Uniquant

A Mined ERC-8004 Quantum Agent with a Self-Hook

May 2026

Abstract

UNIQUANT (ticker UQUANT) is an autonomous on-chain agent you mine into existence. It is an ERC-20 token released entirely through proof of work — no team allocation, no insider presale, no admin keys, no upgrade path — that is simultaneously its own Uniswap V4 swap hook and its own miner. One bytecode, one address, three roles. Once deployed, the rules cannot change.

The supply is 21 million tokens, Bitcoin scale. Five percent funds an open genesis sale, five percent seeds a Uniswap V4 UQUANT/ETH pool, and the remaining ninety percent is mined: anyone with a browser can solve a keccak256 puzzle bound to their wallet and call `mine()` for the current era's reward. Mining is deliberately slow — a ten-minute target and a one-mint-per-block cap stretch emission across years, not weeks.

Around the token sits an identity layer. Holders claim a soulbound *Miner Agent* NFT — one of ten 1/1 quantum-agent artworks whose tier sharpens live with their balance — and the contract itself is built to register as a trustless ERC-8004 agent. Looking forward, that identity is designed to migrate to post-quantum (quantum-resistant) signatures, secured by the same keccak256 primitive that powers the mining.

This document describes why the design exists, how each piece works, where it differs from a hard-coded "locked-LP" launch, and what limits remain.

Why this exists

Most token launches are pre-mints. A team deploys a contract, mints the supply to a multisig, and decides distribution later. The "fair launch" variants swap the multisig for a vesting contract, which is just a slower version of the same problem.

A different model has existed since 0xBitcoin in 2018: the contract refuses to mint to anyone. New tokens emerge only when someone solves a hash puzzle on chain. The schedule is mechanical, the distribution is permissionless, and the team holds no special lever because there is no team-controlled supply to abuse.

UNIQUANT takes that model and adds three things. The first is **address-bound proofs of work**: a mined nonce only works for the wallet that found it, so solutions cannot be lifted from the mempool. The second is a **self-hook**, made possible by Uniswap V4 hooks shipping in January 2025 — the token contract is also the swap hook for its own pool and skims a 1% fee on every trade into its own balance. The third is an **agent identity layer**: the contract is shaped to register as an ERC-8004 agent, and every participant can claim a soulbound NFT identity that scales with their stake.

Address-binding kills mempool front-running at the cryptographic level. The self-hook removes the need for a third-party fee router. The identity layer makes ownership legible to agent-aware tooling. None of this requires trusting a team with the supply.

How the contract is organized

The contract moves through four phases.

Phase 0, Empty. Deployed and holding nothing. Constructor parameters set the Uniswap V4 PoolManager, PositionManager, and Permit2 for the target chain. The deployer becomes the `controller`, the only address with privileged functions (fee withdrawal and pool management — see below). No supply or schedule parameter is mutable from outside.

Phase 1, Genesis. Anyone can call `mintGenesis(units)` with `units * 0.01 ETH`. Each unit yields 1,000 UQUANT, capped at five units per transaction so a single mempool slot cannot sweep the allocation. Excess ETH is refunded in the same call. Three days after deploy, if the pool has not been seeded, `refundGenesis` opens and lets any holder of genesis UQUANT redeem it for ETH at the original price (see Refund escape hatch).

Phase 2, Seeding. Once the genesis cap of 1,050,000 UQUANT sells out, anyone can call `seedPool()`. It mints the remaining 19,950,000 UQUANT to the contract, creates the V4 UQUANT/ETH pool with the contract as its hook, deposits all genesis ETH plus 1,050,000 UQUANT as liquidity, and sets the initial mining difficulty. The V4 LP position NFT is minted to the controller. A fallback `partialSeed()` covers a stalled genesis; the controller can call it no earlier than thirty minutes after deploy.

Phase 3, Mining. The remaining 18,900,000 UQUANT is released through proof of work. Each successful `mine(nonce)` pays the current era's reward, starting at 100 UQUANT and halving every 100,000 mints. Mining ends when the cap is reached.

Phase transitions are gated by storage flags. No admin function can skip a phase, rewind state, or change the supply schedule.

The mining protocol

A mined nonce is valid for a wallet at a given epoch when

```
keccak256(abi.encode(challenge, nonce)) < currentDifficulty
```

where

```
challenge = keccak256(abi.encode(chainId, contractAddress, miner, epoch))
epoch     = blockNumber / 600
```

Four properties matter.

The challenge changes every six hundred blocks, roughly twenty minutes on Base. A miner who fails to find a solution within one epoch cannot carry the work forward — the seed shifts and the search restarts. This caps the cost of difficulty mispricing and gives the network natural retargeting windows.

The `miner` field is `msg.sender`. Bob's search is for a different challenge than Alice's. Even if Bob copies Alice's pending transaction and rebroadcasts it under his own address, his version computes a different hash and almost certainly fails the difficulty check. Front-running mined solutions is impossible at the cryptographic level, not merely at the ordering level.

Replay protection lives in a `usedProofs` mapping keyed on `keccak256(miner, nonce, epoch)`. The same triple cannot be claimed twice.

Mining is intentionally slow. Difficulty retargets every 2,016 mints — Bitcoin's cadence — toward a target of **300 blocks per mint**, i.e. one mint per ten minutes on Base's two-second blocks. The expected window per retarget is 2,016 × 300 = 604,800 blocks, about fourteen days at target rate, clamped to a factor of four per period. A hard cap of **one mint per block** removes any opening burst: a second `mine()` in the same block reverts, so no party with surplus hashpower can sweep consecutive blocks.

Starting difficulty after seeding is `type(uint256).max >> 42` — roughly one valid hash in 4.4 trillion. That is orders of magnitude harder than a naive launch, deliberately so: it prevents the first minutes from emitting a meaningful slice of supply, and the retarget then converges to the ten-minute target.

Because the reward halves every 100,000 mints, the supply is front-loaded but the tail is long: the bulk of the 18.9M mining allocation emits over the first several years, and full exhaustion lands on the order of eight years at target rate. UNIQUANT is a slow asset by design.

The self-hook

A Uniswap V4 hook is a contract whose address carries specific bits in its lower fourteen bits, encoding which lifecycle callbacks it implements. The same contract that implements `transfer` and `balanceOf` for the ERC-20 also implements `beforeInitialize`, `beforeSwap`, and `afterSwap` for the pool. To make the address carry the right bits it is deployed via CREATE2 with a mined salt; the search is offline, reproducible, and takes seconds. Anyone can verify the result: mask the address with `0x3FFF` and check it equals `0x20CC`.

After seeding, the pool exists at: currency0 native ETH, currency1 UQUANT, tickSpacing 200, hooks the UQUANT contract itself. The hook intercepts swaps and routes **1% of the ETH side of every swap** into the contract's own balance. Accumulated ETH sits there until the controller calls `claimFees()`, which transfers the contract's entire ETH balance to the controller. This is the only value-moving privileged function; it cannot touch the token supply, the mining schedule, or any holder's balance — only ETH the contract earned from swap fees.

On liquidity — read this carefully. UNIQUANT does *not* hardcode a liquidity lock. The hook enables only the initialize and swap callbacks; it does not gate `addLiquidity` / `removeLiquidity`. The seed liquidity is a standard Uniswap V4 position whose NFT is held by the controller, who can therefore manage it through the normal PositionManager — including adding to it or withdrawing it. This is a deliberate choice for this launch: it keeps the controller able to respond to the pool (for example, to step in during disorderly trading) rather than welding the liquidity shut forever. The trade-off is explicit: liquidity trust rests on the controller, not on the bytecode. We state this plainly rather than market a lock that isn't there.

Tokenomics

Allocation	Amount	Share
Genesis sale	1,050,000 UQUANT	5%
Seed LP	1,050,000 UQUANT	5%
Mining	18,900,000 UQUANT	90%
Total	**21,000,000 UQUANT**	**100%**

Genesis is priced at 0.01 ETH per 1,000 UQUANT, fixed. A fully subscribed genesis raises 10.5 ETH, all of which goes into the V4 pool as the ETH side of seed liquidity. Nothing reaches the deployer at genesis.

The mining schedule halves every 100,000 successful mints. Era zero pays 100 UQUANT per mint, era one pays 50, era two 25, and so on; cumulative rewards reach the 18.9M cap around era four. At the ten-minute target this is roughly 144 mints per day, which is why full emission stretches across years.

These percentages hold at full sell-out. If `partialSeed` runs below the cap, unsold genesis UQUANT is never minted (see Seeding strategy), so the genesis and LP shares shrink and the mining share grows proportionally — the 18.9M mining allocation is fixed in absolute terms regardless.

Seeding strategy

Seeding is the only timed decision the controller makes. Two functions enable it:

```
seedPool()    — permissionless, requires genesisMinted == GENESIS_CAP
partialSeed() — controller only, requires block.timestamp >= deployedAt + 30
minutes
```

Both call the same internal logic; the only difference is whether genesis reached the full 1,050,000 UQUANT cap. `seedPool` is the happy path; `partialSeed` is the escape hatch for a stalled sale.

The useful property of `partialSeed` is how it treats unsold genesis. The body uses `genesisMinted`, not `GENESIS_CAP`:

```
function _seedBody() internal {
    uint256 eth     = genesisEthRaised;
    uint256 tokenLP = genesisMinted;
    _mint(address(this), tokenLP + MINING_SUPPLY);
    // create the pool with (eth, tokenLP) as liquidity
}
```

Unsold genesis UQUANT is never minted — not burned, not held, not stuck in storage; it simply never exists. If genesis stops at 300,000 UQUANT sold, total supply becomes 300,000 (held by buyers) + 300,000 (LP) + 18,900,000 (mining) = 19,500,000 UQUANT, and the pool is seeded with 300,000 UQUANT against whatever ETH was raised.

The 30-minute delay on `partialSeed` is the only anti-grief constraint: it stops the controller from instantly seeding an empty pool, which would yield zero liquidity and break the AMM. The `genesisMinted > 0` check enforces that at least one external buyer participated.

The downside of early seeding is economic, not technical. Mining supply is fixed at 18,900,000 UQUANT regardless of how much genesis sold; the LP side scales with sales. Seeding at 30% gives roughly a third of the full-sellout depth, and a thin pool against active mining means each mined token presses price harder. The recommended path is a **threshold seed** — commit publicly to seeding at a chosen percentage with a deadline fallback — so the community has a number to rally around without the open-ended uncertainty of a strict sell-out that may never arrive.

Refund escape hatch

Genesis collects ETH on the contract and trusts that a seed function will eventually run. If neither does, the ETH would otherwise be stranded. `refundGenesis` covers two failure modes — a controller who disappears, and a controller who tries to seed but cannot:

```
function refundGenesis(uint256 tokenAmount) external nonReentrant {
    if (genesisComplete)                                  revert GenesisAlrea
dyComplete();
    if (block.timestamp < deployedAt + REFUND_GRACE)      revert RefundGraceN
otPassed();
    if (tokenAmount == 0 || tokenAmount % GENESIS_UNIT != 0) revert MustBeUni
tMultiple();

    uint256 units   = tokenAmount / GENESIS_UNIT;
    uint256 ethBack = units * GENESIS_PRICE;

    _burn(msg.sender, tokenAmount);
    genesisMinted    -= tokenAmount;
    genesisEthRaised -= ethBack;

    (bool ok,) = msg.sender.call{value: ethBack}("");
    if (!ok) revert EthTransferFailed();

    emit GenesisRefund(msg.sender, ethBack, tokenAmount);
}
```

`REFUND_GRACE` is three days. Once it passes, any holder of genesis UQUANT can redeem at the original 0.01 ETH per 1,000 rate, provided the pool has not been seeded. Partial refunds are allowed.

Three properties keep it safe. It is gated on `!genesisComplete`: after seeding, the ETH is liquidity in the pool, not on the contract, so refunds become impossible by design — post-seed exits happen by selling into the AMM. The unit-multiple check matches genesis granularity and avoids dust drift against `genesisEthRaised`. And `_burn` happens before the ETH transfer, so combined with `nonReentrant` there is no path to replay a refund.

The three-day window is short enough that a committed controller can seed first, and long enough that a good-faith controller can debug a failing `partialSeed` without refund traffic competing for the same ETH. After three days with no seed, the deployment quietly unwinds: buyers withdraw, the contract balance trends to zero, and nobody is left holding tokens they did not burn back.

Verification

The contract source is published under the MIT license with its build configuration. After deployment to Base mainnet it is verified on Basescan; the same source verifies on Sourcify or any compatible explorer. The bytecode, constructor arguments, and CREATE2 salt are reproducible from source.

The mining algorithm is implemented in Rust, compiled to WebAssembly, and runs entirely in the browser. The Rust source ships in the same repository; the WASM bundle is a static asset served from disk with no server-side computation.

Before mining, anyone can confirm two things independently. First, the hook bits: take the deployed address, mask with `0x3FFF`, and verify the result equals `0x20CC`. Any address that fails this is not the real contract. Second, the constants: read `TOTAL_SUPPLY`, `MINING_SUPPLY`, `GENESIS_CAP`, `BASE_REWARD`, `ERA_MINTS`, `EPOCH_BLOCKS`, `ADJUSTMENT_INTERVAL`, and `MAX_MINTS_PER_BLOCK` directly from chain; each must match the values stated here. Note that, by the liquidity design above, the pool is *not* hook-locked — do not expect a liquidity-removal call to revert.

Agent alignment

The UNIQUANT contract maps cleanly onto the ERC-8004 agent model. The EIP describes a "trustless agent" through three properties — stable identity, observable reputation, verifiable behavior — and UNIQUANT satisfies all three without a wrapper.

The identity is the contract's own address, fixed at deploy time, derived from the CREATE2 init-code hash, with no proxy in front and no admin key to rotate. The address is the agent.

The reputation is on-chain state: `totalMints`, `totalMiningMinted`, accumulated swap fees, genesis ETH raised. Every metric is queryable directly; there is no oracle to trust because every claim the agent makes can be checked against its own storage.

The behavior is the source: the bytecode is the spec, and both are immutable. The mint function never pays beyond `MINING_SUPPLY`; `claimFees` only moves ETH the contract earned.

On the canonical ERC-8004 Identity Registry (`0x8004A169FB4a3325136EB29fA0ceB6D2e539a432`, the same address on every chain), UNIQUANT registers with a single call to `register(agentURI)`, where `agentURI` points to a hosted manifest (`/agent.json`) listing capabilities, endpoints, and validation hooks. Indexers such as 8004scan then resolve the agent automatically from the registry NFT's `Transfer` event. *(Registration is performed once the production contract is deployed; the assigned agent id is published on the site and in `agent.json` at that time.)*

Miner Agent NFTs

A separate soulbound ERC-721 collection, `MinerAgent`, gives each UNIQUANT participant a queryable on-chain identity without touching the core token. One NFT per address, claimable once, soulbound after mint: any wallet holding at least 1 UQUANT may call `claim()`. Genesis buyers, miners, and aftermarket buyers all qualify equally — the only gate is the live balance check.

The artwork is a curated set of **ten 1/1 quantum-agent pieces** — side-profile cybernetic characters with prism quantum visors — named Quantum Azure, Obsidian Fracture, Emerald Hood, White Signal, Olive Cipher, Blue Lattice, Oracle Veil, Cap Node, Bronze Relay, and Polygon Beret. They are organised as five tiers times two variants. The tier scales dynamically with the holder's live UQUANT balance — Initiate (under 1,000), Bronze (1k–10k), Silver (10k–100k), Gold (100k–1M), Platinum (1M and above) — so a wallet that grows from Silver to Gold visibly upgrades its badge

with no on-chain action. The variant is fixed at mint, hashed deterministically from the tokenId, so two holders at the same tier may display different artwork.

The images are pinned to IPFS as a single content-addressed CIDv1 folder, provably immutable: even if the original pin disappears, anyone re-pinning the same files reproduces the same CID. Metadata is served from `/api/agent/<tokenId>.json`, which reads `ownerOf(tokenId)` on `MinerAgent` and `balanceOf(owner)` on UNIQUANT live on every fetch, then assembles OpenSea-compatible JSON pointing at the matching IPFS image. Each token also back-references the parent ERC-8004 agent so an indexer can resolve the registry entry without a separate hint.

Transfers between externally-owned accounts revert with `Soulbound()`. The collection grants no new rights over UQUANT, changes no supply, and adds no tradeable surface; its job is to make ownership legible to agent-aware tooling and give each participant one durable identity.

Quantum-resistant identity

An on-chain agent is meant to be permanent, and permanent identity should plan for the one thing that breaks today's signatures: the quantum computer. This section states the direction honestly — it is design and roadmap, not yet shipped on-chain.

Every account on Ethereum and Base signs with ECDSA over secp256k1. A sufficiently large quantum computer running Shor's algorithm can recover a private key from its public key and forge signatures. The timeline is uncertain — likely years — but the asset at risk is precisely the thing you cannot re-issue: a persistent identity. Hashing is far more resilient; Grover's algorithm only halves a hash's effective strength, fixed by using wide outputs. Signatures are the soft spot, not the keccak256 mining.

The plan is to bind the agent identity to a **post-quantum signature scheme** from the hash-based family (SPHINCS+ / Winternitz / Lamport). These rely only on the security of a hash function — the same keccak256 primitive that mines UQUANT — so the identity is secured by the same primitive that brings it into existence, with no new trust assumption and no exotic curve. Base makes this practical: post-quantum verification is gas-heavy and would be economically absurd on Ethereum L1, but on Base, where gas settles in cents, on-chain verification becomes affordable.

The roadmap is incremental and stated plainly. **Phase 1 (design):** the threat model and direction, published — this section, and the site's `/quantum` page. **Phase 2 (registration):** agents register a post-quantum public key alongside their ERC-8004 identity, with signatures verified off-chain and attested. **Phase 3 (on-chain verifier):** a Solidity verifier for hash-based signatures gates high-value actions.

To be unambiguous: UNIQUANT runs **classical** keccak256 proof of work. There is no quantum computer in the loop and we will never claim one. *Quantum-resistant* means resistant to attacks *by* quantum computers, via post-quantum cryptography — a real, NIST-standardized field — shipped incrementally, with the current phase always stated.

Limits and caveats

The contract has no upgrade path. If a critical bug surfaces post-launch, there is no fix; the mitigation is Bitcoin's — keep the surface small, audit before deploy, fork code that has run in production.

Liquidity is controller-managed, not bytecode-locked (see The self-hook). This buys flexibility at the cost of a trust assumption: holders trust the controller not to withdraw the pool adversarially. This is the most important thing to understand before buying, and we will not dress it up as a lock.

Mining costs gas. On Base a `mine()` transaction costs cents. The reward in dollar terms must exceed that for mining to stay rational; as price falls toward the mining cost, hashrate drops, throughput falls below the ten-minute target, and difficulty retargets down. Equilibrium price is whatever clears that condition.

Address-binding makes solutions unstealable from the mempool but does nothing against a miner who controls multiple wallets. As with Bitcoin, the cost of computing power — not identity — is what discourages over-extraction.

The 1% swap fee paid to the controller is an ongoing extraction from swap volume, not a token tax: the swap executes at the pool's posted price and the fee is taken from the ETH side via the hook, paid equally by buyers and sellers. It can never be raised, lowered, or rerouted because the contract is immutable.

UNIQUANT is not a promise. There is no team behind it in the conventional sense and no roadmap it is contractually obligated to deliver. The post-quantum direction is an intention, not a covenant. The contract does what its bytecode does, and nothing more.