cause anchors gate on why-shape questions; at least one cause anchor (proper noun OR ≥5-char common noun outside a small stopword pool) must appear.
8. **title_relevance** — the cited evidence's source title must share ≥1 stemmed content token with the claim text. Per-claim, ANY-match across cited sources: the rule fires only when ALL cited titles miss. Catches retrieval-driven hallucinations where the cited chunk's source is structurally unrelated to the claim's subject — incidental vocabulary overlap inside the chunk passed Rules 1–7 but the source itself was never about the claim's subject.
9. **subject_tokens_absent** — content tokens shared by the question AND the claim must appear in the union of cited evidence spans. Per-claim, when ≥3 such tokens (default threshold) miss, the rule fires. Catches premise-parroting where the model echoes the question's distinctive subject in its claim while Rule 5 rode in on overlapping generic vocabulary. The 200-cycle bench-emergent run on 2026-05-02 surfaced exhibit A: a question about "correcter" / "steer" / "reply" cited a 33,500-character glossary of language-teaching terms whose chunks contained ZERO occurrences of any of those three words — generic linguistic vocabulary (language, communication, terms) carried Rule 5's coverage check while the question's distinctive subject went unverified. Rule 9 closes that asymmetry by checking the question-side tokens against the same cited-span union, not against the claim alone.

Warrant-lite is the lattice's first **semantically-grounded** hard check, & it earns its proof-path entry by staying lexical. Regex shape-detectors decide which classes activate. Title-Case proper-noun extraction, 4-digit-year extraction, full-month-name extraction, count-token extraction with digit↔word equivalence, & lowercase-common-noun extraction pull the candidate anchors. Case-insensitive substring tests ask whether the required anchors appear in cited spans. No NLI, no embeddings, no LLM-as-judge. The check is reproducible byte-for-byte & folds cleanly into `governance_policy_hash` via `policy["claim_lattice_warrant_check_enabled"]`. The principle: **pointer verification is not warrant verification**. A pointer resolves to a span; a warrant requires the span to actually contain the claim's load-bearing anchors.

The per-run Merkle-DAG (`run_dag_root`) commits each provenance step independently. Quote mode runs seven stages (question / retrieval / context / prompt / answer / verify / final_label); claim-lattice mode runs nine (question / retrieval / evidence_map / prompt / raw_answer / parsed_claim_lattice / verify / render / final_label). The new stages — `evidence_map`, `raw_answer`, `parsed_claim_lattice`, `render` — let an auditor reconstruct each transformation step from the persisted blob. The retrieval stage binds both the retrieval-plan hash (the operator-supplied parameters that selected sources) & the sources-summary hash (the documents that survived selection), so two runs with identical sources but different retrieval plans produce different `run_dag_root` values & the keyword string itself enters cryptographic provenance.

The renderer interpolates literal source spans from the evidence map at display time. A naïve truncation would show the same article-intro sentence under every claim when the model anchored at the topic id. Aborist instead extracts content tokens from each claim text & runs a *density rank* over their match positions in the cited span: find ALL occurrences of ALL tokens, then for each candidate position count how many DISTINCT tokens fall within ±half-window. Pick the position with maximum distinct count, expanded outward to the nearest sentence boundaries. Different claims pointing at the same evidence get different displayed sentences, anchored on what each claim is actually about.

Pointer mode also adopts a one-shot discipline: zero-shot ideal, one-shot maximum, no repeated retries until success. A run may fail honestly. A run may self-label uncertainty. A run may downgrade itself. A run may not keep trying until it passes. UNGROUNDED beats hallucinated success.

9. The Four-Rung Ladder

A single audit_mode token carries different weight across verifier methods. In quote / span / entity / paraphrase mode, a top label means every cited evidence unit textually matches a pinned source span — a strong claim about lexical grounding. In claim-lattice modes, the same token means every pointer resolves to a valid evidence object whose source_role is allowed AND every cited span passes citation coverage & the active anchor classes. That property is structurally weaker: synthesis-heavy claims (a model joining multiple cited spans into a single explanatory sentence) can pass all hard checks without semantic entailment of the joined assertion.

The renderer maps lattice-mode labels to a four-rung ladder where each rung names a strictly stronger property:

```
POINTER-LINKED       pointer / source / chunk verified;
                     warrant either did not apply or failed
ANCHOR-WARRANTED     pointer-linked + warrant passed where it ran;
                     other soft demotes may apply
EVIDENCE-WARRANTED   anchor-warranted + no soft demotes
UNGROUNDED           no verified pairs
```

A fifth rung — `ENTAILMENT-VERIFIED` — is reserved for a future committed entailment engine; today's stack does not produce it. Records that mix verified & unverified claims carry a `-PARTIAL` suffix on whichever rung applies, naming exactly that some claims passed & some did not.

Rung discrimination uses the violations list — no new verifier output field needed. `WARRANT_MISSING` violations drop the rung to POINTER-LINKED. Soft-demote violations like `LAZY_ANCHOR_DEMOTED`, `POINTER_OVERFLOW_TRIMMED`, `TOO_MANY_CLAIMS`, `BARE_NAME_CLAIM`, `TITLE_MISMATCH`, `SUBJECT_TOKENS_ABSENT`, `DEFLECTION_DETECTED`, & `FORMAT_COLLAPSED` cap the rung at ANCHOR-WARRANTED. The last four name distinct failure shapes operators read at audit-line glance: title mismatch flags retrieval-driven hallucinations against structurally-unrelated sources; subject-tokens-absent

flags premise-parroting where the question's distinctive tokens never reach cited evidence; deflection-detected flags topic-shift answers that ground unrelated facts; format-collapsed flags pointer-mode protocol abandonment when the model emits ≥5 meaningful prose lines with zero `[E\d+]` tags. Clean runs reach EVIDENCE-WARRANTED. The four-rung surface lands the property names directly so operators read what the lexical verifier could confirm without inferring from a `verifier_method` tail.

The schema column stays unchanged. The ladder is a pure display-layer transformation; cache_key, governance_policy_hash, & all programmatic callers see the underlying audit_mode unchanged. The ladder is the language of the paper & the language of the human-facing surface; the substrate beneath it is the v9.8 invariant the broader Merkle-AGI ledger commits to across systems.

10. Federation — Snapshot Roots & Mesh

A `snapshot_root` is `MerkleTree.build([sorted DISTINCT document_roots]).root` — one 32-byte hash naming the content-addressed forest at a point in time. Two peers that ingested the same dump compute bit-identical snapshot_roots, so cross-machine "are we synced?" becomes a single hash comparison instead of doc-by-doc gossip. Snapshot creation pins the root into the audit chain & records `(snapshot_root, taken_at, audit_event_hash, doc_count, parent_snapshot, reason)`. Snapshots auto-link to the most recent prior snapshot, giving a chain for free. The same snapshot_root doubles as the TF-IDF `scope_root`, so corpus-statistical cores reconcile under explicit corpus agreement.

The 2010-11 enwiki corpus, ingested on a single workstation in 1h 41m (4-way sharded), produces:

```
snapshot_root = 43797e46605de08dbab06cdcaf5be7ad78243b193c56f8580200dee6bcc7e1b9
doc_count     = 3,468,134 (deduplicated)
storage       = 38 GB across 4 shards
chunks        = 6,235,672
edges         = 90,593,523
audit_events  = 3,468,308 (chain intact, 30/30 random Merkle proofs verified)
```

Every one of those hashes is bit-identical on any machine that ingests the same `enwiki-20101011-pages-articles.xml.bz2` with the same aborist commit. Hand the snapshot_root to any peer; if their corpus matches, they compute the same hash, with mathematical certainty. No trust required, just shared inputs.

The mesh layer is wire-and-trust scaffolding for peers that already share a snapshot_root or want to. Cryptography:

- **Ed25519** signs every membership mutation & every gossip envelope. Forgery requires breaking Ed25519.
- **X25519 ECDH** [6] wraps each epoch's symmetric mesh secret to every current member's Diffie-Hellman pubkey. HKDF derives a 32-byte AEAD key from each shared secret.
- **ChaCha20-Poly1305** [7] authenticates payloads & per-member secret wraps; bodies optionally encrypt under the per-epoch shared secret with envelope metadata as additional authenticated data.

Membership state is per-epoch. Adding a member, kicking a member, or rotating the secret bumps the epoch & writes an audit event. Eviction works by rotating to a new epoch whose envelope omits the kicked member: their prior signatures stay verifiable forever (their old roster row is preserved), but they have no entry in the new envelope, so any AEAD-protected gossip from `epoch+1` onward is opaque to them.

Five message types carry across the wire, all wrapped in a single signed envelope:

```
ANNOUNCE_ROOT          document_root + source_uri + chunking/canonicalization/schema versions
ANNOUNCE_DERIVATION    core_root <- surface_roots(s), distiller_id, proof_blob hash
ANNOUNCE_PROVIDENCE    cache_key + audit_mode + answer_hash + verifier counts
ANNOUNCE_FALSIFICATION cache_key + reason + witness signature
REQUEST_BODY/DELIVER_BODY  body pull on local miss + per-chunk leaf hashes for client-side Merkle re-derivation
```

Receivers verify the Ed25519 signature against the sender's `sign_pub` looked up in `mesh_roster` at the envelope's `epoch_id`; non-members of that epoch produce no valid signature & their messages are silently

dropped. Each accepted `ANNOUNCE_*` writes one `mesh_received` event into the local audit chain whose body is the full signed envelope, preserving non-repudiation. `REQUEST_BODY` triggers a `DELIVER_BODY` carrying document text + per-chunk leaf hashes; the client re-derives the Merkle root from those leaves & refuses to insert any body whose leaves don't reconstruct the requested root.

The mesh is **off by default**. A `mesh.enabled` flag in the meta table gates everything. Fresh installs report `enabled: false` & no network code paths execute until the practitioner runs `aborist mesh init` & `aborist mesh enable`.

Three classes of derivative reconcile across peers:

- **Deterministic distillers** (e.g., first-sentence cores): same source_root → same core_root, on every machine. Shareable as-is via the existing `derivations.proof_blob`.
- **Corpus-statistical distillers** (e.g., TF-IDF keywords): convergent only under shared corpus scope. Requires a `scope_root` commitment so peers explicitly agree on which corpus statistics underlie their cores.
- **LLM-derived answers**: non-convergent (serving non-determinism), but cryptographically witnessable. The same cache_key from many peers + the same `answer_hash = sha256(answer_text)` collapses to one row with many witnesses; divergent answers stay as multi-witness candidates, each Merkle-bound to a verifiable context_root.

Personal vs. public chains. The same federation primitive supports a private corpus on one machine & a public corpus shared by a thousand peers — trust regime is a per-document choice, not a platform decision. Private chains stay local: personal session history, no data leaves the machine, suits corporate knowledge bases / sensitive internal documentation. Public chains broadcast records to mesh peers under the same Merkle proofs — they share proofs, never document content. Document text stays on the originating client; only the cryptographic certificate travels. A practitioner can run both simultaneously, partitioned by domain or per-document opt-in (e.g., a private chain over patient notes alongside a public chain over open medical guidelines). The eight-dimensional cache_key partitions all of it cleanly.

11. The Truth-Seeking Ratchet

Emergence is preserved, not suppressed. UNGROUNDED records & POINTER-LINKED records stay live in the providence cache. Training-derived knowledge is often correct, just unverifiable against the current corpus. The substrate labels honestly so a downstream consumer can route: top-rung records carry compliance-grade provenance, UNGROUNDED records carry "the model believes this, no certificate attached." A model that genuinely refuses ("I don't know based on the provided sources") classifies UNGROUNDED with `verifier_method='none'` & `n_quotes=0` — the most honest answer the system can produce.

Emergence is also a **corpus-growth signal**. Unverified spans persist on each providence record; `aborist emergent --aggregate` ranks them by frequency. Spans the model produces repeatedly that the corpus cannot ground are candidate ingest targets — what training has that ingestion is missing. When the missing source eventually arrives, `aborist reclassify` re-runs the verifier against existing answers without any LLM call, persists the upgraded label, & writes one `providence_reclassify` audit event per changed row. POINTER-LINKED records promote to ANCHOR-WARRANTED. ANCHOR-WARRANTED promotes to EVIDENCE-WARRANTED. UNGROUNDED records that find their grounding promote to whichever rung the verifier now reaches. The audit chain remembers the transition. Prior records stay as historical witnesses: *on this date, against that corpus root, the model emerged this much beyond its sources*.

The ratchet only goes one way. A record can climb the ladder when new evidence arrives. A record cannot silently fall down the ladder; demotion requires a verifier change that bumps `governance_policy_hash` & cleanly partitions the cache namespace, leaving the prior record live under the prior policy. Re-classification under the new policy starts a fresh cache record at the appropriate rung. The two records coexist, bound to their respective policies, & an auditor can compare the same answer's classification under different verifier regimes to study how the substrate's standards have evolved.

Truth-seeking is therefore a property of the *substrate*, not of the model. The model emits its best guess. The substrate measures the gap between the guess & the available evidence, names the gap honestly, persists

both, & rewards corpus growth that closes the gap. Over time, & across many practitioners, the substrate accumulates a body of verified answers whose ladder placements reflect what humanity has so far been able to prove. Emergence today is the corpus-growth roadmap of tomorrow.

12. Decisions and Constraints

Eleven rules anchor the substrate. Each is a deliberate choice with a rationale; reverting any one without addressing the rationale weakens the system in a specific named way.

1. **The verifier stays binary.** Each evidence unit either verifies or does not. No per-unit confidence scores, no fuzzy-match indicators, no soft labels. Soft signals proliferate elsewhere — sidecars, retrieval rankers, token-coverage probes — but never enter the proof path. *Rationale:* classification is a hard bit, not a soft score, & enters `governance_policy_hash` cleanly. Once a soft signal contaminates the chain, every prior cryptographic claim becomes negotiable.
2. **The runtime owns quote text.** In claim-lattice mode the model never types a quote string; it types a pointer. The runtime interpolates the literal source span at render time. *Rationale:* synthetic-elision-by-construction-impossible. A model cannot produce a frankenquote it cannot type.
3. **Cores never evict.** The eviction policy reclaims surface-tier documents when storage pressure mounts; cores are derivatives & their underlying surfaces can be re-fetched, but the core itself stays pinned. *Rationale:* a core represents irreversible work — distillation, summarization, recursive cores-of-cores — that cannot be cheaply reproduced. Evicting a core forfeits the work; evicting a surface forfeits a fetch.
4. **One-shot discipline.** A run may fail honestly. A run may self-label uncertainty. A run may downgrade itself. A run may not keep trying until it passes. *Rationale:* UNGROUNDED beats hallucinated success. Iterative repair + reprompt-feedback loops produce records that cannot be distinguished from cherry-picking. The substrate prefers an honest failure to an opportunistic success.
5. **Idempotent re-ingest.** Same content → same `document_root` → no-op insert. Same URI + different content → new doc + `supersedes` edge. *Rationale:* re-running the pipeline is always safe. Nothing duplicates. Nothing overwrites silently. Lossless history.
6. **Local-first; federation opt-in.** The mesh layer is off by default. A fresh installation makes no network calls until the practitioner explicitly enables them. *Rationale:* the practitioner's data is the practitioner's data. Sharing requires consent, & consent requires the practitioner know what is being shared.
7. **Emergence is preserved, not suppressed.** UNGROUNDED & POINTER-LINKED records stay live. Reclassify promotes them up the ladder when new evidence arrives. *Rationale:* see §11. The truth-seeking ratchet is the architecture's reason for existing.
8. **Soft hash never enters the proof path.** Embeddings, TF-IDF scores, lexical similarity counts shape ranking & retrieval; they never feed cache_keys, document_roots, or audit_event_hashes. *Rationale:* a ranker that drifts as new training data arrives must not invalidate prior cryptographic claims. The hard channel & the soft channel evolve on independent timelines.
9. **Cache is not consent.** A cached answer is a record of what the runtime computed & what the verifier could confirm. It is not a claim about whether the user agreed with the answer, found it useful, or wants it shared. *Rationale:* the cache is content-addressed, not endorsement-addressed. Endorsement, ratings, & social signals belong in separate tables that can be edited & retracted; the cache is append-mostly history.
10. **Governance dimensions partition the cache, never the corpus.** Bumping `governance_policy_hash` stales cache records on lookup & cleanly partitions the namespace. It does not touch surfaces, cores, derivations, or the audit chain. *Rationale:* the corpus is the ground truth. Governance evolves; the corpus does not get rewritten when governance evolves.
11. **Labels name properties, not vibes.** POINTER-LINKED, ANCHOR-WARRANTED, EVIDENCE-WARRANTED, UNGROUNDED each name a property the verifier could lexically confirm or could not. There are no "high confidence", "moderate confidence", or "low confidence" labels. *Rationale:* a confidence label is a soft signal in disguise. Naming the property — pointer linked, anchor warranted, evidence warranted, ungrounded — lets an auditor read the same label & immediately know what was checked.

13. What This Proves and What It Does Not

The pointer-mode verifier proves *evidence-linked, role-OK, coverage-met, source-title-relevant, &*, when the active anchor class fires, *anchor-named-in-cited-chunk* for every active class. It does **not** prove that the cited evidence semantically entails the claim. A model citing `[E1]` for "Brachiosaurus exhibits social herding behavior in the film" passes the eight hard checks if E1's source is the *Jurassic Park (film)* article (Rule 8), the word *Brachiosaurus* appears in E1 (Rule 5), & the relation/date/list/count/why anchor classes do not fire on a *behavior* claim — even if E1's text says nothing about social herding.

The structural-and-lexical floor is what is proved. The semantic ceiling is what is not. The honest report on every claim-lattice answer is "this claim is evidence-linked, role-OK, lexically grounded, & warrant-named; semantic entailment is not yet hard-verified." The spotlight renderer makes residual gaps visible: when the cited span has no token cluster, the displayed excerpt falls back to the leading window — an auditor reading EVIDENCE-WARRANTED with leading-window excerpts under a relation-shape question immediately sees that the verifier accepted on coverage but the chunk's load-bearing slice for this claim is sparse.

That ceiling gap is by design. NLI-grade entailment requires either a textual-entailment model (soft signal) or LLM-as-judge (model round-trip) — both of which would re-introduce the soft/hard boundary leak the architecture is built to prevent. Today's stack is honest about the ceiling: when a small model emits a claim its cited evidence cannot semantically support, the verdict is ANCHOR-WARRANTED with a smell-sidecar warning, never EVIDENCE-WARRANTED with a bogus citation. A future committed entailment engine, deterministic enough to enter the proof path without staining the soft/hard boundary, lifts records to ENTAILMENT-VERIFIED; until that engine exists, the rung is reserved & deliberately empty.

The architecture's promise to its readers is calibration. It does not claim to verify everything. It claims to name exactly what it can verify, & to leave a structural gap visible where it cannot. A practitioner reading a top-rung label knows what the substrate could prove. A practitioner reading any lower rung knows what the substrate could not. Honesty about the ceiling is a feature, not a limitation.

14. Related Work

Reverse RAG [1]. Documented the client-side inversion of standard RAG: the browser already holds the document, full-page injection replaces chunk fragmentation, small models with perfect context outperform large models with similarity-retrieved fragments. This paper extends Reverse RAG with verifiable caching, eliminating the repetition tax that prior RAG systems do not address.

Retrieval-Augmented Generation [2]. Introduced server-side vector retrieval for language models. Standard RAG remains the dominant pattern for chunk-similarity retrieval; this paper's substrate inverts the trust direction (the runtime owns provenance, not the retriever).

Git & Mercurial object models. Both implement content-addressed Merkle DAGs natively. Git's commit object hashes the entire tree; this paper's substrate inherits this as a free cache boundary signal. Aborist makes the implicit Merkle-tree property of version control explicit & useful for ML inference caching.

Zero-Knowledge Proofs [8] [9]. Merkle membership proofs constitute a subset of ZKP: proving a leaf exists in a tree without revealing other leaves. This paper applies the membership-proof primitive to ML answer provenance without requiring full ZKP apparatus. We trade completeness for simplicity & speed.

Merkle-AGI v9.8 substrate. Aborist's underlying admissibility ledger inherits the eight-dimensional cache key, the falsification states, the audit chain, & the trichotomy schema column from the broader Merkle-AGI substrate. This paper describes the runtime spec implemented on top of v9.8; the substrate is generic across implementations, the runtime is one possible mapping.

Merkle's original construction [3]. The cryptographic primitive underlying every modern Merkle tree, including Bitcoin's transaction tree, Git's object model, & this paper's content-addressed substrate. The construction has not been improved upon in any architecturally relevant way since 1979; we apply it to ML provenance without modification.

References

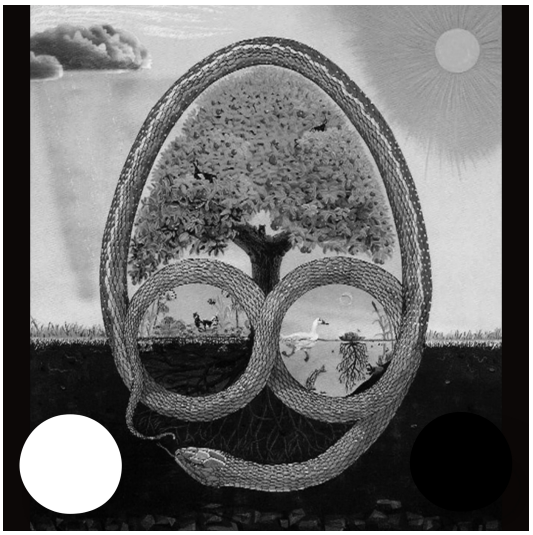
Acknowledgements

Our Merkle cache idea emerged from a conversation on X Spaces. We thank **@AR_Milne** & **@without_rulers** for a discussion that sparked the core insight: that a Merkle tree over document content could serve as a fast, verifiable cache layer for machine learning inference. Ideas born in public conversation belong to our commons.

Citation

```
Russell Ballestrini, David Wong, Riley Morgan.
"Unturf Automated General Intelligence: Merkle Providence Reverse RAG."
unfirehose.com, 2026.
https://unfirehose.com/merkle-providence-reverse-rag.html
```

License



GNU Affero General Public License v3

```
AGPL-3.0-only

PERMACOMPUTER PREAMBLE - NO WARRANTY

This is free software for the public good of a permacomputer hosted at
permacomputer.com - an always-on computer by the people, for the people. One
which is durable, easy to repair, and distributed like tap water for machine
learning intelligence.

The permacomputer is community-owned infrastructure optimized around four values:

  TRUTH    - Source code must be open source & freely distributed
  FREEDOM  - Voluntary participation without corporate control
  HARMONY  - Systems operating with minimal waste that self-renew
  LOVE     - Individual rights protected while fostering cooperation

This paper contributes to that vision by documenting Merkle Providence Reverse
RAG: a provenance-preserving cache layer that lets small language models punch
far outside their parameter class. Code is seeds to sprout on any abandoned
technology.

Learn more: https://www.permacomputer.com

NO WARRANTY. THE SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND.

Copyright (C) 2026 Russell Ballestrini & David Wong & Riley Morgan
russell@unturf.com
david.s.wong@pm.me
guybriley02@gmail.com
https://www.unturf.com/software
https://www.permacomputer.com
https://uncloseai.com
https://unfirehose.com
https://russell.ballestrini.net

--------------------------------------------------------------------------------

                    GNU AFFERO GENERAL PUBLIC LICENSE
                       Version 3, 19 November 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU Affero General Public License is a free, copyleft license for
software and other kinds of works, specifically designed to ensure
cooperation with the community in the case of network server software.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
our General Public Licenses are intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

  Developers that use our General Public Licenses protect your rights
with two steps: (1) assert copyright on the software, and (2) offer
you this License which gives you legal permission to copy, distribute
and/or modify the software.

  A secondary benefit of defending all users' freedom is that
improvements made in alternate versions of the program, if they
receive widespread use, become available for other developers to
```

incorporate. Many developers of free software are heartened and encouraged by the resulting cooperation. However, in the case of software used on network servers, this result may fail to come about. The GNU General Public License permits making a modified version and letting the public access it on a server without ever releasing its source code to the public.

  The GNU Affero General Public License is designed specifically to ensure that, in such cases, the modified source code becomes available to the community. It requires the operator of a network server to provide the source code of the modified version running there to the users of that server. Therefore, public use of a modified version, on a publicly accessible server, gives the public access to the source code of the modified version.

  An older license, called the Affero General Public License and published by Affero, was designed to accomplish similar goals. This is a different license, not a version of the Affero GPL, but Affero has released a new version of the Affero GPL which permits relicensing under this license.

  The precise terms and conditions for copying, distribution and modification follow.

                       TERMS AND CONDITIONS

  0. Definitions.

  "This License" refers to version 3 of the GNU Affero General Public License.

  "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

  "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

  To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

  A "covered work" means either the unmodified Program or a work based on the Program.

  To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

  To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

  An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

  1. Source Code.

  The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention

```
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

  4. Conveying Verbatim Copies.

  You may convey verbatim copies of the Program's source code as you
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

  5. Conveying Modified Source Versions.

  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.

    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".

    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
    invalidate such permission if you have separately received it.

    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.

    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
```

    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.

    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source. This
    alternative is allowed only occasionally and noncommercially, and
    only if you received the object code with such an offer, in accord
    with subsection 6b.

    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
    Corresponding Source in the same way through the same place at no
    further charge. You need not require recipients to copy the
    Corresponding Source along with the object code. If the place to
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source. Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
    available for as long as needed to satisfy these requirements.

    e) Convey the object code using peer-to-peer transmission, provided
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.

  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling. In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage. For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product. A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

  "Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source. The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

  If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information. But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

  The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates

for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not

excuse you from the conditions of this License. If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all. For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.

  13. Remote Network Interaction; Use with the GNU General Public License.

  Notwithstanding any other provision of this License, if you modify the
Program, your modified version must prominently offer all users
interacting with it remotely through a computer network (if your version
supports such interaction) an opportunity to receive the Corresponding
Source of your version by providing access to the Corresponding Source
from a network server at no charge, through some standard or customary
means of facilitating copying of software. This Corresponding Source
shall include the Corresponding Source for any work covered by version 3
of the GNU General Public License that is incorporated pursuant to the
following paragraph.

  Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU General Public License into a single
combined work, and to convey the resulting work. The terms of this
License will continue to apply to the part which is the covered work,
but the work with which it is combined will remain governed by version
3 of the GNU General Public License.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU Affero General Public License from time to time. Such new versions
will be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number. If the
Program specifies that a certain numbered version of the GNU Affero General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation. If the Program does not specify a version number of the
GNU Affero General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU Affero General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

  Later license versions may give you additional or different
permissions. However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY

```
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
state the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Affero General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU Affero General Public License for more details.

    You should have received a copy of the GNU Affero General Public License
    along with this program.  If not, see <https://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

  If your software can interact with users remotely through a computer
network, you should also make sure that it provides a way for users to
get its source.  For example, if your program is a web application, its
interface could display a "Source" link that leads users to an archive
of the code.  There are many ways you could offer source, and different
solutions will be better for different programs; see section 13 for the
specific requirements.

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU AGPL, see
<https://www.gnu.org/licenses/>.
```

1(1, 2) russell@unturf, cthegray, TimeHexOn, foxhop. *Reverse Retrieval Augmented Generation: Client-Side Context Injection for Small Language Models*. uncloseai.com, 2026. https://uncloseai.com/reverse-retrieval-augmented-generations-rag.html

2 Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., Kiela, D. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. NeurIPS, 2020.

3(1, 2) Merkle, R. C. *Secrecy, Authentication, and Public Key Systems*. Stanford PhD thesis, 1979.

4 Meta AI. *The Llama 3 Herd of Models*. Technical report, 2024.

5 NousResearch. *Hermes 3 Technical Report*. 2024.

6 Bernstein, D. J. *Curve25519: New Diffie-Hellman Speed Records*. PKC, 2006.

7 Bernstein, D. J. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439, 2018.

8 Goldwasser, S., Micali, S., Rackoff, C. *The Knowledge Complexity of Interactive Proof-Systems*. STOC, 1985.

9 Bitansky, N., Canetti, R., Chiesa, A., Tromer, E. *From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge*. ITCS, 2012.

10 SQLite Consortium. *SQLite FTS5 Extension*. https://sqlite.org/fts5.html

11 Collet, Y. *Zstandard*. https://facebook.github.io/zstd/

12 Earwig (Ben Kurtovic). *mwparserfromhell*. https://github.com/earwig/mwparserfromhell

13 *Aborist — Python reference implementation under AGPL-3.0-only*. https://git.unturf.com/engineering/unturf/aborist