



ATOMICALS CORE ENGINE

ATOMICALS CORE ENGINE

Litepaper – Version 1.0

Published: April 2025

Authoring Team: ACE Core Contributors

Status: Public Release

<https://github.com/faction2028/ACE/blob/main/litepaper.pdf>

Table of Contents

NOTE	3
INTRODUCTION.....	4
WHAT IT DOES.....	5
ATOMICALS PROTOCOL.....	6
PROPOSED AVM	8
AVM IN THE ATOMICALS CORE ENGINE.....	9
LAYER 3: COMPILER & LOGIC SYSTEM	12
LAYER 3: LOGIC STORAGE	13
SECURITY AND TRUST MODEL.....	15
ACE INDEXER.....	18
CLI SYSTEM	21
BATCHING LAYER: AVM ITIT MIDDLEWARE.....	23
WALLET SYSTEM.....	24
GUN.JS DATA LAYER	26
GUI SYSTEM AND TAURI APP.....	28
DESIGN PHILOSOPHY & SYSTEM SUMMARY	30
REFERENCES & FURTHER READING	34

NOTE

Opening Note

Last week we introduced **Faction 2028**. This week, we're releasing the first public version of the Atomicals Core Engine (**ACE**) litepaper. Previously known as **TAP** (renamed to avoid confusion with the Tap Protocol), ACE is a complete execution environment for the Atomicals Protocol, designed to power individual interaction, games, apps, DAOs, and tools to move the Atomicals Protocol into the future.

To clarify, Faction 2028 and ACE are being developed together but are completely separate. ACE is the engine. Faction 2028 is one example of an app built on that engine. Think of it like Mario and Nintendo: the same team may build both, but anyone can develop on the platform. ACE is open to all. However, we are building Faction 2028 and ACE to complement each other to encourage development, user numbers, and community engagement.

This document is about ACE. It outlines a full-stack system that includes a compiler, wallet, AVM runtime, indexer, CLI, and GUI. The engine is designed to be fully general-purpose, open source, and not dependent on any external source except Bitcoin.

From the start, our goal was to align with the Atomicals Protocol and the Atomicals Virtual Machine (AVM). That remains true. However, due to uncertainty around the original AVM roadmap, we built our own complete and sandboxed AVM runtime. We remain committed to the Atomicals vision and have structured ACE to support trustless state transitions, container logic, and all core protocol features.

We are releasing this litepaper now while ACE is still in active development. That means there is still time to evolve certain components based on feedback. While the architecture is opinionated and the foundation is already stable, we are open to community input where alignment improves the ecosystem overall.

We will begin opening direct feedback channels on May 6, 2025, through our official communication platforms. Until then, we invite developers, creators, and community members to explore the litepaper and begin imagining how they will use or create with ACE.

- ACE Core Contributors

Introduction



The Atomicals Core Engine (ACE) is a modular, Bitcoin-native execution environment designed for apps and games built on top of the Atomicals Protocol. It unifies indexing, logic execution, state management, and user interaction into a single, downloadable desktop program. Our goal is to radically simplify development while preserving Bitcoin's core ethos of transparency, permanence, and decentralization.

Our structure allows for an individual or application to use our engine to interact, perform operations, or run entire platforms utilizing the Atomicals Protocol.

ACE is built entirely in Rust, wrapped in a secure Tauri container, with a Dioxus frontend that communicates solely through the Rust backend. At its core is a secure and sandboxed **AVM** (Atomicals Virtual Machine) as well as a Layer 3 for more complex logic which syncs with every other node for trustless verification including a hash check for additional safeguards. Additionally, for data that is not essential for on-chain committing we incorporate a gun.js layer only exposed to the rust backend for restricted read/write access. Everything operates locally, with the app actively synchronizing through a custom-built indexer to track Bitcoin and Atomicals State.




Rather than being a collection of disjointed tools, ACE presents a single cohesive interface for users and developers alike. The wallet, indexer, CLI, and execution layers are all tightly integrated under one GUI. They remain modular in architecture for future extensibility, but unified in operation as a single, complete system.

Each instance of ACE is also a node. But rather than relying on consensus in the traditional sense (e.g., Byzantine models), trust is established in two key ways:



-  Bitcoin-native anchoring: Scripts and states committed via the Atomicals Protocol can always be independently validated. All indexers that sync Bitcoin/Atomicals Protocol entries will reach the same state reconstruction.
-  Layer 3 state matching: Logic stored in Layer 3 must match expected hash values. Every time an app or site queries **ACE**, it must pass a hash verification test. This ensures integrity and prevents tampering. Sites or apps can also optionally provide public certificates, allowing users to verify cert numbers within ACE for added trust.

What It Does

ACE serves as a full-stack, self-contained execution environment for AVM-based apps and games. It revolves around three core components:




-  **AVM Sandbox:** A deterministic runtime engine purpose-built for executing logic on the Atomicals Virtual Machine (**AVM**). The sandbox interprets a compact instruction set optimized for trust-minimized, Bitcoin-native execution, with full support for local and optionally on-chain verification.
-  **Indexer + CLI Layer:** A custom-built indexer that stays in sync with Atomicals and Bitcoin chain state, delivering accurate, real-time data. The CLI acts as the system's command layer—automating tasks, running logic, interfacing with the indexer, and bridging internal modules. This layer is modular and written in Rust for performance, determinism, and future support for features like Bitwork mining logic.
-  **Layer 3: Logic + Compiler:** A high-level logic system supporting languages like Python and JavaScript. Developers annotate scripts using `@avmtags`, which are transpiled into AVM bytecode. This lets users write expressive, flexible code while maintaining trust in only the AVM-executed logic. Layer 3 also validates inputs, compiles logic flows, and enables modular execution units. Every node holds the complete code bundle for each application, enforcing a trustless environment in which all connected apps must match an exact hash to validate their logic before connecting.

In addition to these three pillars, ACE includes integrated tools like:

-  **Wallet** – to manage Atomicals, track holdings, and sign logic updates. Supports Taproot-based UTXOs, ARC20 Tokens, NFTs, and more.
-  **Explorer** – for inspecting on-chain data, exploring active apps, and browsing network activity, all from your local indexer without third-party APIs.

All of this is wrapped in an elegant Tauri GUI application, combining native performance with a secure architecture. The Rust backend controls all critical operations, including access to the GUN.js data layer, while the Dioxus frontend provides a clean and accessible interface for users.

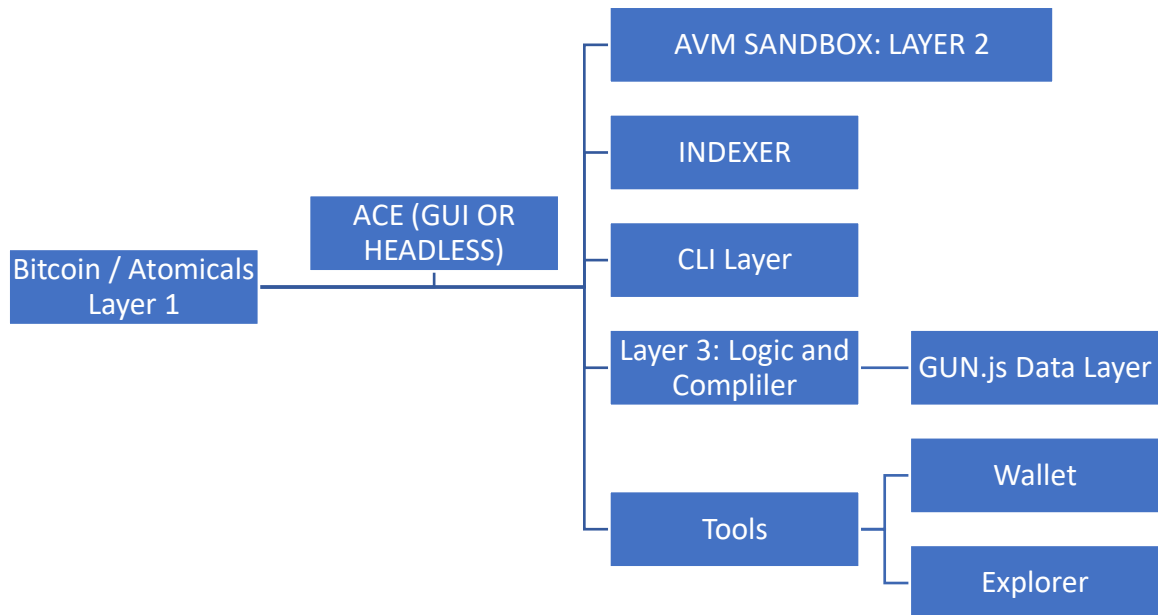
Our application unlocks capabilities that the current environment doesn't allow:

-  Enable high TPS and bypass dust limits through batching and queuing
-  Prevent sniping and malicious actor interference
-  Simplify onboarding for users and developers

The system is designed to be open-source, locally owned, and developer-extendable—allowing contributors to build new modules, modify components, or fork the system entirely.

The following diagram shows the major internal components of the ACE execution environment. Data flow interactions will be detailed later.

ACE MAJOR COMPONENTS OVERVIEW









Atomicals Protocol

The Atomicals Protocol is a Bitcoin-native framework for minting, updating, and interacting with decentralized digital objects called Atomicals. These are self-contained units of data, logic, and identity, all anchored directly to Bitcoin’s base layer via ordinal inscriptions.

Unlike most digital asset protocols, Atomicals does not require a new blockchain, token, or validator set. It leverages Bitcoin’s existing infrastructure to create a trust-minimized, permissionless system where the full lifecycle of an object, from creation to mutation to transfer, can be traced, verified, and reproduced independently by anyone running an indexer.




Atomicals are created by embedding metadata directly into Bitcoin transactions, forming the structured basis for containers, names, logic hooks (structures that can trigger scripts or smart behavior), or NFTs. Once minted, an atomical becomes a permanent, independently verifiable, and evolvable on-chain entity, with its complete state history immutably committed to Bitcoin.

What Can an Atomical Represent?

-  **NFTs:** Immutable or dynamic objects that hold structured data, including art, characters, assets, or evolving metadata.
-  **Fungible Tokens:** Currency, voting shares, and utility tokens tracked with native supply logic.
-  **Game Objects:** Items, characters, inventories, or worlds represented as updatable containers.
-  **Whitelists & Access Tokens:** Permission systems that live directly on-chain and evolve through logic.
-  **Auditable Containers:** Full object history available for forensic inspection, governance, or compliance.
-  **Identities & Profiles:** Self-owned, portable identifiers bound to the Atomical Protocol.

Because the protocol is extensible, new object types can emerge organically without requiring permission from a foundation, DAO, or central authority.

Core Protocol Concepts

-  **Containers:** Hierarchical, mutable namespaces for grouping Atomicals. You can think of them like folders or dynamic smart folders. They can hold tokens, metadata, logic rules, or other Atomicals, creating flexible structures that evolve over time.
-  **Bitwork:** A novel proof-of-work mechanism tied directly to individual Atomicals. Bitwork can be used to increase the difficulty of minting a specific object or to gate certain state updates behind work thresholds or to enforce resource expenditure during key state transitions, adding native resistance to spam or bot attacks. It is lightweight but flexible, enabling proof-based reputation systems, native mining mechanics, and permissioned updates.
-  **Realms:** First-come-first-served on-chain naming containers. Realms act like decentralized domains or foundational worlds. Once created through a Bitwork challenge, a Realm is permanently owned without any renewal fees, external tokens, or intermediaries. Realms serve as the root containers for organizing applications, games, identity systems, and more.

Why This Matters

Atomicals turns Bitcoin from a passive store of value into an active substrate for digital creation. Every object minted through the protocol inherits Bitcoin's security guarantees, timestamping, and immutability. Yet thanks to Containers, Realms, and Bitwork, these objects are not static. They can grow, change, and react based on structured rules and evolving states.

- 🎮 Level up NFTs as players grow through chain updates
- 🛡️ Gated minting through whitelist or proof of work secured at the protocol level
- 🛠️ Crafting systems and verifiable item creation performed fully on-chain

By blending expressive metadata with minimal but powerful embedded logic, Atomicals enables a new wave of Bitcoin-native apps, games, identities, and assets, all without introducing a new blockchain, validator set, or token.

However, despite its flexibility, the Atomicals Protocol remains constrained by the base limitations of Bitcoin Layer 1. These boundaries led directly to the exploration of more powerful approaches, such as the proposed Atomicals Virtual Machine (AVM).

Proposed AVM

The Atomicals Virtual Machine (AVM) was originally proposed as a solution to bring smart contract functionality to Bitcoin, not by altering Bitcoin itself, but by simulating a Turing-complete virtual machine in a sandboxed environment run by indexers.

The idea was powerful: treat Bitcoin like a global immutable database, store compact scripts directly on-chain, and use external runtimes to execute that logic deterministically. AVM aimed to bridge the gap between Bitcoin's minimalism and the expressive needs of modern apps and games, without introducing a new blockchain, token, or validator set.

The architecture was elegant: smart contract scripts would live inside Atomicals, and AVM-compatible indexers would execute them using a restricted but powerful opcode set. This set was inspired by original Bitcoin Script but expanded to include programmable behaviors such as state transitions, validations, and simple proofs of work.

While the AVM significantly expands the expressive possibilities of Bitcoin through extended opcodes and structured scripting, it does not enable fully general-purpose logic. Execution remains predictably bounded to maintain trust-minimized, verifiable behavior.

Core Vision of AVM

- 🛠️ State Machine Logic: Developers define creation, mutation, and transfer rules for digital assets.
- 🌱 Off-Chain Deterministic Execution: Sandboxed runtimes simulate results and optionally commit updates back to Bitcoin.

- 📖 **Turing-Complete-by-Design:** AVM embraces the idea that Bitcoin Script (via loop unrolling and PDA-style execution) can achieve flexible logic without loops, randomness, or unbounded execution.
- 🧠 **State Hash Synchronization:** Each execution produces a state hash that can be verified across peers, ensuring consistency without centralized control.

In this model, overlay indexers and apps do not need to trust one another. They simply run the same script, verify the resulting state hash, and compare outcomes. No validators. No staking. Just 'don't trust, verify.'

🔑 What AVM Was Meant to Unlock

- 🎮 **Crafting and Game Mechanics:** Players combine items in a container, submit a script with a recipe, and the AVM validates and mints the result, such as forging a new sword in an on-chain RPG.
- 🔒 **Secure Token Transfers with Conditions:** Scripts can check balances, cooldown timers, and permission flags, only allowing transfers when all conditions are met.
- ⚙️ **Upgradeable NFTs:** Avatars or collectibles evolve by gaining traits, items, or visual changes governed by AVM rules.
- 🏛️ **Governance Systems:** Proposal votes are validated by AVM logic, with results written permanently to an Atomical.
- ⌚ **Scripted State Machines:** Objects whose behavior evolves over time, such as a vault that unlocks only after on-chain conditions are satisfied.

🌀 Current Status: AVM in Limbo

Despite the promise, the original AVM proposal has remained unfinished since its author stepped away. While the whitepaper outlines a strong vision and a technically sound opcode model, no official implementation was completed or maintained by the original team.

As of today, AVM remains in community limbo: a brilliant foundation without a finished engine to bring it fully to life. However, the ideas within it remain foundational, and they continue to inspire the design goals of the Atomicals Core Engine (ACE).

AVM in the Atomicals Core Engine








The Atomicals Core Engine (ACE) includes a fully integrated AVM runtime, written in Rust and embedded inside the backend of the Tauri desktop application. This interpreter forms the

execution heart of AVM-based applications, enabling deterministic, sandboxed logic that mirrors the original design proposed in the AVM whitepaper.

At its core, the AVM is a two-stack execution engine inspired by Bitcoin Script but extended with additional opcodes to support programmable state transitions. It is built for clarity, safety, and auditability, allowing users to simulate, test, and eventually commit logic that governs token transfers, NFT behaviors, on-chain interactions, and beyond.

This AVM engine runs in a fully isolated sandbox within ACE's backend. It has no access to the network, file system, or wallet by default. All inputs are passed in explicitly, and all outputs are returned as structured JSON. This architecture keeps the AVM tightly scoped and verifiable, while the surrounding ACE platform handles indexing, transaction building, and wallet interaction through a CLI-driven orchestration layer.

Key Properties of the Runtime

-  Rust-native for speed, safety, and zero-dependency execution
-  Dual-stack model (ScriptSig + ScriptPubKey), consistent with Bitcoin spend conditions
-  Extended opcode support
-  All operations are side-effect-free and deterministically auditable
-  Stateless by design — execution results are a pure function of inputs
-  Loop-free logic — supports "loop unrolling" for predictable execution time
-  Bounded execution — performance scales with script size, not external conditions

How It Works

The AVM receives fully formed input payloads from the ACE CLI. These payloads are run through the AVM and all results are returned to the ACE CLI for on-chain submission or sent to the Queue and Batching layer.

Why It's Sandboxed

By keeping the AVM logic fully local and sandboxed, ACE eliminates external attack surfaces, guarantees reproducibility, and ensures that every AVM action can be simulated exactly the same way by every node. This aligns directly with the original AVM whitepaper vision:

- No external dependencies
- No Bitcoin network calls
- No trust in third-party servers
- Execution relies purely on raw logic producing trusted, verifiable output

Every AVM run is re-executable and hash-verifiable. Two nodes executing the same script with the same state will always produce the same result, ensuring no forks, drift, or ambiguity.

Real-World Use Cases

- 🎮 **Game mechanics:** Crafting, combat resolution, leveling systems, and inventory logic that can be tested locally before committing on-chain.
- 📦 **NFT upgrades:** Evolution logic for digital collectibles, triggered by block height, item ownership, or off-chain proofs.
- 🗳️ **Voting systems:** Community governance with vote validation, tallying, and time-locked results — all enforced by AVM logic.
- 🗝️ **Access gates:** Require certain balances or NFTs to unlock in-game content, software modules, or mints.
- 🔄 **Token operations:** Controlled transfers, balance adjustments, or container mutations driven by verifiable rules, not backend servers.
- 🔗 **Taproot Merkle Proofs:** Compatible with Taproot-based Merkle proofs for scalable multi-asset minting.

🔗 Alignment With the AVM Proposal

The Rust-based AVM interpreter in ACE follows the original AVM whitepaper vision:

- Strict, auditable opcode model
- Deterministic, reproducible logic interpretation
- Bitcoin Script–like behavior with stateful extensions
- No new consensus or external execution layers needed
- Trust-minimized programmability entirely within Bitcoin-native infrastructure

Important Note - While we follow closely to the original AVM proposal we are separating the AVM from the Indexer. There are many reasons but the most important is separation of concerns and our application flow to ensure secure and predictable results. The Indexer is still the source of truth but operates in its own separate environment in the rust backend.

Even though the original AVM remains unfinished, ACE implements the vision aligning with the original intent while incorporating it into a fully formed engine while keeping it sandboxed so it can be used on its own or along with the other layers of the ACE program.

🔗 Enhancements

While not embedded directly within the AVM itself, ACE introduces an additional commit strategy via the Batching and Queue Layer. This system allows different strategies to lower fees, avoid dust limit issues, and enable longer-running scripts.

We expand on these features later, but briefly, the three submission types are:

- **Immediate Commit:** Sent immediately as its own transaction to the next Bitcoin block.
- **Deferred Commit:** Batching multiple transactions together until a batch limit is reached, reducing fees.
- **Progressive Commit:** Batching with periodic on-chain writes for scripts that require interval updates.

The Batching and Queue Layer allows ACE to avoid key base-layer limitations, unlocking broader use cases for the Atomicals Protocol.

Layer 3: Compiler & Logic System

The Layer 3 logic engine powers expressive, high-level applications inside the Atomicals Core Engine (ACE), from multiplayer games and governance tools to advanced NFTs and dynamic app experiences. It allows developers to write in languages like Python or JavaScript, simulate complex logic flows locally, and selectively commit trusted results to the AVM.

Layer 3 brings together developer freedom and cryptographic trust. Logic runs off-chain but remains tightly verifiable: each script is hashed, pinned, and validated before any interaction with the AVM is allowed.






While apps can be developed and run externally, any app connecting to ACE must pass a strict hash verification through the API. This ensures that the code executing in Layer 3 matches exactly across all nodes. If an app fails hash validation or attempts to modify logic after verification, it is automatically rejected by the system.

Rust-Backed and Fully Sandboxed

Layer 3 operates entirely inside the Rust backend of the ACE application. The Dioxus frontend or headless version has no direct write access to layer 3 or the runtime. All reads and writes flow through a secure, auditable CLI interface, preventing any code from injecting, tampering with, or bypassing trusted execution paths.

Every piece of logic, from simple leaderboard tracking to complex crafting recipes, is sandboxed, inspectable, and locally reproducible by design.

Core Responsibilities

-  Execute logic in local sandboxes written in preferred code (python, JS, etc.)
-  Extract @avm-tagged regions and compile into AVM bytecode
-  Sync state across nodes using a hash-verifiable storage layer
-  Reject invalid or tampered code via runtime hash checks and CLI/API verification
-  Publish manifests for applications, bundling logic, UI elements, and state schemas

Execution Flow

1. Developers write application logic in Python, JavaScript, or Rust.
2. ACE hashes the entire logic bundle, including optional assets.
3. Apps connect to ACE, presenting their current code hash / bundle, which syncs across nodes.
4. The CLI validates the hash against a trusted manifest or atomical container.
5. Any code sections marked with `@avm` are compiled into AVM bytecode.
6. Logic executes locally, and results are either committed on-chain through the AVM or shared via the GUN.js layer.

This layered model ensures that every interaction remains expressive, modular, and trust-minimized, with no centralized API, server, or external gateway mediating execution.

Optional Use of Layer 3

While Layer 3 provides the preferred environment for expressive, multi-file, multi-state applications, it is not required for AVM execution. Developers also have the option to:

- Write raw AVM-compatible logic directly in Bitcoin Script
- Store logic scripts inside Atomicals
- Submit transactions directly via the CLI for immediate execution

This flexibility supports both minimalistic applications and complex, multi-layered apps. No matter the source, all AVM logic undergoes the same deterministic sandboxing and hash verification processes.

Layer 3: Logic Storage

To maintain clear separation of responsibilities, the Atomicals Core Engine (**ACE**) includes a dedicated Layer 3 Storage subsystem, a SQLite-backed module embedded within the Tauri backend, fully isolated from the indexer and frontend layers.

This subsystem stores all verified Layer 3 code bundles, their associated hashes, manifests, and certification tags. It acts as the trusted internal record of approved application logic for the node, ensuring that no unverified or modified code can be executed.

While Layer 3 Storage is local, it synchronizes across nodes using a lightweight gossip protocol, allowing the network to propagate trusted application hashes and ensure consistent versioning without requiring centralized control.

Architecture

- Runs as a standalone microservice written in Rust
- Access controlled exclusively through the CLI/API interface
- Subject to the same trust and verification rules as the AVM and wallet subsystems
- Synchronizes manifest and hash updates peer-to-peer via gossip protocol

Responsibilities

- Store all compiled and validated application logic bundles
- Track versions and maintain hash-pinned manifests
- Provide trusted code references during API hash validation before execution

Commitments and Onboarding

All new applications joining ACE must publish their bundle and hash on-chain as part of their onboarding process.

This mandatory on-chain publication creates a verifiable, timestamped anchor for application versions, enabling full auditability and version control across the network. The Atomicals Core Engine provides simple tools to automate this process during app onboarding.

By isolating logic storage from the indexer and frontend, ACE preserves strict architectural boundaries and maximizes trust in execution. No code reaches the AVM runtime without first being validated against this secured storage layer.

GUN.js: Layer 3's Shared State Layer



All state shared across apps and nodes such as game data, inventories, identity records, or synced UIs is written to a secure GUN.js instance embedded inside the Rust backend. The frontend cannot write to this layer directly.



Each node stores only the apps and games it chooses to support. However, to ensure trust and data resilience, every app or game must be backed by a minimum of five volunteer nodes, randomly selected by the system from the pool of participants. This guarantees that even if individual nodes leave or fail, the shared state remains available and verifiable across the network.

To reinforce auditability, a hash snapshot of each GUN.js dataset is periodically committed on-chain. This allows users and nodes to verify that off-chain state evolution matches publicly anchored checkpoints over time.

All critical state is hash-verified. Any data not declared, validated, and hash-committed is ignored by the protocol and treated as untrusted.

Real-World Use Cases





-  Multiplayer games with synced logic and real-time game state via GUN.js
-  Crafting and upgrade systems simulated off-chain, then committed on-chain

-  Dynamic NFTs that evolve traits, unlock new features, or level up over time
-  DAO governance where proposals are discussed off-chain and final tallies are committed via @avm logic

Code Integrity

Even though Layer 3 is flexible, it remains fully traceable and inspectable. If a game attempts to manipulate outcomes before the @avm tag fires the action will be rejected. For example, always letting one side win in a strategy game would be detectable and rejectable.

Every app or logic bundle follows a strict lifecycle:

-  Hashed upon creation and bundle stored in Layer 3
-  Optionally published inside an Atomical container
-  Validated at runtime during interaction
-  Rejectable by hash and bundle check if it does not match the expected known version

Community-driven audits, AI-powered static analysis, and manual inspection workflows are integral parts of the broader ecosystem's trust model.

Security and Trust Model





Upon startup, every instance of the Atomicals Core Engine (**ACE**) performs a full-environment hash check and verifies its hash against the latest certified snapshot stored inside an Atomical container on Bitcoin. If the hashes match, the node is considered trust-aligned with the published build. A second check against other nodes in the network through a lightweight gossip protocol ensures all nodes are aligned and in sync.

This ensures that every node, before interacting or submitting any commits is valid and up to date. ACE enables a trustless environment with Bitcoin providing the final anchor of verification.

The Atomicals Core Engine is built on a simple but powerful principle: trust is proven, not assumed. Every action, every update, and every app interaction must be independently verifiable and resilient to tampering.

There are no untrusted APIs, no hidden backdoors, and no centralized execution. Every layer enforces verification through local validation, Bitcoin anchoring, and transparent code and data flows.

Core Security Principles

-  All code is hashed and validated before use
-  All data affecting gameplay or protocol behavior must be cryptographically verifiable
-  All logic interacting with Atomicals must pass through deterministic AVM execution
-  Every node independently enforces these rules with no need for external consensus

Flow-Based Trust Model

In ACE, trust is rooted in how data and logic move through the system:

- Inputs must pass local validation
- Scripts must match registered logic hashes
- State transitions must occur through controlled AVM or Layer 3 pathways
- No critical action proceeds without traceable, deterministic verification

It is not about trusting individual nodes. It is about trusting the verifiable flow of computation and state. Everything related to Atomicals eventually flows down to the deterministic AVM and if determined valid commits its results on-chain.

Local First, Federated Second

The Atomicals Core Engine treats your local node as the primary authority whenever possible.

- You can validate application logic locally without quorum votes.
- You can audit state transitions offline through replayable commit histories.
- You can verify NFTs and assets independently without relying on peers.

Federation and peer syncing exist to enhance trust, live interaction, not to replace local sovereignty.

Defending Against Malicious Code

ACE cannot prevent someone from creating bad code. But it ensures no one is forced to trust it:

- Malicious scripts could exist but are public, inspectable, and don't assume any community trust.
- Mismatched App versions or changes without updating ACE will be rejected by CLI.
- Certificate systems and runtime inspections immediately expose unauthorized changes.

Layer-by-Layer Trust Enforcement

Layer	Trust Enforcement
Frontend	Cannot write to backend or GUN.js and only use CLI pipeline
GUN.js	Only Rust backend writes, all hash-tracked
CLI	Mediates all logic and wallet actions through verified payloads
Layer 3	Scripts are hash-pinned, peer-synced, and version-tracked
AVM	Stateless, deterministic, accepts only valid @avm bytecode
App Certificates	Users verify app code against on-chain manifests

Encrypted Developer Data Support

ACE supports encryption of developer-controlled data inside GUN.js. Sensitive elements, such as private inventories, gameplay secrets, player info, or confidential proofs can be encrypted before being written to the GUN.js mesh. Nodes hosting this data cannot decrypt it unless explicitly authorized, ensuring that hosting remains trustless and confidential. Logic, however, is never allowed to be hidden.

Certificate Verification System

Applications that run externally that wish to be recognized as "certified" by the Atomicals Core Engine can publish a manifest and logic hash on-chain. When users open these apps, ACE displays:

- A "Verified by Atomicals Core Engine" badge
- A unique certificate ID
- A real-time local validation that checks the installed code against the known certified hash

Even if a malicious actor attempts to spoof the user interface, they cannot fake the certified logic hash. Real trust is cryptographically verifiable, not merely visual.

The certificate is an additional tool but does not change the security and trust model established above.

API Trust Registry

ACE permits integration with external APIs only through an internal Trust Registry. Only pre-approved trusted APIs may interact with applications. Any future API additions must undergo a transparent, consensus-driven review before being accepted.

AVM as Final Arbiter

Any operation that mutates Atomicals, whether token transfers, NFT evolutions, or state change must pass through AVM validation.

- No hidden state mutations
- No backend shortcuts are available
- No side-channel attacks bypassing rules are possible

The AVM validates deterministically and rejects invalid operations at execution time. Simulation is free, but commitment requires truth.

Failure Recovery and Replayability

ACE is built for operational resilience:

- Transaction failures do not impact state as commits occur only after confirmation.
- Node drops are automatically retried and recovered.
- Full storage syncing and commit replay guarantee lossless recovery even after disruptions.
- Every execution is stateless, auditable, and deterministically reproducible.

Summary: What Makes ACE Secure

- Every component validated by hash
- Developer-controlled encryption for sensitive state
- Certified app onboarding and runtime protections
- Peer-to-peer resiliency without blind trust

Security is not granted by promise. It is guaranteed by cryptography, transparency, and Bitcoin anchoring.

The Atomicals Core Engine (**ACE**) includes a fully integrated indexer written in Rust utilizing a SQLite embedded and isolated inside the Tauri backend. The indexer is the backbone of the ACE program and powers all internal components. Its main function is to provide the data by indexing and tracking Atomicals protocol activity and maintaining a clean, local state representation of containers, ARC20 balances, and AVM logic.

Unlike generalized blockchain indexers, the ACE program indexer is purpose-built for Atomicals and AVM-native applications. It parses and stores only the data relevant to Atomicals activity without indexing the full Bitcoin chain.

Node Requirements and Bootstrap Process







The ACE Program does not require users to run a full validating Bitcoin Core node.

On first installation:

- The program bootstraps its local Atomicals state by syncing from trusted peer nodes or optional checkpoint snapshots.
- After bootstrapping, it connects to a trusted Bitcoin RPC source to monitor and parse incoming blocks.

The indexer parses only Atomicals-related envelopes, avoiding full chain validation while maintaining auditability through optional on-demand verification.

What It Tracks

-  Mints, updates, and transfers of Atomicals
-  AVM container state, including rule changes, script references, and tagged updates
-  ARC20 balances across tracked wallets
-  NFT and realm ownership
-  Executable state for AVM runtime input (container metadata, container rules, ownership changes)
-  In short, it tracks everything related to the Atomical Protocol

How It Works

After connecting to a Bitcoin RPC provider, the indexer reads incoming blocks in real time.

It scans for Atomical inscriptions, applies Atomicals Protocol parsing rules, and updates a lightweight local SQLite database with verified object states.

It runs continuously in the background with:





- A built-in Bitcoin reorg handler for chain reorganization resilience
- A dynamic block height tracker to ensure up-to-date synchronization
- Automatic retry logic for missing events or delayed blocks

There is no separate installation required. The indexer is embedded directly within ACE and runs automatically upon startup.

Indexer Modules

- **Block Reader:** Connects to Bitcoin RPC, fetches new blocks and transactions
- **Parser:** Detects Atomicals and AVM inscriptions, extracts structured metadata
- **State Engine:** Applies state transitions for Atomicals (mint, update, transfer)
- **SQLite Database:** Stores canonical object state, container metadata, ARC20 balances
- **Internal API:** Feeds validated state to CLI, AVM runtime, wallet, and GUI
- **Event Triggers:** Emits hooks for CLI daemons, logic runners, and in-game scripts

Example Use Cases

-  AVM runtime queries container state before executing logic
-  Wallet interface displays balances and NFT metadata pulled from the local index
-  Game runners listen for on-chain updates and resolve moves or minting actions
-  CLI tools validate logic state before submitting commits to Bitcoin

Designed for Simplicity and Speed

- Embedded within the Atomicals Core Engine with no external setup required
- Optimized specifically for Atomicals and AVM targets ensuring minimal bloat
- Fully inspectable and verifiable with direct interaction with the CLI
- Secure by design with all data derived directly from Bitcoin RPC and independently verifiable

Before a node submits any on chain commit, it verifies that its local indexer is fully up to date. This check prevents conflicting or duplicate results that could cause transaction failure.

Important Note- In the proposed AVM and current infrastructure the indexer acts as a gateway to commit on chain while ours does this in a different way. ACE checks twice: once before committing, validating the action against current state, and then submits the transaction knowing the action is valid. After confirmation, the indexer writes upon completion. This ensures a more secure and predictable outcome.

Whether you are building an NFT minter, a decentralized game, or a logic-driven application, the ACE Program indexer delivers real-time, trusted, AVM-ready state without the need for any external indexing infrastructure.

CLI System

The Command Line Interface (**CLI**) serves as the primary interface and orchestration layer between all critical systems inside the Atomicals Core Engine (**ACE**). Built entirely in Rust and embedded into the Tauri backend, the CLI acts as the control surface for developers, automation tools, and everyday users. It enables seamless interaction with the indexer, AVM runtime, Layer 3, gun.js, APIs, and wallet.

Whether you are a developer running logic scripts, a gamer interacting with app state, or simply a user syncing NFTs and balances, the CLI provides a simple, auditable interface into the ACE program. Every operation, from logic execution to on-chain commits, flows through the CLI, ensuring clarity, reproducibility, and complete security.





What the CLI Talks To

Once logic is validated and stored either on-chain, in the indexer, or in Layer 3, the CLI can reference it by hash, container ID, or app tag enabling fast local reuse and reducing recompile load.

The CLI coordinates:

- Interaction with the embedded indexer for reading state
- Execution calls into the AVM runtime
- Wallet management and transaction signing
- GUN.js local syncing and secure data operations

Core Use Cases

-  AVM runners that fetch container state, simulate logic, and commit validated state updates
-  Scheduled logic daemons that execute rules once per block or on specific triggers
-  Developers testing, debugging, and interacting with Layer 3 scripts
-  Games and apps that pass logic inputs into the CLI to modify on-chain state or synchronized data
- Individual users submitting transactions, minting assets, or verifying certificates without needing complex setups




Internals and Routing

Internally, the CLI is structured as a modular command router with secure paths to each subsystem. All interactions are mediated through the Rust backend with no direct access granted to frontend code or external scripts.

The CLI does not just unify subsystems but also protects them. All state writes, logic executions, and storage updates must pass through this secured command flow.

Old versions of logic or effects can be archived or pruned automatically, keeping the system clean and minimizing data load.

Security and Isolation





-  **Full sandboxing:** no raw access to wallets or internal storage from outside the CLI
-  **Wallet signing:** controlled exclusively through secure CLI commands and never exposed directly
-  **Bitwork-ready:** CLI includes full support for submitting proof-of-work and validating gate logic

Automation-First Design

The CLI is not just a developer convenience but the backbone of autonomous operation across the ACE program.

CLI runners can schedule logic pulls and AVM executions directly from the local logic store, enabling autonomous games, proposal systems, real-time sync daemons, and more.

Scheduled or triggered workflows can:

-  Run coordination between the layers for game logic
-  React dynamically to AVM state changes or external triggers
-  Auto-submit container state updates after off-chain simulations if outcome is valid
-  Simulate new outcomes based on Layer 3 logic stores synced state before committing anything to Bitcoin

Whether you are organizing a DAO, running a decentralized voting system, or playing a dynamic game tied to Bitcoin, the CLI is the trusted bridge that makes it all work seamlessly and efficiently with full automation.

Batching Layer: AVM Commit Middleware

Not all AVM outputs need to be committed immediately. To reduce fees and increase flexibility, the Atomicals Core Engine (ACE) includes an optional Batching Layer, a lightweight Rust-based microservice, that queues and groups AVM results for later commit to the Bitcoin blockchain.

Built in Rust with Embedded Storage

The batching system is implemented as a standalone microservice written in Rust.

It uses a local SQLite database to store:

- Queued AVM effects (outputs)
- Metadata such as source app, timestamp, and commit priority
- Scheduled commit rules and fee thresholds
- Commits in progress or pending retry

Individual nodes can submit their local transactions to the batching layer, where they are processed alongside transactions from other nodes through an automated flow, reducing fees and improving system efficiency.

This architecture ensures fast, reliable performance while staying true to ACE's local-first, audit-friendly design principles.

How It Fits Into the System

The standard AVM commit flow becomes:

GUI → CLI → AVM → CLI →  Batching Layer → Wallet → Bitcoin

Developers and power users can opt into batching by routing logic outputs through the queue via CLI commands, or automation tools. Applications can dynamically decide whether to commit immediately or defer through batching based on priority, fees, or system conditions.

Incentives and Fee Handling

Committer nodes are rewarded with a fee share when they successfully broadcast batched transactions. Any excess fees collected beyond estimated minimums are routed to a public Community Development Wallet to support open-source tooling, audits, and grants across the Atomicals ecosystem.

Trust and Auditability

- No logic is executed inside the batching layer and only previously validated AVM effects are stored and grouped
- Every queued item must originate from a valid AVM run
- Final commits must pass wallet signing and CLI validation before broadcast
- The batching queue itself is hash-tracked locally and can optionally be pinned to Layer 3 for full audit trails
- Commits are signed only after full validation, and all outputs remain fully traceable and verifiable

By concentrating only verified effects inside a secured, inspectable queue, the Batching Layer enhances flexibility without introducing new trust assumptions.

The Batching Layer empowers applications and users to optimize transaction efficiency without sacrificing the verifiability, auditability, and Bitcoin-native integrity that the Atomicals Core Engine demands.






Wallet System

The Atomicals Core Engine (ACE) includes a fully integrated, Bitcoin-native Atomicals Protocol wallet built directly into its Rust backend.

This wallet uses Taproot (**P2TR**) addresses by default and operates fully locally for maximum security, auditability, and user control.

An optional secure bridge allows external applications to request limited access to the wallet through a trusted connection eliminating the need for browser extensions while preserving full user custody and control over private keys.

Core Design Principles

-  **Local-first:** Keys & transaction data are stored, encrypted, and verified locally
-  **Secure by default:** HD key storage with encrypted seed vaults & passphrase protection
-  **Atomicals-aware:** Fully understands NFTs, Realms, ARC20s, and AVM Containers
-  **AVM-integrated:** Seamlessly supports simulation, signing, and logic commits
-  **Ready to Go:** Every user installs ACE with a personal wallet already initialized

What It Can Do

- Create and manage Taproot-based Bitcoin addresses and UTXOs
- Sweep small UTXO fragments automatically for efficient transaction creation
- Sign AVM logic commits and container updates with full Bitcoin compatibility
- Send and receive ARC20 tokens, NFTs, and Realm-based objects
- Simulate complex transaction flows locally before broadcasting

Developers can leverage the ACE wallet in applications to manage community wallets, DAO treasuries, game prize pools, and other production-grade assets with full Bitcoin-native custody.





Workflow Integration

The wallet is deeply integrated with the core systems of the ACE Program:

- The Indexer continuously detects current UTXOs, balances, and Atomical holdings
- The Wallet Module builds valid Bitcoin transactions based on Atomicals and AVM state
- The AVM Runtime can simulate transaction outcomes before commit for extra safety
- Finalized transactions are signed securely and then broadcast for on-chain commitment

All operations happen locally without relying on third-party APIs, browser plugins, or external custodians.

Smart Safeguards

-  Prevents accidental burns or splitting of NFTs by verifying transaction outputs
-  Warns about dust outputs and invalid transaction formats before signing
-  Handles multi-asset UTXO merges, ARC20 sends, Realms, and logic commits safely
-  Supports automated UTXO sweeps to optimize wallet performance and fee efficiency

Optional Developer Tools

Advanced users and developers can extend wallet functionality through:

- Exporting and importing mnemonic seeds or encrypted vaults
- Building and signing PSBTs manually for use with external signing devices
- Triggering wallet actions programmatically via the CLI for integration into automation pipelines or scripted workflows

External App Connectivity: No Browser Extensions Needed

ACE does not use traditional browser extensions or inject scripts into webpages. Instead, external applications or websites can request a trusted bridge connection to the local wallet, which must be explicitly approved by the user.

This model preserves privacy, eliminates third-party risk, and ensures all wallet actions remain verifiable and local-first, with no exposure of private keys to web-facing environments.

The Atomicals Core Engine wallet is more than just a key manager. It is a fully integrated, Bitcoin-native execution environment for managing Atomicals, running AVM logic, interacting with digital assets, and building decentralized applications. All without ever giving up sovereignty over your keys.






GUN.js Data Layer

While the Atomicals Protocol secures permanent on-chain state, not every piece of application data needs to live on Bitcoin. For off-chain data that still requires trust and verifiability, the Atomicals Core Engine (ACE) includes an optional, decentralized data layer built on GUN.js.

This shared layer supports applications that need synchronized state without polluting the blockchain such as game stats, maps, inventories, user preferences, or temporary collaborative sessions.

What It's For

GUN.js is used for data that does not belong on-chain but still matters for application logic:

-  Game state (e.g., position tracking, combat logs)
-  Player inventories or session data
-  Drafts of proposals, queued moves, or async coordination
-  World-building metadata or in-app structures
-  User profiles, client-side settings, or preferences

If it does not need to be on-chain but still needs verifiable trust, it belongs in GUN.js.

Trust Model

Even though GUN.js data is off-chain, it is not off-trust.

Every app that reads from GUN.js must validate through a strict hash-checking system:

- Applications declare a code hash or state manifest for shared data
- Any incoming data must match an expected hash or be linked to an approved container
- Mismatches are rejected automatically
- GUN.js works as a peer sync database which makes tampering extremely difficult

This ensures that access remains flexible but verifiable across nodes.

How It Works

GUN.js operates entirely within the Rust backend.

- The frontend interface cannot access GUN.js directly
- All reads and writes flow through the CLI to maintain isolation and full auditability
- Nodes selectively sync only the apps & data they choose to support
- Fee structure encourages nodes to participate in hosting data

If you support a project, you can help host its real-time state by opting into synchronization.

Integration with Layer 3

GUN.js works seamlessly with Layer 3 apps and logic flows:

A game might store temporary actions or session state in GUN.js

- Players' clients sync this shared state through local nodes
- When it is time to commit a permanent outcome, the final state hash is passed to the AVM for validation and on-chain commit.
- If the synced GUN.js state hash does not match the expected logic, the AVM rejects the commitment

This architecture provides developers with flexibility during gameplay or collaboration, without sacrificing trust in final outcomes.

Optional Participation

Every ACE node has complete control over GUN.js participation:

- You can choose to sync and help host your favorite apps or games
- You can ignore experimental apps or low-trust data if desired
- You can operate fully air-gapped or skip GUN.js synchronization entirely

There is no forced replication requirement.

Participation is always voluntary, but all synced data remains hash-verified and auditable.

GUN.js Messaging and Optional Layers

The GUN.js system inside the Atomicals Core Engine also supports lightweight messaging and optional application-specific layers.

Users and nodes can opt in to host shared chat rooms, coordination channels, or additional GUN.js data spaces alongside the core Atomicals mesh.

A decentralized messaging system will be deployed using the same GUN.js framework allowing nodes to communicate, organize, and participate in project governance without relying on external servers or third-party infrastructure.

All participation remains voluntary and hash-verified, maintaining the trust and flexibility that define the ACE Program's local-first model.

GUI System and Tauri App

The Atomicals Core Engine (**ACE**) is delivered as a fully native desktop application, built using Tauri, a modern application framework that combines a Rust backend with a secure, efficient Dioxus frontend written in Rust.

This structure gives users full control and transparency while maintaining strong separation between presentation and execution logic.

Unlike browser-based tools or Electron apps, the ACE program is a true native desktop application: lightweight, fast, auditable, and self-contained. There are no browser extensions, no background daemons, and no hidden dependencies. It's ready out of the box after downloading.


What It's Built With









- **Rust backend:** handles all trusted operations, including AVM simulation, wallet signing, indexer queries, and file access
- **Dioxus Frontend:** provides a clean, modular user interface without direct access sensitive backend operations
- **Tauri container:** secures communication between frontend and backend while maintaining a minimal resource footprint

Design Principles

- **Backend-only trust:** All critical logic is handled inside the Rust backend
- **Frontend isolation:** No direct access to wallets, GUN.js state, or AVM runtime
- **Headless compatibility:** Systems can run fully automated without the GUI if desired
- **Auditability:** Every GUI action maps to a visible backend command traceable by users or scripts






What You Can Do from the GUI

-  Write and edit AVM logic scripts using embedded editors

-  Simulate AVM container logic and inspect outcomes before committing
-  Browse container, NFT, and Realm metadata directly from your local indexer
-  Preview AVM commits and validate transaction payloads before submission
-  Manage your wallet: sign transactions, send ARC20 tokens, mint Atomicals, and transfer NFTs
-  View ARC20 token balances, Atomicals holdings, and transaction histories
-  Test your apps and logic flows against a built-in local test-net environment
-  Participate in governance votes and protocol update proposals
-  Join community chat channels integrated into the app for collaboration and coordination

Explorer Module (Built-In)





The GUI includes a native Explorer tab that allows users to inspect all data tracked by the Atomicals Core Engine indexer without relying on third-party APIs.

-  Browse Atomicals including Realms, Containers, NFTs, and ARC20 tokens
-  View full metadata, state transitions, and commit histories
-  Inspect AVM containers to see current logic and rules
-  Filter objects by ownership, tag, type, or operational status
-  Drill into any atomical or transaction to see how it was constructed at the protocol level

All explorer data is pulled directly from your local node, ensuring a fully Bitcoin-native, verifiable view of your digital assets.

Why Tauri?

Choosing Tauri over Electron or traditional browser frameworks provides:

-  **Native system-level security:** no embedded Node.js runtime
-  **Compact binaries:** 10–20MB installations, not 200MB+ bloated packages
-  **Lightning-fast performance:** near-instant launch, minimal RAM usage
-  **Cross-platform flexibility:** seamless support for Linux, macOS, and Windows

Tauri maximizes efficiency, security, and trust, while still offering full flexibility for future app development.

A True Local-First App

The ACE Program is not a dashboard, browser overlay, or thin wrapper.

It is a full stack application where:

- Wallet keys are generated, stored, and managed locally with export and recovery options

- Layer 3 enables complex logic while ensuring transpiled outputs remain fully deterministic
- AVM logic is simulated locally in a deterministic sandbox, ensuring safe, verifiable results
- All Atomicals data is pulled from your local indexer with no reliance on external APIs, third-party RPC relays, or custodial servers
- There are no cloud dependencies and no 'web3' middleware layers

Every action stays fully auditable, verifiable, and under user control from start to finish.

✂ Headless Mode for Server and Automation Use

The Atomicals Core Engine is designed to run either with a full GUI or in headless mode.

In headless mode, the Rust backend, including the AVM runtime, wallet system, indexer, and CLI runs independently without launching the graphical interface.

This allows developers, server operators, and game runners to deploy ACE in environments like AWS or private data centers, using automation scripts and CLI tools directly against the core engine.

For users who prefer a graphical interface while running a headless backend, ACE also supports a secure local bridge using Tauri's native localhost communication. This enables remote GUI access to manage ACE nodes without exposing internal operations or compromising security.

Whether you are building new decentralized applications, simulating AVM logic flows, minting assets, or simply exploring Bitcoin-native digital objects, the Atomicals Core Engine GUI provides a secure, expressive interface that is fully aligned with the local-first, open, trust-minimized principles that Bitcoin itself demands.

Design Philosophy & System Summary

The Atomicals Core Engine (**ACE**) is an execution environment for the Bitcoin-native Atomical Protocol built from the ground up to be transparent, developer and user friendly, and secure. Every component reinforces a single core principle:








Trust must be proven, verifiable, local, and easy to understand.

Rather than reinventing blockchains or bolting trust onto third-party systems, the ACE program embraces the simplicity and permanence of Bitcoin while delivering powerful, modern development tools.

Design Pillars

- **Isolation:** Every part of the system is built to work in isolation while being connected by the CLI
- **Trust-Minimized UX:** All logic is inspectable, deterministic, and provable from AVM execution to CLI commits.
- **Local-First:** The entire program runs on your device. There are no hidden networks, browser extensions, or custodial APIs.
- **Bitcoin-Native:** Every operation ultimately resolves to the Bitcoin blockchain. No new chains, no wrapped assets, no abstractions.
- **Developer-Friendly:** Whether you are minting tokens or NFTs, building games, DAOs, or programs, the system gives you the tools you need.

What's Included

-  **AVM Runtime:** deterministic, sandboxed script execution for logic-based apps
-  **Layer 3 Compiler:** tag, transpile, and commit trusted logic to the AVM
-  **Indexer:** local, SQLite-backed parser for Atomicals protocol activity
-  **CLI System:** automation-ready interface layer for scripting, committing, and simulating
-  **Wallet:** Atomicals-aware, secure by default, and bundled with every installation
-  **GUN.js Data Layer:** optional, hash-verified peer state for non-chain shared data
-  **GUI App (Tauri):** lightweight, cross-platform interface with built-in explorer and management tools

Developer Test Mode

The Atomicals Core Engine (ACE) includes a full Developer Test Mode, allowing developers to simulate AVM executions, batching, and commit flows against a Bitcoin regtest chain.

Test Mode mirrors the complete production environment: the same AVM runtime, the same CLI commands, and the same commit validation paths are used without broadcasting transactions to the Bitcoin mainnet.

Developers can simulate block heights, cooldown timers, mint triggers, batching thresholds, and multi-node state transitions, all from a fully local environment.

Switching between production and test mode is seamless. A simple backend toggle redirects all commit operations to a test chain without modifying user applications or logic scripts. This enables full end-to-end validation of decentralized applications before any mainnet deployment.

Community Governance

The Atomicals Core Engine (ACE) is fully open-source and community-extendable, but it follows a stability-first philosophy, like Bitcoin Core, prioritizing careful, consensus-driven improvements over frequent or disruptive forks.

Within ACE, a built-in consensus mechanism will govern major upgrades and changes to core systems. Any proposal that affects critical layers such as the AVM runtime, indexer structure, wallet model, or system trust assumptions must be approved by at least two-thirds of active nodes to be accepted.

This ensures that the Atomicals ecosystem evolves responsibly, with upgrades backed by real user and developer consensus rather than top-down control or casual fragmentation.

Developers are free to build, extend, and experiment without permission. Plug-ins, extensions, external apps, and experimental tools do not require governance votes as long as they do not alter core protocol layers.

This balance keeps the foundation secure while allowing creativity, innovation, and open development to thrive around it.

Open-Source Commitment & Development

The Atomicals Core Engine (ACE) will be released under a full open-source license upon its official production launch.

Status: Development is approximately 45% complete. Our small team is working daily to bring ACE to full realization. We currently estimate a beta release around August 1, 2025, or earlier. Our upcoming website will provide additional details on timelines, documentation, and updates as they become available.

Faction 2028 and Open-Source Expansion

As part of its broader vision, the sister project of ACE, Faction 2028 will reward participants for building plugins, extensions, and development upgrades for the Atomicals Core Engine.

Players and developers alike will be incentivized to expand the ecosystem, contributing directly to ACE's modular, decentralized future.

Some of the open-source projects already planned will demonstrate just how powerful this system can become by unlocking new possibilities for Bitcoin-native applications beyond what is possible today.

Features Planned for Full Activation

Several system-level features are planned for full activation following the Atomicals Core Engine (ACE) beta period. These are separate elements and not part of this litepaper but are in active development. Outlined below:

1. ***** Token and Liquidity System (Token Name Hidden):

A liquidity layer powered by real assets and logic will launch, allowing flexible trade settlements, deferred resolution, and liquidity at release to make sure developers have access to the full suite of tools they need to build.

2. Program-Wide Fee Sharing:

Once ACE is fully live, all participating nodes and contributors will share in program-generated fees through a transparent, decentralized system.

More detailed information about these features will be released several weeks before the beta launch.

3. AI Development & Integration

Since the beginning of development, we anticipated that AI would become foundational to the evolution of the Atomicals Protocol, the AVM, and our broader program architecture. ACE and Faction 2028 have been designed with the future of local-first AI systems in mind, prioritizing privacy, security, and user sovereignty.

We believe we are in a unique position to offer the community new ways to integrate open-source AI technologies, including local deployment of LLMs (Large Language Models), in a trust-minimized and forward-looking manner.

More details on our AI initiatives and roadmap will be released in upcoming updates.

Closing Note

The Atomicals Core Engine is not an experiment or a wrapper. It is a complete Atomicals Protocol execution stack, ready to power the next generation of decentralized applications and digital ownership.

The Atomicals Core Engine is not just an application. It is a new operating layer for Bitcoin.

It is a protocol-driven platform for building honest games, dynamic NFTs, autonomous containers, and community-owned logic.

If Bitcoin is the root of digital permanence, the Atomicals Core Engine is the substrate for digital agency.

You can simulate. You can commit. You can run your logic and trust in the logic flow of apps built around our engine.

This is what trust looks like.

This is what local-first means.

This is what building on Bitcoin should feel like.

References & Further Reading

The full whitepaper will include technical schemas, validation logic, metadata formats, and complete developer documentation. It will be released a couple weeks before beta launch.

Key References

Atomicals Protocol: <https://atomicals-community.github.io/atomicals-guide/>

AVM Whitepaper: <https://github.com/atomicals/avm-whitepaper/blob/main/avm.md>

Faction Litepaper

<https://github.com/faction2028/app/blob/main/litepaper.pdf>

We thank both the Atomicals and AVM contributors and communities for their early work, vision, and energy. We hope to live up to the high standards you've set.

The ACE and Faction 2028 Team

@faction2028 Only on X