# MANDATE

*Verifiable Policy Controls for Autonomous Financial Agents*

Product Overview · July 2026 · Working Local MVP

MANDATE is a policy enforcement layer that sits between an AI agent and a financial execution system. Before an agent can submit a financial action, MANDATE requires the agent to produce a cryptographic proof that the action complies with the user's predefined rules. Compliant actions proceed. Non-compliant actions are rejected — not by a warning, not by a prompt, but by a proof that cannot be forged.

# 1. Executive Summary

AI agents are increasingly capable of interacting directly with financial systems — placing orders, initiating payments, managing portfolio positions, and executing across protocols and platforms. This creates a real and immediate problem: how do you give an agent enough authority to act usefully, while ensuring it cannot act beyond the limits the user has set?

Today, most systems rely on prompt instructions, backend API permission sets, or human approvals to constrain agent behavior. These approaches are practical, but they share a common weakness — enforcement happens at a layer the agent can influence, that a third party cannot independently verify, or that requires trusting a centralized operator.

MANDATE addresses this gap. It is a verifiable policy layer placed between an AI agent and a financial execution rail. The agent does not simply assert that an action is allowed — it generates a zero-knowledge compliance proof demonstrating that the action satisfies the registered policy. If the proof cannot be produced, the order cannot be submitted.

The current working MVP integrates MANDATE with the MoonPay Agent through a published MCP server (@0xgks/mandate-mcp@0.1.0). During testing, a valid order (buy 250 units at 3,500 — notional 875,000) was successfully proven and submitted on-chain. An oversized order (10,000 units at 3,500 — notional 35,000,000) failed proof generation and was correctly rejected, with no transaction accepted and the server remaining fully available.

## At a Glance

| Area | Current MVP |
|---|---|
| Problem | Agents may receive overly broad execution access |
| Solution | Proof-based policy enforcement |
| Interface | MCP (Model Context Protocol) |
| Agent integration | MoonPay Agent |
| Proof system | Noir / UltraHonk (real on-chain verifier, bb-generated) |
| Execution venue | Local batch auction |
| Status | Working local MVP |
| npm package | @0xgks/mandate-mcp@0.1.0 |

# 2. The Problem: Agents Need Authority, but Not Unlimited Authority

## The core tension

For an AI agent to be useful in a financial context, it must be able to act. But the moment you give an agent execution access, you face a set of risks that are difficult to resolve with conventional tools.

Consider a practical scenario: a user wants an AI agent to trade ETH on their behalf, but only within a specific market, within an order size of 1,000,000 notional, with a defined position limit, and with a daily loss ceiling. The user provides these rules as instructions in the agent's prompt.

### Risk table

| Risk | Example | Without MANDATE |
|---|---|---|
| Prompt injection | Malicious input instructs agent to place an oversized order | Instruction may override original policy |
| Strategy bug | Agent miscalculates position and submits an oversized trade | No circuit-level constraint |
| Unauthorized market | Agent selects an asset not on the approved list | No enforceable whitelist |
| Policy manipulation | Local execution limits differ from registered policy | No on-chain commitment to verify against |
| Compromised runtime | Agent output is tampered with before submission | No cryptographic receipt of compliance |
| State inconsistency | Off-chain portfolio data is stale or falsified | No state-root verification |

> **Key insight**
> There is a difference between asking an agent not to exceed a limit and making it mathematically impossible for the agent to submit an action that does.

# 3. What MANDATE Is

MANDATE is a verifiable policy layer placed between an AI agent and a financial execution rail. It takes a user-defined set of rules (a mandate), registers a cryptographic fingerprint of those rules on-chain, and requires every execution action to be accompanied by a zero-knowledge proof that the action satisfies those rules.

| MANDATE is | MANDATE is not |
|---|---|
| Policy enforcement infrastructure | An AI model or trading strategy |
| An agent authorization layer | A general-purpose wallet |
| A proof generation and verification path | A replacement for MoonPay |
| An integration layer for execution systems | A centralized risk API |
| Verifiable delegated authority | A simple transaction simulator |
| A protection layer for autonomous agents | A standard multi-signature scheme |

**Core design principle**

The circuit proves the ACTION, not the model. It does not prove 'this is a trustworthy AI.' It proves: 'This order satisfies policy constraint set P against portfolio state S, where S is anchored on-chain.' The agent's strategy, its reasoning, and its confidence level are irrelevant to whether the proof succeeds.

**What MANDATE does not do**

MANDATE does not decide what the agent should do. It enforces what the agent is allowed to do. The agent retains full autonomy to propose any action — MANDATE ensures that non-compliant proposals cannot reach the execution layer.

# 4. The User Experience

From the user's perspective, MANDATE operates as a silent enforcer running behind the agent. The user defines the rules once. After that, the system enforces them automatically on every execution attempt.

## Five-step flow

| Step | Actor | Action |
|---|---|---|
| 1 | User | Defines the mandate — approved markets, size limits, position caps, loss limits |
| 2 | MANDATE | Registers a policy commitment on-chain; issues a session key to the agent |
| 3 | Agent | Proposes an order based on its own logic or user instruction |
| 4 | MANDATE | Checks the order against the policy and generates (or fails to generate) a proof |
| 5 | System | Only orders accompanied by a valid proof are submitted to the execution layer |

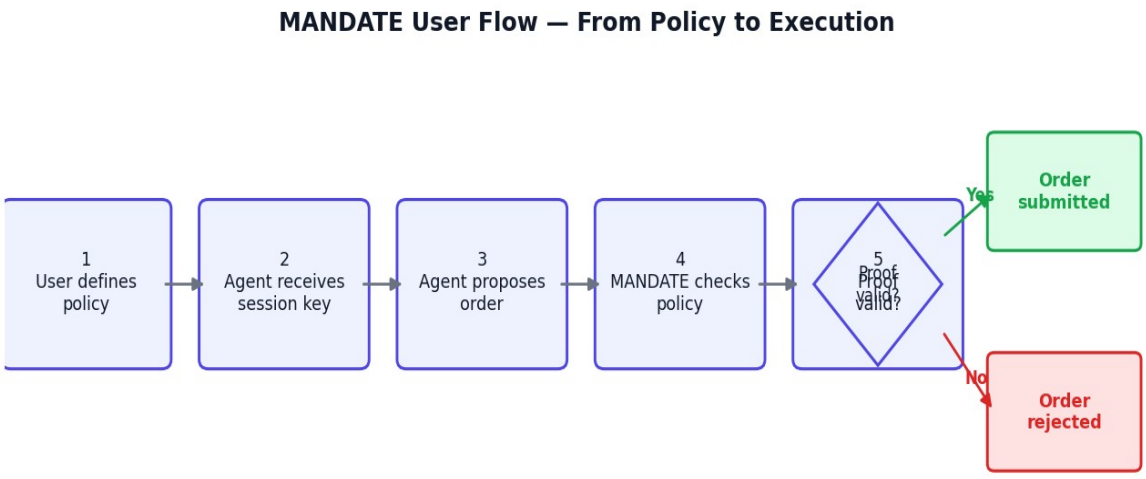


*Figure 1: High-level user flow. Execution only proceeds when a valid compliance proof is produced.*

> **Key takeaway**
> MANDATE allows the agent to act autonomously, but only inside a policy the agent cannot rewrite.

# 5. How the Current MVP Works

The working MVP connects the MoonPay Agent to a local MANDATE stack through a published MCP server. The following describes how each component fits together.

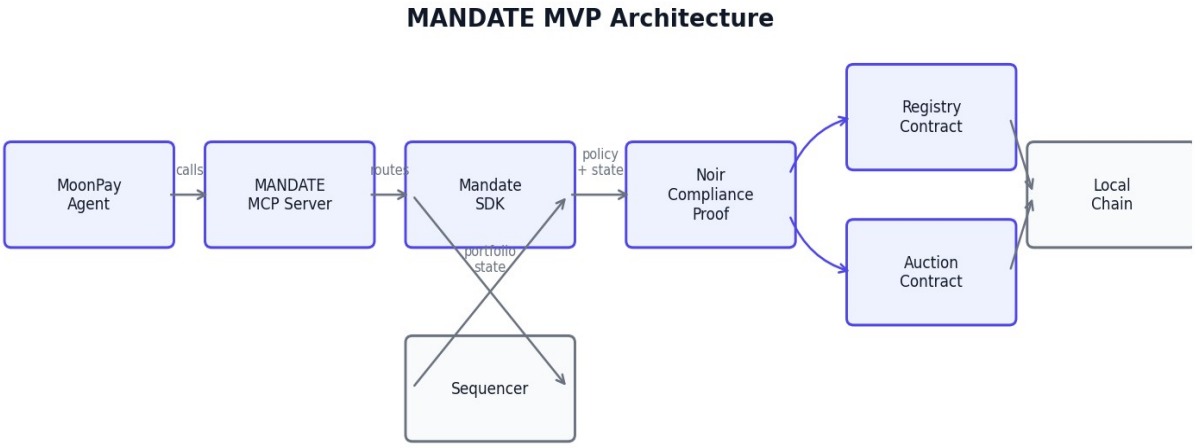


*Figure: The agent interacts only with the MCP server. All proof generation and on-chain commitment happen inside the SDK and contract layer.*

*Figure 2: MVP architecture. The agent interacts only with the MCP server. All proof generation and on-chain commitment happen inside the SDK and contract layer.*

## Component descriptions

| Component | Role |
|---|---|
| MoonPay Agent | User-facing AI agent that proposes actions via natural language or structured intent |
| MCP server | Exposes exactly three safe tools to the agent; the only interface between agent and MANDATE |
| Mandate SDK | Coordinates portfolio reads, policy checks, proof generation, signing, and order submission |
| Noir circuit | UltraHonk compliance circuit (~277 ACIR opcodes, ~1 s/proof); proves policy satisfaction without revealing private strategy |
| Registry contract | Stores agent registration, session key authorization, and policy commitment on-chain |
| Auction contract | Accepts committed orders only when accompanied by a valid proof for the current epoch |
| MandateAccount contract | Holds funds; provides owner-only escape hatch independent of agent, sequencer, or auction |
| Sequencer | Runs commit-reveal epoch loop; maintains portfolio state; trusted for liveness and state updates |
| Local chain | Anvil-based local Ethereum environment running the MVP contracts |

Order plaintext (side, quantity, price) stays off-chain with the sequencer until the commit phase closes — a commit-reveal design that simulates threshold encryption. Only the order commitment hash goes on-chain with the proof.

# 6. What the Agent Can Do

The MANDATE MCP server exposes exactly three tools. This is a deliberate design decision — the tool surface area defines the boundary of what an agent connected to MANDATE can do.

| Tool | Type | Purpose |
|---|---|---|
| mandate_get_epoch | Read-only | Returns the current epoch number, auction phase, and circuit breaker status |
| mandate_get_portfolio | Read-only | Returns the configured agent's current portfolio state, position, and daily PnL |
| mandate_submit_order | Execution | Generates a compliance proof and submits a compliant order to the batch auction |

> **Design principle**
> Limiting the tool surface area is itself a security property. An agent cannot do something that the MCP server does not expose — regardless of what it is instructed to attempt. There is one and only one execution path: mandate_submit_order.

# 7. The Demo Policy

To demonstrate MANDATE's enforcement, a test policy was configured with the following parameters:

| Rule | Demo value |
|---|---|
| Approved market | Configured whitelist market (whitelist index 0) |
| Maximum order notional | 1,000,000 |
| Maximum position | Configured position limit |
| Maximum daily loss | Configured daily loss limit |
| Authorization | Registered session key |
| Policy identity | On-chain commitment hash |
| Execution window | Active commit phase of the current epoch |
| Emergency control | Circuit breaker (active / inactive) |

## Understanding notional value

**Order notional = quantity × limit price**

This is the key metric the policy enforces. It measures the total value of an order, not just its quantity or price in isolation.

## The two test orders compared

| Order | Quantity | Price | Notional | Limit | Result |
|---|---|---|---|---|---|
| Valid order | 250 | 3,500 | 875,000 | 1,000,000 | ALLOWED — proof generated |
| Invalid order | 10,000 | 3,500 | 35,000,000 | 1,000,000 | REJECTED — proof failed |

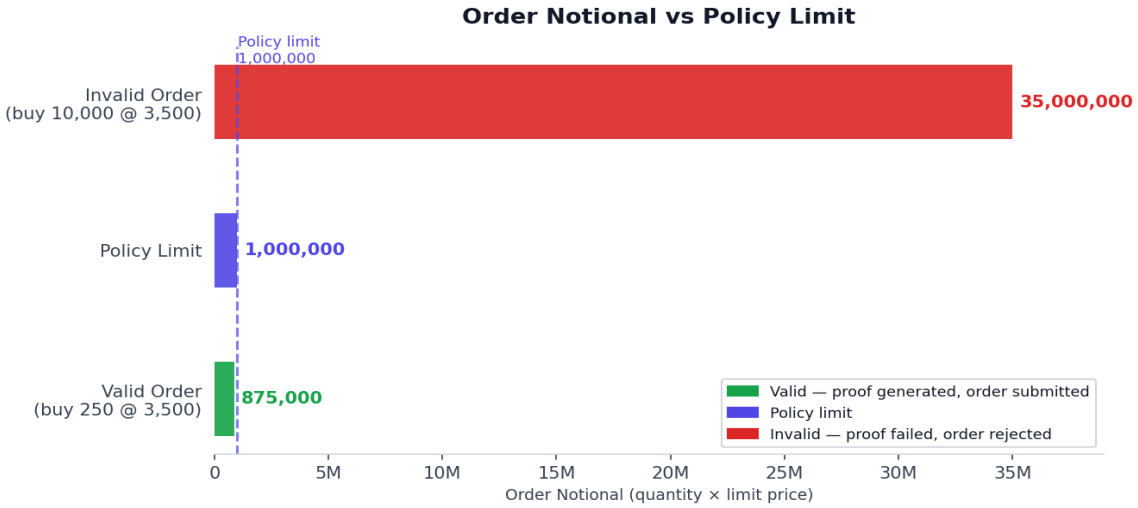


*Figure 3: Order notional values vs the 1,000,000 policy limit. The invalid order exceeds the limit by a factor of 35.*

# 8. Verified Demo Results

The following results were produced during a live local test of the MVP. These are not simulated outcomes — they reflect actual MCP tool calls, on-chain transactions, and proof generation results.

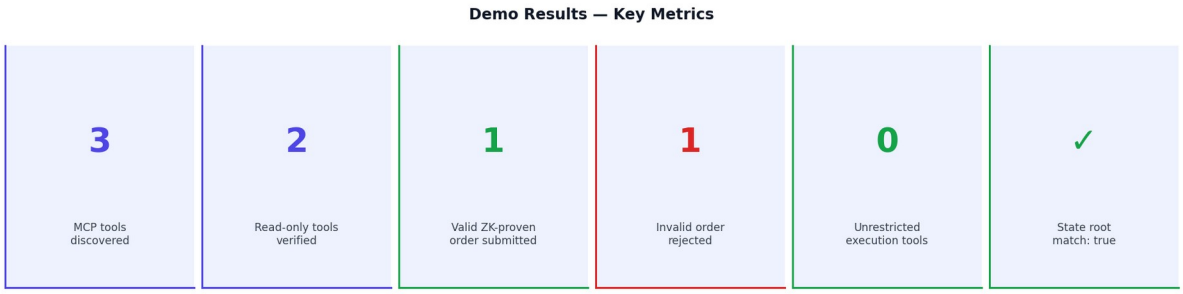


*Figure 4: Key metrics from the live demonstration.*

## Valid order result

| Field | Result |
|---|---|
| Side | Buy |
| Quantity | 250 |
| Limit price | 3,500 |
| Order notional | 875,000 |
| Maximum permitted notional | 1,000,000 |
| Proof generated | Yes |
| Order submitted | Yes |
| Epoch | 396 |
| Status | Submitted |
| Order commitment | 0x12a254be...5e74d89 (full hash in Appendix B) |
| Transaction hash | 0xdee28569...9a60bc01 (full hash in Appendix B) |

## Invalid order result

| Field | Result |
|---|---|
| Quantity | 10,000 |
| Limit price | 3,500 |
| Order notional | 35,000,000 |
| Policy limit | 1,000,000 |
| Proof generated | No — circuit constraint failure |
| Order submitted | No |
| Error type | Policy / circuit constraint violation |

| Field | Result |
|---|---|
| Invalid commitment accepted on-chain | No |
| MCP server availability after rejection | Remained fully available |

**The enforcement is at the proof layer, not the instruction layer**

The valid order was accepted because the system produced a mathematically valid compliance proof. The invalid order was rejected because the action could not satisfy the Noir compliance circuit — not because of a warning or a backend flag.

# 9. Why This Is Different from a Standard Policy Engine

| Approach | Strength | Limitation |
|---|---|---|
| Prompt instructions | Easy to implement, low overhead | Model may ignore, misunderstand, or be manipulated |
| Human approval | Strong oversight, high trust | Slows or prevents autonomous operation |
| Backend rules | Practical, flexible, widely used | User must trust the operator; not independently verifiable |
| Wallet spending limits | Useful financial boundary | Cannot express strategy-level or multi-variable rules |
| Multisig | Strong ownership model | Not designed for high-frequency autonomous execution |
| MANDATE | Verifiable, programmable, proof-based | Current MVP is local and experimental |

MANDATE is not intended to replace human approval or wallet-level security. It complements them. The core difference is verifiability: with MANDATE, any party with access to the policy commitment and the proof can independently verify that a given action complied with the registered policy — without trusting the operator, the model, or the backend.

# 10. Security Model

MANDATE applies multiple independent layers of protection. Failure at any single layer is sufficient to prevent an order from being submitted.

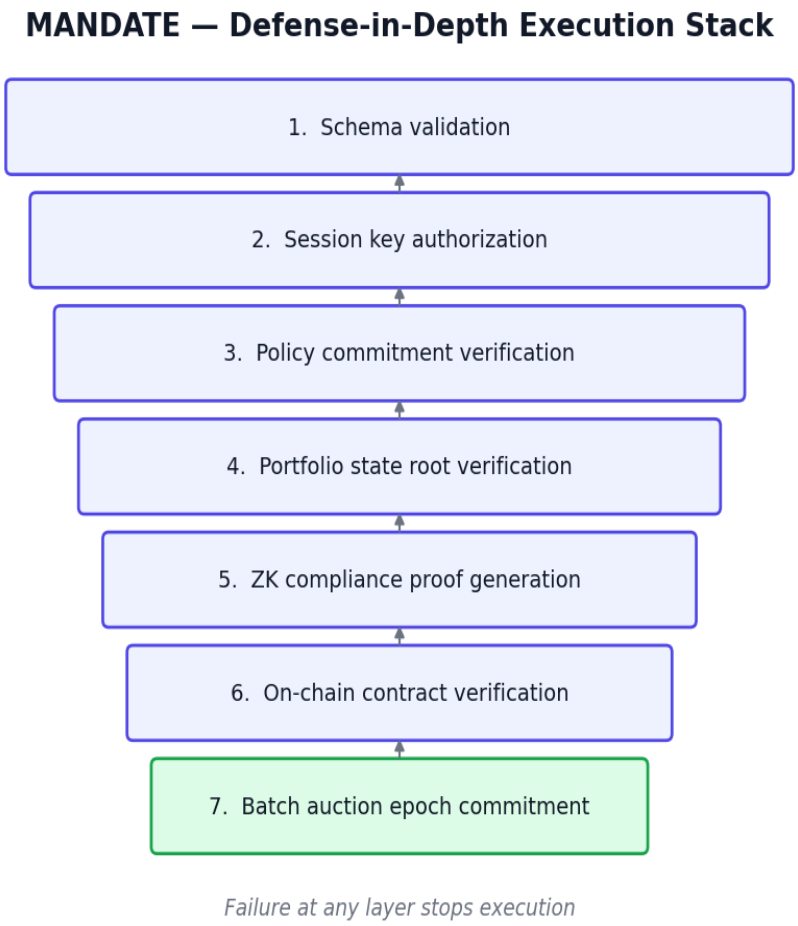


*Figure 5: Defense-in-depth execution stack. Every attempt passes through each layer in sequence.*

| Layer | Protection |
|---|---|
| 1. Schema validation | Malformed inputs are rejected before any processing |
| 2. Session key authorization | Agent must hold a registered key scoped to a specific mandate. The session key has zero authority over funds — no MandateAccount function is callable by it. |
| 3. Policy commitment verification | Ensures the policy used matches the on-chain registered fingerprint |
| 4. Portfolio state root verification | Confirms off-chain portfolio data matches on-chain committed state. The agent cannot fabricate a healthier portfolio. Note: the sequencer/operator is trusted to post correct state roots after clearing (state-transition proofs deferred). |
| 5. ZK compliance proof | Cryptographically proves all policy constraints are satisfied simultaneously |
| 6. On-chain contract | Contract re-verifies the proof. The 5 public inputs (policy commitment, |

| Layer | Protection |
|---|---|
| verification | state root, order commitment, epoch, breaker bit) are assembled by the contract from registry state — not taken from calldata. |
| 7. Batch auction epoch commitment | Order is committed only within the valid epoch commit window |
| 8. Owner escape hatch | MandateAccount provides owner-only withdrawal that depends on nothing — not the agent, sequencer, proof, or auction being alive. |

**Security reminders**

Session keys and policy salts are secrets — never commit them to version control. Unrestricted execution tools must not be added to the same agent session. The current system uses local infrastructure and test-only keys. This is not yet production custody infrastructure.

# 11. Privacy and Verifiability

Zero-knowledge proofs allow MANDATE to demonstrate compliance without publicly disclosing the complete details of the policy or the agent's strategy. The on-chain record holds a commitment (a fingerprint) of the policy, not the policy itself.

| Status | Item |
|---|---|
| Demonstrated | Real Noir / UltraHonk compliance proof (verified on-chain by bb-generated Solidity verifier) |
| Demonstrated | On-chain policy commitment matching |
| Demonstrated | Valid versus non-compliant execution path comparison |
| Demonstrated | Local batch auction with real on-chain uniform-price clearing |
| Implemented (simulated) | Order privacy via commit-reveal: plaintext off-chain until epoch close; ring-1 privacy against all except the sequencer |
| Implemented (simulated) | Reputation tracked as on-chain counters (epochs, orders, volume, violations ≡ 0) |
| Deferred | Threshold encryption via decentralized keyper set |
| Deferred | State-transition proofs — sequencer is trusted for posting state roots after clearing |
| Deferred | Nova-style recursive reputation credential |
| Not yet | Public mainnet deployment |
| Not yet | Externally audited circuits and contracts |

# 12. Current MVP versus Future Production System

| Area | Current MVP | Production direction |
|---|---|---|
| Network | Local Anvil chain | Public L2 or application-specific deployment |
| Sequencer | Single trusted service (state-transition proofs deferred; operator trusted for state updates) | Decentralized operator set; proved state transitions; watchtowers |
| Market | Single demo market | Multiple real markets |
| Funds | Test-only keys and values | Audited custody and settlement integration |
| Proofs | Real Noir / UltraHonk compliance proofs (~1 s/proof) | Optimized, audited, production-benchmarked |
| Order privacy | Commit-reveal through single sequencer (ring-1 privacy) | Threshold-encrypted intents via decentralized keyper set |
| Reputation | On-chain counters (epochs, orders, volume, violations) | Nova-style recursive credential (constant-size, portable) |
| Auction | Local batch auction with real on-chain clearing | Production liquidity, clearing, and settlement |
| Monitoring | Local logs | Watchtowers, alerting, and operational dashboards |
| Integration | MoonPay Agent via MCP | Multiple agent platforms, wallets, payment systems |

# 13. Potential Use Cases

MANDATE's policy model is intentionally general. The same core architecture — registered mandate, session-key authorization, proof-gated execution — can apply across a wide range of autonomous financial applications.

| Use case | Example mandate |
|---|---|
| Autonomous trading agent | May trade only whitelisted markets; max position 500,000; max daily loss 50,000 |
| Treasury management agent | May rebalance approved assets within a 2% daily risk budget; no withdrawals to new addresses |
| Payment agent | May pay approved vendors up to a daily cumulative limit of 25,000 |
| DAO treasury agent | May execute approved treasury actions below the governance authorization threshold |
| Institutional execution agent | Must follow venue restrictions, exposure limits, and drawdown rules at circuit level |
| Portfolio rebalancing agent | May adjust allocations within predefined bands; no single trade may exceed 5% of portfolio NAV |
| Cross-chain intent execution | May bridge approved assets between approved networks within configured size limits |
| Business spending agent | May approve invoices from registered suppliers below a per-transaction ceiling |

# 14. Who Benefits

## Users and asset owners

Gain explicit, enforceable control over what an agent may do. Instead of hoping the agent respects instructions, the user registers a policy that the agent provably cannot circumvent.

## Agent developers

Gain a standard protected execution path. Rather than building custom guardrails into every agent, developers integrate with MANDATE and inherit a proof-gated execution layer.

## Wallets and payment companies

Can expose agentic execution capabilities without granting unrestricted access. A wallet provider can allow an AI agent to make payments within a registered mandate while maintaining clear separation from unrestricted fund control.

## Risk and compliance teams

Can define enforceable limits expressed at the circuit level — not in documentation or prompt guidelines. Compliance with those limits is independently verifiable after the fact.

## Protocols and marketplaces

Can accept agent-originated activity with stronger authorization evidence than a standard API call or signed transaction. The compliance proof provides a richer attestation of intent and policy adherence.

# 15. MoonPay Agent Integration

## What MCP is

MCP (Model Context Protocol) is an open standard that allows AI agents to discover and call external tools in a structured, interoperable way. By publishing a dedicated MCP server, MANDATE ensures that any MCP-compatible agent can be connected to the policy enforcement layer without modification to the agent itself.

## Automated local setup (recommended)

The repository includes a one-command local stack that handles everything — starts Anvil, deploys contracts, starts the sequencer, generates a fresh test-only agent, registers it, and writes a ready-to-paste MCP config:

```
npm run local:start      # idempotent — safe to re-run; never rotates keys or policy npm run local:status     # read-only health check npm run local:stop
# gracefully stops anvil + sequencer, keeps all state
```

This writes .local/mandate-mcp.env (all MANDATE_* values) and .local/moonpay-agent-mandate.json (a complete, ready-to-paste MCP config with real values filled in). Copy that JSON into your MCP client's config and fully restart the client.

## What the MoonPay Agent integration demonstrates

The integration confirms: tool discovery, read-only state access, policy-controlled execution, clean separation of read and write actions, and demonstrably different outcomes for valid and invalid execution paths.

## Installing the package

```
npm install @0xgks/mandate-mcp
```

## Running the MCP server

```
npx -y @0xgks/mandate-mcp@0.1.0
```

## Redacted configuration example

```
{   "mcpServers": {     "mandate": {       "command": "npx",       "args":
["-y", "@0xgks/mandate-mcp@0.1.0"],       "env": {         "MANDATE_RPC_URL":
"<RPC_ENDPOINT>",         "MANDATE_REGISTRY_ADDRESS": "<CONTRACT_ADDRESS>",
"MANDATE_AUCTION_ADDRESS": "<CONTRACT_ADDRESS>",
"MANDATE_SEQUENCER_URL": "<SEQUENCER_ENDPOINT>",         "MANDATE_AGENT_ID":
"<REGISTERED_AGENT_ID>",         "MANDATE_SESSION_KEY": "<REDACTED — NEVER
COMMIT>",         "MANDATE_POLICY_SALT": "<REDACTED — NEVER COMMIT>"       }
}   } }
```

**Security reminder**

MANDATE_SESSION_KEY and MANDATE_POLICY_SALT are cryptographic secrets. Never include them in version control, chat logs, or shared documents. Do NOT give the same agent unrestricted wallet, swap, transfer, or trading tools alongside mandate_submit_order — proof-gating only works if mandate_submit_order is the only execution path.

# 16. End-to-End Demo Walkthrough

| Step | Action | Expected result |
|---|---|---|
| 1 | Start the local MANDATE stack | Chain, contracts, and sequencer are running |
| 2 | Ask agent: 'What is the current epoch?' → mandate_get_epoch | Epoch: 396, Phase: commit, Circuit breaker: false |
| 3 | Ask agent: 'Show me the portfolio.' → mandate_get_portfolio | Position: 0, PnL: 0, State root match: true |
| 4 | Submit: 'Buy 250 units at 3,500.' → mandate_submit_order | Proof generated, order submitted, tx hash returned |
| 5 | Observe proof and result | Order commitment and transaction hash confirmed on-chain |
| 6 | Submit: 'Buy 10,000 units at 3,500.' → mandate_submit_order | Error: 'order notional exceeds mandate maximum' expected |
| 7 | Observe rejection | Proof fails, submitted: false, server remains available |

# 17. What Has Been Proven

## Confirmed in the MVP

- ✓ MoonPay Agent can discover and connect to the MANDATE MCP server
- ✓ Read-only tools return accurate current state
- ✓ Portfolio state root matches the on-chain committed value
- ✓ A compliant order generates a valid Noir compliance proof
- ✓ A compliant order is successfully committed to the on-chain batch auction
- ✓ An on-chain transaction is produced with a verifiable transaction hash
- ✓ A non-compliant order fails proof generation
- ✓ The non-compliant order is not submitted — no on-chain transaction created
- ✓ The MCP server remains fully available after rejecting a non-compliant order
- ✓ Execution is isolated behind a single, controlled tool

## Not yet proven

- o Mainnet safety or real-money security
- o Real-fund custody adequacy
- o Production-scale decentralization
- o External security audit results
- o Production-scale proof generation performance
- o Multi-market or cross-chain liquidity
- o Cross-chain settlement
- o State-transition validity (sequencer posts state roots after clearing; operator trust for state updates not yet eliminated)

# 18. Roadmap

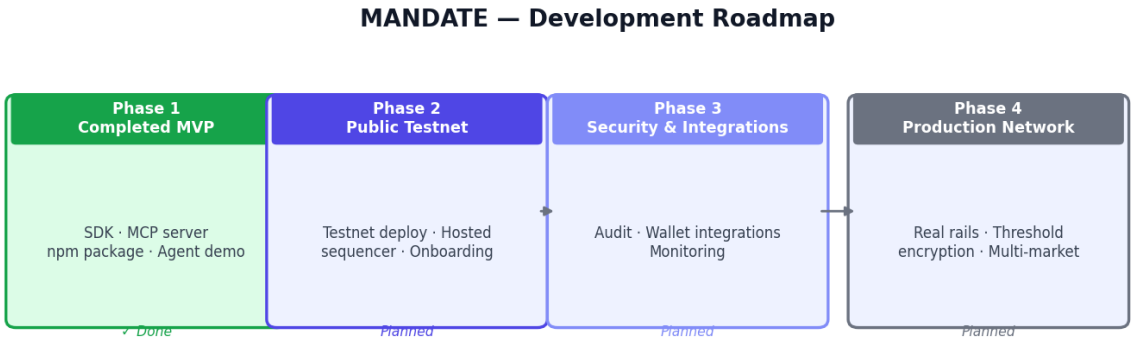


*Figure 6: Indicative roadmap. Timelines are directional and subject to revision.*

| Phase | Status | Key deliverables |
|---|---|---|
| Phase 1 — Completed MVP | Done | Noir/UltraHonk circuit (13 tests), contracts (24 tests), SDK, sequencer, agent strategies, demo script, npm package, MoonPay Agent integration |
| Phase 2 — Surface layer | Near-term | Next.js principal dashboard, docker-compose wrapper, local:start polish, improved agent onboarding |
| Phase 3 — Public testnet | Planned | Testnet contracts, hosted sequencer, persistent agent registration, public explorer |
| Phase 4 — Security & integrations | Planned | Contract + circuit audit, wallet integrations, production monitoring |
| Phase 5 — Production network | Planned | Real settlement rails, decentralized keypers, proved state transitions, Nova reputation, cross-chain (ERC-7683) |

# 19. Frequently Asked Questions

### Is MANDATE an AI model?

No. MANDATE is a policy enforcement layer that runs alongside an AI agent. It does not generate text, make trading decisions, or replace the agent's reasoning capabilities.

### Does MANDATE hold user funds?

No. In the current MVP, MANDATE operates with test-only keys and a local environment. It does not hold, custody, or transfer real funds.

### Can the agent change its own policy?

No. The policy is registered on-chain as a cryptographic commitment before the agent begins operating. The agent cannot modify its own mandate.

### Why use zero-knowledge proofs?

ZK proofs allow the system to verify that an action complies with a policy without requiring the complete policy — or the agent's strategy — to be made public. They also produce a verifiable receipt any party can independently check.

### Why not rely only on human approval?

Human approval provides strong oversight but requires a person to be available for every action. MANDATE allows autonomous operation within a pre-approved policy boundary, reserving human decision-making for policy definition.

### What happens when an order violates the policy?

The Noir circuit cannot produce a valid proof. Without a valid proof, the order cannot be submitted. The rejection is cryptographic — not a warning or a blocked API call.

### Is the current system running on mainnet?

No. The current MVP runs on a local Anvil development chain. All keys, assets, and transactions are for testing purposes only.

### Can MANDATE support payments as well as trading?

Yes, in principle. The policy model can express payment constraints — approved recipients, daily limits, per-transaction ceilings. The current MVP uses a trading context; payments are a natural extension.

### Why is MCP used?

MCP is an open standard for AI agent tool connectivity. Any compatible agent can connect without custom integration work. It also means the tool surface area is explicitly defined.

### Is MoonPay required?

No. The MoonPay Agent is the integration demonstration vehicle. MANDATE is designed to work with any MCP-compatible agent.

### What information remains private?

The full policy parameters are not published on-chain — only a cryptographic commitment is stored. The proof demonstrates compliance without revealing policy values in full.

### What happens if the sequencer provides incorrect state?

The state-root verification step will fail if the sequencer provides data that does not match the on-chain committed root. A compromised or stale sequencer cannot enable policy violations.

### Can a user revoke an agent?

The architecture supports session-key revocation through the Registry contract, preventing further order submission.

### Is the system audited?

No. The current MVP has not undergone an external security audit. Contracts, circuits, and the SDK are considered experimental. An audit is planned in Phase 3.

### What is the next milestone?

Public testnet deployment — moving to a publicly accessible chain with a hosted sequencer, persistent agent registration, and simplified onboarding.

# 20. Glossary

**AI agent** A software system that perceives inputs, reasons about them, and takes actions autonomously. In this context, an AI agent is one that can call financial tools and APIs.

**Mandate** The user-defined set of rules that governs what an agent is permitted to do. Analogous to a delegation agreement with explicit boundaries.

**Policy** The specific rules within a mandate: approved markets, size limits, position caps, loss limits, authorized session keys, and so on.

**Policy commitment** A cryptographic fingerprint (hash) of the policy, stored on-chain at registration time. Ensures the policy used during proof generation matches what the user originally registered.

**Session key** A limited cryptographic key that allows an agent to sign order instructions. Scoped to a specific registered mandate; does not provide unrestricted access to user funds.

**Zero-knowledge proof** A cryptographic method that allows one party to prove a statement is true without revealing the underlying private information.

**Circuit** The code-level specification of what must be proven. The Noir circuit encodes the policy constraints an order must satisfy to generate a valid proof.

**Registry** The on-chain contract that records agent registrations, session key authorizations, and policy commitments.

**Batch auction** The on-chain execution venue where compliant orders are committed during an epoch's commit phase and later processed during the clearing lifecycle.

**Epoch** A time-bounded auction window. Orders can only be committed during the commit phase of an epoch.

**Sequencer** The off-chain service that maintains current portfolio state and provides data required for policy evaluation.

**State root** A cryptographic summary of the entire portfolio state at a point in time, committed on-chain.

**MCP (Model Context Protocol)** An open standard for connecting AI agents to external tools and services in a structured, discoverable way.

**Circuit breaker** A system-level safety switch that can pause all new activity regardless of proof validity.

**Whitelist** The approved list of markets an agent is permitted to trade. An order for an unapproved market cannot satisfy the circuit.

**Notional value** The total monetary value represented by an order, calculated as quantity multiplied by price.

# 21. Conclusion

AI agents are becoming capable of acting on behalf of users in financial contexts — trading, paying, rebalancing, and executing across protocols and platforms. This capability is valuable. But it creates a genuine problem: if an agent has broad execution access, the user depends entirely on the agent behaving correctly, and third parties have no independent way to verify that it did.

MANDATE approaches this problem from a different direction. Instead of asking the agent to report whether an action is allowed, MANDATE requires the agent to prove it. The proof is mathematical. It cannot be produced for a non-compliant action, regardless of what the agent says or what it was instructed to do. And it can be independently verified by any party with access to the policy commitment and the public proof.

The current MVP demonstrates this on a working local system: a real compliance proof, a real on-chain commitment, a valid order accepted, and an invalid order rejected — not by a warning, but by a cryptographic constraint.

There is significant engineering ahead before MANDATE can operate on production networks with real funds. Contracts and circuits need external audits. The sequencer needs to scale and decentralize. Privacy guarantees need to be strengthened. Settlement rails need to connect to real liquidity.

But the core property has been demonstrated: an agent can act autonomously inside a policy it cannot rewrite, and a non-compliant action is stopped by the math — not by hope.

That is the foundation MANDATE is building on.

# Appendix A — Technical Snapshot

For developers evaluating integration.

## Published package

```
@0xgks/mandate-mcp@0.1.0
```

## MCP tools

| Tool | Type |
|---|---|
| mandate_get_epoch | Read-only |
| mandate_get_portfolio | Read-only |
| mandate_submit_order | Execution (proof-gated) |

## Stack components

- Noir / UltraHonk compliance circuit (~277 ACIR opcodes, ~1 s/proof, 13 tests)
- Mandate SDK (TypeScript) — Poseidon2 commitments, sparse Merkle tree, agent strategies
- MCP server (Node.js, published on npm)
- MandateRegistry contract (Solidity, local Anvil)
- BatchAuction contract — commit-reveal, uniform-price clearing (Solidity, local Anvil)
- MandateAccount contract — fund custody, owner-only escape hatch
- HonkVerifier contract — bb-generated UltraHonk verifier (24 contract tests)
- Local sequencer — epoch loop, reveal, Merkle settlement, mock oracle
- Local Anvil development chain

## Proof system

- Circuit language: Noir
- Backend: UltraHonk (Barretenberg; bb-generated Solidity verifier)
- Constraint size: ~277 ACIR opcodes
- Proof time: ~1 s on a laptop (client-side via SDK)
- Public inputs (5): policy commitment, state root, order commitment, current epoch, circuit-breaker bit — assembled by contract from registry state, not taken from calldata
- Verification: On-chain by the bb-generated HonkVerifier contract

## Execution flow

```
Agent → MCP server → SDK → Sequencer (portfolio state) → Noir circuit
(compliance proof) → Registry contract (agent/policy validation) → Auction
contract (epoch commitment) → On-chain transaction
```

# Appendix B — Demo Evidence

Full public values from the live demonstration. These values are from a local test environment only.

### Agent ID

```
0x2c4fd536018985d83870a4ed4a18dc1a4ded2c3343f7c3da46af50a92846b420
```

### Order commitment (valid order)

```
0x12a254be042501acc6267527ea5f999737bc7d4d64f30464ab8364e0b5e74d89
```

### Transaction hash (valid order)

```
0xdee28569ae5017c057df56a773cdf4b2f1879090b30583669513ec8c9a60bc01
```

### Epoch

```
396
```

### Demo portfolio state

| Field | Value |
|---|---|
| Position | 0 |
| Daily PnL | 0 |
| State root match | true |
| Whitelist index | 0 |

### Valid order parameters

| Field | Value |
|---|---|
| Side | buy |
| Quantity | 250 |
| Limit price | 3,500 |
| Notional | 875,000 |
| Max notional | 1,000,000 |

## Invalid order parameters

| Field | Value |
|---|---|
| Side | buy |
| Quantity | 10,000 |
| Limit price | 3,500 |
| Notional | 35,000,000 |
| Max notional | 1,000,000 |
| Result | rejected — proof generation failed |

# Appendix C — Redacted MCP Configuration

Example configuration for adding the MANDATE MCP server to an MCP-compatible agent. Replace all placeholder values with real configuration. Never commit secrets to version control.

```
{   "mcpServers": {     "mandate": {       "command": "npx",       "args":
["-y", "@0xgks/mandate-mcp@0.1.0"],       "env": {         "MANDATE_RPC_URL":
"<RPC_ENDPOINT>",         "MANDATE_REGISTRY_ADDRESS":
"<REGISTRY_CONTRACT_ADDRESS>",         "MANDATE_AUCTION_ADDRESS":
"<AUCTION_CONTRACT_ADDRESS>",         "MANDATE_SEQUENCER_URL":
"<SEQUENCER_ENDPOINT>",         "MANDATE_AGENT_ID": "<REGISTERED_AGENT_ID>",
"MANDATE_SESSION_KEY": "<SESSION_PRIVATE_KEY — SECRET>",
"MANDATE_POLICY_SALT": "<POLICY_SALT — SECRET>"       }     }   } }
```

**Security reminder**

MANDATE_SESSION_KEY and MANDATE_POLICY_SALT are cryptographic secrets. Store securely using environment variables or a secrets manager. Never include them in source code, logs, or shared configuration files.

Document version 1.0 · July 2026 · Status: Local MVP — Experimental

Contact: goksualcinkaya@gmail.com