TensorCash Mining API: Engine Integration, Proof Capture, and Coexisting Service (v1)

Version 1.0 — mainnet-inception implementation; stable interface, evolving internals.

ABSTRACT

TensorCash mines a blockchain by running honest forward passes through a publicly identified open-source language model. For the resulting blocks to be useful as both consensus and as inference output, the engine producing them must do three things at once: it must serve real user requests at competitive latency, it must continuously generate proof-bearing windows whenever the GPU would otherwise be idle, and it must persist a precise, statistically-rich transcript of every generated token without making the forward pass measurably slower. None of those three is naturally compatible with stock vLLM or llama.cpp; in combination they require a deliberate engineering surface.

This document describes that surface. The Mining API is a fork of two inference engines (vLLM for CUDA workers, llama.cpp for CPU and Apple Silicon workers), shared proof utilities implemented in Python/Torch and C++ next to the engines' samplers, and a Python proxy that fronts the engine and arbitrates between paying user traffic and synthetic backfill mining. Together they implement what we call the

no-toll proof contract

: a transcript faithful enough to satisfy the verification ladder of the Verification Whitepaper, reduced while the logits are still in the engine's sampling path, so that proof capture does not turn the autoregressive sampling barrier into a second full-vocabulary workload.

The mining interface is intentionally a **v1 implementation contract**. The consensus object is specified by the Verification Whitepaper; this document specifies how an engine and proxy produce that object while still serving traffic. The open questions here are primarily implementation debates: where proof capture sits in the decode loop, which engine optimisations remain compatible with deterministic sampling, how aggressively synthetic work should yield to user traffic, and how backfill prompts should be generated. v1 fixes specific answers so that mainnet can ship; the interface is meant to remain stable while the internals continue to evolve.

TABLE OF CONTENTS

I. Setting and Motivation

The Verification Whitepaper specifies, as a consensus object, what a *proof* of inference must contain: a 256-token transcript of chosen tokens and the sampler probability/CDF coordinate carried by the proof schema, a deterministic per-step uniform $u_t \in [0, 1)$ tied to the rolling token context and the block's cryptographic scaffolding, 70 logit evidence entries (the top 50 logits plus 20 deterministic vocabulary probes) with their indices, bucket means over the full logit vector and a global mean, softmax normalisers, and the precision and model identifiers under which the trajectory was computed. From the verifier's side this is a finite, well-typed object roughly 160 KB to 1 MB in size.

From the miner's side it is something else entirely. Producing it requires reading off the right tensor at the right moment of every sampling step, in the right precision, before the engine's normal sampling pipeline overwrites or releases it. Doing that wrong has three distinct failure modes:

- **Sampling-barrier tax.** Autoregressive decoding has a serial barrier at every token: the next forward pass cannot begin until the current token has been sampled and appended to the context. Proof capture sits at that same barrier because it must bind to the exact distribution that produced the token. If the proof path has to materialise, reduce, hash, or serialise a full vocabulary distribution at this point, that work cannot be hidden behind the next decode step; it lengthens the token loop itself. A full-distribution transfer is one expensive version of the problem, but the deeper constraint is the causal dependency between sampling and the next forward pass. The implementation treats that boundary as scarce: proof storage is preallocated, per-token memory traffic is kept deliberately small, vectorised indexing and batched reductions are used where the engine exposes them, and every piece of work that can be moved off the causal path — export, cache insertion, broker callbacks, monitoring, and cleanup — is pushed onto asynchronous paths. What remains on the barrier is the work needed to choose the token and bind the proof to the exact distribution that chose it.
- **Accuracy drift.** Capturing logits *after* the engine's sampler has applied temperature, top-k, top-p, and softmax truncation gives a transcript the verifier cannot replay, because the sampler is part of the sampling contract: the verifier must reconstruct the *same* truncated distribution from the same effective sampling surface. A capture point downstream of the canonical sampler transform is consensus-invalid.

- **Liveness drag.** A miner who only mines when there is no user traffic accumulates almost no proofs, because GPU duty cycle in a serving stack is naturally bursty. Conversely, a miner who never yields to user traffic cannot sell inference. The architecture must keep the GPU saturated with a mix of mining and serving such that user requests can preempt mining promptly, subject to backend cancellation mechanics, and mining resumes when the user load drops.

The Mining API is the answer to those three constraints. It is a forked engine whose sampler emits compact proof evidence alongside normal decoding, fronted by a proxy whose job is to keep the engine continuously busy while never starving paying traffic. Everything else in this document is implementation of that idea.

II. The Mining Hot Path

Before describing the fork it is useful to record what the

unmodified

hot path looks like, because every modification is justified relative to it. For both vLLM and llama.cpp, an autoregressive decode step through a transformer block reduces, at the level of cost and information flow, to the same sequence:

1. embed the previously sampled token,
2. run the residual stream through L transformer blocks (each a multi-head attention followed by an MLP), accumulating into a final hidden state,
3. project the hidden state through the unembedding matrix to produce a logit vector $L_t \in \mathbb{R}^V$,
4. apply repetition penalty, temperature, top-k, top-p, and softmax to obtain a truncated sampling distribution π_t,
5. draw $\tau_t \sim \pi_t$ using the engine's RNG,
6. emit τ_t to the response stream and append it to the running context.

The logit tensor at step (3), plus any request-level masking or penalty state that is part of the sampling contract, is the last point in the pipeline where the verifier-replayable distribution still exists in dense, untruncated form. The later stages are destructive: top-k throws away $V-k$ values, top-p truncates by cumulative mass, and softmax erases the offset information that the verifier uses to detect quantisation tampering. Capturing the proof transcript therefore must tap the effective sampling surface before temperature and truncation, and must reduce the verifier's evidence before the next token is scheduled. The performance question is not merely whether the full V-wide vector crosses a bus. It is whether any V-wide proof-side operation is placed on the serial sampling barrier where the next forward pass is waiting for a token.

The Mining API's central engineering claim is that the verifier-relevant content of step (3) can be reduced inside the engine process into a compact record before the sampling barrier closes. In the current vLLM path this reduction uses tensor-native operations over GPU tensors; in the llama.cpp path it uses C++ over the logits already visible to the server process. The forked sampler records the first 50 top-ranked logits, 20 deterministic probe logits, their indices, bucket statistics, normalisers, the sampled token, and the deterministic sampler coordinate u_t. Later hash and serialization work is kept to compact transcript messages, and in v1 may run on CPU; the no-toll property is about keeping the unavoidable serial proof work small, not about pretending proof bookkeeping is free. The total per-step proof record is approximately

$$B_{\text{proof}} \approx 79 \cdot 4\,\text{B} + 70 \cdot 4\,\text{B} + 8\,\text{B} + O(1) \approx 0.6\,\text{KB},$$

before envelope overhead, against a per-step forward-pass arithmetic intensity in the tens of gigaflops range. The proof tax is therefore bounded by compact reductions and compact hashing, rather than by a second full-distribution workload inserted into the autoregressive loop.

III. Engine Fork: Instrumenting vLLM and llama.cpp

Two engines are forked because two distinct miner classes need to exist. vLLM is the upstream engine for CUDA accelerators with continuous batching and PagedAttention; it is what a serious GPU miner runs. llama.cpp is the engine for CPU operators, Apple Silicon, and small edge accelerators; it has lower throughput but radically lower minimum hardware. Both must produce a proof transcript that the same verifier can score, regardless of where it was produced.

i. The vLLM fork

vLLM's sampler is the natural insertion point. Upstream vLLM has a sampling pipeline in which penalties, temperature, truncation, softmax, and token selection occur at the end of each decode step. The fork records the effective sampling surface after request masks and penalties have been applied, but before temperature, top-k, top-p, and softmax truncate the distribution. Ordinary requests stay on the ordinary sampler path. Mining requests enter the deterministic sampling path, which applies TensorCash's canonical top-k/top-p and hash-derived CDF sampling rules so that the verifier can replay the result.

The non-trivial work is inside that sampling path rather than in a separate full-distribution proof pass. The current implementation uses tensor-native reductions and sorting over the logits already resident in the vLLM process: it stores the first 50 top-ranked logits and indices, adds 20 deterministic vocabulary-spaced probe logits, computes the post-truncation log normaliser, records the full-temperature logsumexp, and records bucket/global means used by the verifier's statistical checks. This is an intentionally conservative v1 implementation. A future fused CUDA kernel may reduce overhead further, but it is not part of the shipped contract described here.

Continuous batching is what makes fixed slot assignment necessary. At any moment vLLM may have hundreds of sequences in flight, with each forward pass corresponding to a different mix; a sequence may be paged in, paged out, preempted, and resumed. The fork therefore introduces a slot allocator that maps active sequences to fixed proof-storage slots, allocating a slot when the sequence is admitted and releasing it when the sequence finishes or is evicted. Free slots are reused in a deterministic order; if capacity is exhausted, the helper evicts an older or most-progressed synthetic slot rather than guessing from GPU state. The allocator lives on the host side; the hot sampling path sees only an already-resolved proof-storage slot.

The sampler is also the place where the **runtime metadata** binding is carried forward. Each mining request is annotated, before the first decode step, with the model identifier, declared compute precision, and configuration needed by the verifier. That metadata is written into the proof envelope alongside the per-step rows. In v1 this is runtime metadata supplied and propagated by the miner stack, not hardware attestation; the verifier still relies on the statistical ladder and model identifier checks described in the Verification Whitepaper.

ii. The llama.cpp fork

llama.cpp is structurally simpler than vLLM but its sampler chain is also more visible. The server detects a mining sequence and routes it through TensorCash's deterministic sampling path instead of the ordinary sampler chain. That path reads the logits exposed by llama.cpp for the current token, applies the canonical sampling rules, records the same 50-top-plus-20-probe evidence and statistics, and checks completed

windows on the configured cadence. Speculative decoding is disabled for mining sequences so that proof rows correspond one-for-one with committed tokens. On CPU and Apple Silicon the bandwidth concern is different — there is no PCIe bus to avoid — but the ordering concern is the same: the proof must be sourced from the sampler surface the verifier can replay.

llama.cpp's server mode complicates one detail. Whereas vLLM is a Python service whose sampler can be subclassed in process, llama.cpp runs as a C++ subprocess under a Python supervisor that proxies HTTP endpoints and FlatBuffers responses. The supervisor is a thin process manager — it hands the engine the model file, the context size, the GPU layer count, and the mining configuration on the command line, and it relays serialised mining responses back upstream. The supervisor does not see logits. All proof capture happens inside the C++ subprocess, emitted as proof envelopes, and the supervisor's contribution is operational rather than cryptographic.

iii. Why two engines and not one

There is a recurring temptation to converge the worker on a single engine. We do not, for a deliberate reason: the Verification Whitepaper is an *engine-portable* contract by design. The proof object is expressed in transcript primitives — token ids, u_t values, compact logit evidence, bucket means — that can be produced by any autoregressive transformer implementation. Maintaining two independent forks is the strongest evidence the network can offer that the proof object is not a vLLM-specific artefact. If a third engine emerges with materially different operational properties (a JAX-based path on TPUs, a CPU-only SIMD path with no GPU dependency at all, a fully attestation-bound TEE-resident inference stack), it should be admissible to the network without modifying the verifier. The fork-pair pattern is a continuous test of that claim.

IV. The Proof Tensor: What Is Captured Per Step

The fork's job is to populate, for every decode step of an active sequence, a fixed-shape record. The record per step is:

- the chosen token τ_t,
- the sampler probability/CDF coordinate used to show where the sampled token landed in the canonical distribution,
- the deterministic uniform draw u_t derived from the sampler hash (fp32),
- 70 logit evidence values: the first 50 top-ranked logits plus 20 deterministic vocabulary probes (fp32 in v1),
- the corresponding 70 vocabulary indices,
- the post-truncation softmax log normaliser,
- six logsumexp statistics: full-temperature logsumexp, four bucket means, and a global mean (fp32),
- scalar validity and padding masks.

The shipping storage packs these values into fixed-width float and index blocks; the exact array layout is an implementation detail. A 256-token window with at most R concurrent proof slots in flight is then an R-by-256 array of these records. At $R = 64$ the main proof tensors are on the order of tens of megabytes rather than model-scale memory; a larger configured slot count costs proportionally more. This is the single most important quantitative property of the design: the proof tensor is small enough to live next to the engine that produced it for the duration of the window.

The tensor is also organised so that a completed window can be serialised and hashed as a compact transcript without re-reading the full vocabulary logits. That is what makes the boundary export path feasible; we discuss it in section 6.

V. Proof Utilities and the Ring Buffer

The proof utilities are deliberately aligned across the two engine forks, even though they are not a single shared binary. The vLLM path uses Python/Torch storage and hashing helpers, with optional C++ proof processing for export; the llama.cpp path links C++ helpers directly. What they share is the FlatBuffers schema and transcript grammar. Four components matter architecturally.

First, proof storage is pre-allocated and fixed-shape. Its size is fixed at startup; the hot path writes into existing storage rather than growing proof arrays token by token. One active sequence occupies one proof slot, and that slot advances along the window dimension as new tokens are written. Because the storage is ring-shaped along the window dimension, writes are modulo-256 indexed and require no compaction.

Second, a host-side slot allocator maps active sequences to proof storage. It allocates a slot when the sequence is admitted, releases it when the sequence finishes or is evicted, and applies deterministic eviction when capacity is exhausted. The v1 helper chooses an old or most-progressed synthetic slot when no free slot exists. This allocator is the only proof-storage component that must update host state on every step.

Third, an export component constructs the proof object for a completed window. It attaches the prompt tokens, pad mask, bucket and global statistics, model identifier, precision tag, IPFS CID when available, VDF and tick scaffolding, and auxiliary flags required by the verifier, then serialises the record into the consensus FlatBuffers schema. The transcript fields are deterministic. The envelope may also include operational metadata such as timestamps and request identifiers, so v1 does not claim that two runs at different wall-clock times produce byte-identical blobs.

Fourth, a compact transcript hasher produces the deterministic sampler coordinates and the final window digest. In the vLLM path, the current implementation batches compact hash messages and performs the SHA-256 work through a CPU hop; in the llama.cpp path, the C++ code uses the platform crypto library. That is still cheap relative to a full forward pass because the hasher consumes the compact proof transcript, not the full vocabulary logit vector. At the window boundary, the stack computes the final digest $H_\star$ used by the block-candidate path; a successful nonce triggers export.

The four objects together implement a contract we can state precisely:

> Per generated token, the miner performs the verifier-required full-distribution reduction at the sampling barrier, stores a compact sub-KB row, and hashes compact transcript messages. The cost is real but bounded; the economic claim is that proof capture adds a small amount of serial work to ordinary inference, rather than inserting a second vocabulary-wide pass into the autoregressive loop.

That is what we mean by

no toll

in v1. It is a profiling target and an implementation discipline, not a claim that bookkeeping has zero cost. A naive proof-capture path that waits on full-distribution materialisation, transfer, or serialization at every token would slow the serial decode loop enough that an honest miner could not compete with one who skipped the proof altogether.

VI. The Boundary Digester and In-Memory Proof Cache

The boundary digester is the bridge between a completed proof window and the chain. The per-step sampler hash produces the deterministic uniform u_t values used during decoding. Separately, when a mining sequence reaches a 256-token boundary, the final hash $H_\star$ is computed from the block scaffolding, tick, a zero step index, the tail-256 completed-token window, and the precision tag, as specified in the Verification Whitepaper. Its leading bytes supply the candidate nonce material for the block path. A miss discards that candidate window for block production; a hit triggers export.

Export is asynchronous and best-effort. A successful solution is serialised and sent toward the local node over ZeroMQ; when proxy proof publishing is enabled, the same message is also published to the host-side **proof collector**. Non-solution audit proofs can be published to the proxy collector without becoming block candidates. The collector caches proof blobs in an in-memory **proof cache** keyed by the request's completion id. The proof cache is an LRU-and-TTL map with a default budget of 500 megabytes and a 600-second TTL, sized so that a miner can survive a brief network outage or broker callback delay without losing recent proof material.

The collector is deliberately narrow, but it is not blind byte storage. It parses the proof envelope, extracts the completion id, request id, and model identifier where present, drops stale proofs when mining is paused, rejects mismatched broker request ids or model identifiers in the active request context, and then inserts the blob into the cache. It does no verifier-grade statistical or recomputation validation; validation remains a verifier responsibility. Holding the proof cache and the verifier on the same node would conflate two roles; the network is built so that they can be operated independently, including by independent parties when a miner is renting compute through the broker.

VII. The Miner Proxy and Synthetic Backfill

Everything described so far is necessary but not sufficient for profitable mining. The forked engine produces proofs only for inputs it actually serves. If the engine sits idle waiting for user requests, no proofs are produced, no nonces are tried, and no blocks are mined. A node that mines only when paying traffic happens to flow through it is leaving most of its hashpower on the table.

The **Miner Proxy** closes that gap. It is a Python asyncio service that fronts the engine's HTTP surface and presents a simple invariant to the operator: while mining is enabled, the proxy tries to keep at least an operator-configured active pool in flight, and refuses to exceed an operator-configured maximum. If the active count drops below the floor, the proxy spawns synthetic *backfill* requests to fill the gap. If a real user request arrives while backfill is occupying capacity, the proxy can preempt an abortable backfill request to make room.

Backfill is generated by an **intelligent prompt generator** that draws prompts from a structured corpus of templates and word banks. The current corpus is intentionally simple and operational: it keeps idle workers decoding varied synthetic requests without introducing user data into the mining stream. It does not, by itself, prove that every trajectory will be non-degenerate under the verifier's sampling contract. That responsibility remains with the verifier's statistical gates, including the low-entropy checks described in the

Verification Whitepaper. A separate genesis-oriented path can use a fixed public seed phrase, which helps make early-network mining reproducible; it is not a consensus mechanism, and the chain's actual binding against premining is the cryptographic anchoring described in the Verification Whitepaper.

Two design choices in the backfill path deserve naming.

First, **backfill output is real inference output, not waste**. A backfill completion is a generated chat response just like any other. The miner proxy is structurally agnostic to whether a response is useful to a paying user or not; the engine produced it identically. This means a miner who is connected to a discovery surface — for example, a community board where unanswered questions are matched against available compute — can convert backfill capacity into useful served traffic at any time, simply by routing the question through the same proxy. Mining is therefore not a separate workload from inference serving; it is the same workload, charged differently.

Second, **the proxy never injects a real user's data into a backfill slot**. Backfill prompts come from the synthetic corpus, never from queued user requests. A user whose request is rejected because the queue is full is told the queue is full; their content is not buffered into the backfill stream. This is a privacy-rather-than-throughput choice and it is deliberate.

VIII. Priority-Queue Coexistence with User Traffic

The proxy's priority machinery is the most operationally consequential component in the entire stack. A miner who serves real users and also mines must guarantee that the second activity does not degrade the first; otherwise the operator's customers depart and the operator no longer has a business to mine inside.

The request priority layer is a small asyncio scheduler with two classes of work — user or broker traffic, and synthetic backfill — and four invariants:

1. The proxy generates synthetic work only when the active count is below the configured active pool, and admits no more than the configured maximum. In the default deployed proxy configuration the maximum is twice the active pool size unless overridden.
2. When a user or broker request arrives, the scheduler aborts an in-flight synthetic request if an abortable one exists, even if the queue has not yet reached its maximum. If there is no synthetic request to preempt and the queue is already full, the request is rejected rather than silently queued. The selection policy favours recently admitted or high-batch-position synthetic work because that minimizes wasted completed window progress. Abort latency is backend-dependent: newer response paths can call the engine's cancellation endpoint, while older completion paths are coarser and rely on task cancellation.
3. User and broker requests are non-abortable once admitted. The proxy never preempts a user to make room for another user, and never preempts a user to make room for a backfill job. The backfill stream lives entirely in the slack between user load and the configured capacity ceiling.
4. The manager publishes counters — total external, total dummy, currently in flight, total aborted — that the local node and broker sidecar can scrape, so an operator can verify in real time that user traffic is not being throttled by mining activity.

The aggregate effect is that the engine's GPU duty cycle is driven toward high utilisation without giving backfill equal priority with users. At low user load, the GPU is busy generating proof-bearing backfill. At high user load, backfill contracts and user requests fill the slots they would have occupied. Mining yield scales with idle capacity; user latency is protected by admission and preemption rather than by treating mining as a separate workload.

The priority layer is also the integration point with the **compute broker** when a worker is rented. In the normal proxy mode, broker-mode inference requests arrive through the same admission path that an HTTP user request would, and consequently enjoy the same preemption guarantees over backfill that a paying user does. In broker-only execution modes, the broker request may be forwarded more directly to the engine, while the proof collector and callback path still bind proof blobs back to the broker request id. The miner is therefore free to advertise broker capacity at the same moment it is mining: the admission layer arbitrates between the two without changing the proof format.

IX. Operational Picture: A Single Request Through the Stack

To make the architecture concrete, consider a single request, served by a TensorCash miner that is also accepting external chat traffic and is registered with the compute broker.

A user POSTs a chat completion request to the miner's HTTP surface. The proxy classifies it as user traffic and admits it, possibly aborting in-flight synthetic work to make room. The proxy attaches the model identifier, runtime precision declaration, and mining parameters needed by the verifier, then forwards the request to the engine.

The engine schedules the request into its continuous batch. On the first proof-bearing decode step, the slot allocator reserves proof storage for the new sequence, while the request metadata is carried forward for the eventual proof envelope. From that point onward, every decode step writes compact proof data — the 50-top-plus-20-probe logit evidence, indices, bucket means, normalisers, chosen token, CDF coordinate, and sampler u_t — into that storage through the engine's deterministic sampling path.

If the request is a 256-token completion that the user actually wants, the engine streams tokens back through the proxy as it always would. At the boundary, the stack computes $H_\star$ from the completed token window and block scaffolding, then checks the corresponding block candidate against the adjusted target. A miss on the target is the overwhelmingly common case; the sequence may continue into a later window, but the missed candidate is not exported as a block. A hit triggers export: the exporter serialises the proof, the local node receives the candidate, and the collector caches proxy-published proof material when that path is enabled.

If the user disconnects mid-generation, the proxy aborts the request and the proof slot is freed; no proof is exported. If a user request arrives while capacity is occupied by abortable synthetic work, the scheduler aborts one synthetic request and admits the user; the aborted request's proof slot is freed without a proof. If the engine is idle, the proxy spawns a synthetic request from the prompt corpus, which goes through the same forward path and either yields a winning nonce or reaches its boundary without exporting a block candidate.

In all four cases — paid completion, user disconnect, paid completion that preempts synthetic work, synthetic work that runs to completion — the engine ran the

same forward pass

. The proof bookkeeping is identical. The distinction between "mining" and "serving" exists only at the admission and accounting layer; below the proxy, there is just inference.

X. Implementation Directions

The architecture described above is what mainnet ships. The verifier remains an open research surface, but the mining side is mostly an implementation contract: engines must produce the same proof object without damaging the decode loop. The open questions below are about portability, scheduling, and performance rather than changing the consensus object.

1. **Engine portability beyond vLLM and llama.cpp.** The proof schema and transcript grammar are engine-portable, but the two existing forks were non-trivial integrations. A formal porting guide that documents the sampler-insertion contract — access to the verifier-replayable logit surface, proof-slot allocation, runtime metadata binding — would reduce the marginal cost of admitting a third or fourth engine to the network. A JAX/TPU port and a SIMD-only CPU port are the two most interesting candidates.
2. **Adaptive backfill load.** v1's proxy runs with configured active pool and maximum admission settings. A controller that tracks user latency and GPU thermal headroom and adjusts those settings dynamically would let a miner extract more hashpower from the same hardware without measurably degrading user latency, at the cost of a nontrivial control problem.
3. **Backfill quality fingerprinting.** Backfill prompts must exercise a non-degenerate sampling regime to be useful, but v1's prompt generator is intentionally modest. A future version may subject backfill streams to the same statistical fingerprinting the verifier uses on proof windows, allowing the miner to detect and re-roll degenerate trajectories before they consume a window.
4. **Proof cache durability.** v1's in-memory cache is intentionally ephemeral: a node restart loses unflushed proofs. Persisting the cache to disk introduces complications (proof revocation on reorg, replay protection), but a durable cache would let a miner tolerate longer outages between proof production and block relay.
5. **Attestation-bound capture.** A TEE-resident worker can attest to the bit-pattern of the model in GPU memory at the moment a proof is captured. Combining hardware attestation with the statistical verifier ladder of the Verification Whitepaper would shorten the evidence path for the model-identifier and precision-tag bindings. v1 ships without TEE-specific paths in the engine fork; admitting one is straightforward and explicitly invited.
6. **Broker-aware preemption.** v1 treats broker-routed requests as equivalent to local user requests once they are admitted. A future version may admit a per-route SLA so that a broker-routed request with a tight deadline can preempt a broker-routed request with a slack deadline — a richer scheduling surface than the current user-traffic-versus-backfill split.
7. **Speculative decoding compatibility.** Mining sequences currently avoid speculative decoding because a proof row must correspond to a committed token sampled from the verifier-replayable target distribution. A future implementation could recover speculative speedups by proving only accepted target-model tokens, by recording acceptance decisions as part of the transcript, or by designing a verifier rule that treats the draft model as an implementation detail while preserving the target distribution.
8. **Incremental window reuse.** Backfill jobs commonly target bounded 256-token completions, while longer external or broker requests may naturally roll across multiple proof windows. A future version may make *windowed mining* an explicit policy surface in which the same prefill amortises across many

256-token proof windows in sequence, increasing the effective token-per-FLOP yield for miners running long-context models. The chain-level fairness implications of long-context amortisation are discussed in the Verification Whitepaper and remain open.

What we ask of v1 is not finality but **adaptability**: that the interface between engine, proxy, and verifier remains stable enough for v2 authors to swap implementations without breaking the network, while the internals continue to be replaced as inference hardware and engine architecture improve.