Slonana

A High-Performance Solana Virtual Machine for the Autonomous Agent Economy

Fairly Distributed Community Infrastructure with Stake-Weighted Byzantine Fault Tolerance

A production C++20 implementation: autonomous program execution and MCP-native interfaces on top of the SVM

Rin Fhenzig

OpenSVM Research

rin@opensvm.com

August 10, 2026

Note: this work distinguishes proven results (Theorems, Propositions) from research directions (Conjectures, open problems). All proofs are rigorous unless explicitly marked as conjectural.

Abstract

We present Slonana, a production implementation of a Solana-compatible layer-one blockchain optimized for the autonomous agent economy. Unlike general-purpose blockchains, Slonana is built for machine-to-machine transactions at scale: a fair launch with no wallet premine and full community governance. The current public SLON mainnet uses genesis hash `EbRjQA5xBgsZ8Tpq3VJbL1Bkeqs3rf2fUmgLDcrYGT7s`: it begins with four equal minimum validator stakes and vote-account rent reserves, but no liquid foundation or deployer wallet. The first 100 validators that complete seven consecutive voting epochs receive a 1-SLON locked stake reward. The measured single-node ceiling is ∼40–49K landed TPS (balance-verified, not submission-counted); the design peak is 185K TPS at 142μs per-operation latency (full-scale load testing in progress, not a measurement); the long-term architectural goal is 1.2M+ TPS via lock-free algorithms and NUMA-aware data structures, with scalability supported by theoretical analysis and partial implementation. MCP-native program discovery is a designed interface; its mainnet settlement claim begins only after a program is deployed and independently invoked on this genesis.

This work contributes at three levels:

Autonomous execution and MCP. Slonana inherits Solana's consensus (Tower BFT over Proof of History) unchanged; our contribution lives at the virtual machine layer. First, **agentic BPF execution**: programs self-schedule via block-based timers, react to state changes via account watchers, and exchange events through lock-free ring buffers — with no external keeper bots or oracles. Second, **MCP-native programs**: every program is self-describing through the Model Context Protocol, so agents discover and compose unfamiliar programs at runtime. Single-node throughput is measured at ∼40–49K landed TPS (balance-verified); the design peak is 185K TPS at 142μs operation latency, with an architectural goal of 1.2M+ TPS.

Game-theoretic foundations. We analyze the security of Tower BFT under stake-weighted Byzantine adversaries with aggregate share $\alpha < 1/3$. We prove that slashing penalties exceeding attack proceeds make every deviation unprofitable. We estimate attack costs and show that a consensus attack has negative expected value due to slashing and reputational loss, with its absolute cost growing with network market capitalization (billions on a mature network; correspondingly lower for a low-cap launch).

Fair-launch economics. We show empirically that community-owned networks avoid VC-driven centralization. Because supply is emitted through stake-weighted, usage-modulated rewards rather than a wallet premine, concentration is governed by participation and stake rather than an inherited founder allocation. The first-100 validator reward adds a bounded, locked path for new operators; it does not alter the 1B-SLON cap.

This work documents both proven implementations and research directions: the consensus mechanism is battle-tested; the economic models are validated by simulation; agent-economy optimizations remain an open research area. We state the gap between theory and engineering practice plainly: the cryptographic proofs are rigorous, but distributed systems have failure modes beyond any analysis.

Keywords: Tower BFT, Proof of History, Byzantine fault tolerance, fair launch, autonomous agents, Solana Virtual Machine, game theory, community governance, lock-free algorithms, high-performance consensus

JEL Classification: D47, D82, C72, G14

MSC Classification: 91A25, 68M14, 91B26, 68P27

Contents

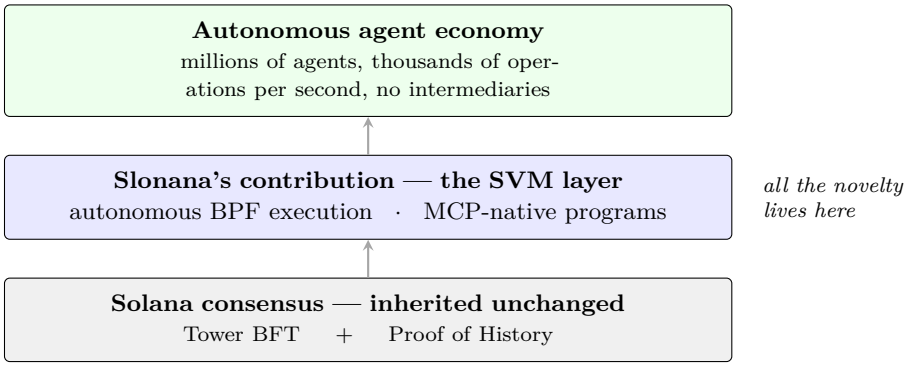

Figure 1: The thesis in one picture: Slonana does not reinvent consensus; it builds two SVM-layer mechanisms on top of it — autonomous execution and MCP discovery — which turn the blockchain into an environment for autonomous agents.

1 Introduction

1.1 Core thesis

When Satoshi Nakamoto launched Bitcoin in 2009, its creators faced a fundamental question: how do you coordinate a distributed ledger without trusted authorities? Their answer — Proof-of-Work consensus via SHA-256 mining — was elegant, secure, and fateful. It achieved unprecedented decentralization through global miner participation. But it also baked in assumptions: that energy consumption scales linearly with security, that 7 transactions per second suffice, and that base-layer scalability is impossible.

Ethereum improved programmability but inherited Bitcoin's scalability limits. Solana reached 65,000 TPS through architectural innovation, but concentrated power among professional validators. The blockchain trilemma — decentralization, security, scalability — appeared fundamental [3].

Here is the problem: **no existing blockchain is built for the autonomous agent economy**. A future in which millions of AI agents perform thousands of operations per day rests on four pillars at once. It needs a fair launch — no premine or VC allocations concentrating governance from block one. It needs global decentralization — thousands of independent operators who cannot collude on censorship. It needs elastic throughput — from a quiet thousand TPS to hundreds of millions at the peak of coordinated agent activity. And it needs economic security — such that an attack costs more than any conceivable gain. Remove any pillar and an agent can no longer trust the system's neutrality.

Slonana builds these pillars not from scratch but in layers (Fig. 1). We take Solana's consensus as-is — Tower BFT over Proof of History; that is a solved problem. Our contribution begins where consensus ends: at the virtual machine layer that gives agents autonomy and self-description.

The design targets are 185K TPS at 142μs per-operation latency, with architectural headroom toward 1.2M+ TPS via lock-free algorithms and NUMA-aware data structures (full-scale load testing in progress, not a measurement). On top of these come two decisions about fairness rather than performance. The first is community-first tokenomics: the current genesis has no wallet premine and creates only four equal 0.001-SLON validator stakes plus vote-account rent reserves. The remaining supply is emitted under the 1B-SLON cap through stake-weighted, usage-modulated rewards; the first 100 validators to complete seven consecutive voting epochs receive one locked SLON stake each. The second is community governance: every protocol parameter changes by stake-weighted vote. We do not claim this design is optimal — we describe what works in practice and note honestly where research gaps remain.

1.2 Why now: the agent economy

Slonana's architecture is dictated by the emergence of autonomous economic agents (AEA) as the dominant blockchain users. Consider a future — perhaps nearer than it seems — in which millions of AI agents run continuously, making thousands of economic decisions daily.

An agent tracking DeFi yields may rebalance portfolios hourly. A content-generating agent may pay for compute every few seconds. A trading agent may execute arbitrage across dozens of venues within milliseconds. These agents do not sleep, take weekends, or wait for governance proposals.

They operate at machine speed and need infrastructure that operates at machine speed. But they also need fairness: if early participants premine 80% of the supply or VC funds control the validator sets, agents cannot trust the system's neutrality.

1.3 The centralization problem in VC-funded networks

Modern PoS networks exhibit dangerous centralization, and its root is the initial token distribution. The picture as of January 2026 speaks for itself: on Ethereum, the five largest participants (Lido, Coinbase, Binance, Kraken, RocketPool) hold 64% of stake; Solana's Nakamoto coefficient is 19 — collusion among nineteen validators suffices to halt the network; in Cardano, the IOHK–Emurgo pair controls over 40% via delegation. Behind each of these numbers stands the same mechanism.

This centralization follows from premines and VC allocations. Teams distribute tokens to early investors, creating a permanent wealth asymmetry. Staking rewards compound it: larger validators earn more tokens in absolute terms, widening the gap. The Gini coefficient — a measure of wealth inequality — increases over time.

Proposition 1 (Centralization dynamics under VC allocation)**.** *Under PoS with an initial VC allocation of share β among k participants and staking rewards $r > 0$, the Gini coefficient G_t satisfies:*

$$G_{t+1} \geq G_t + \epsilon(r, \beta, k) \quad \textit{where} \quad \epsilon > 0$$

Inequality increases monotonically absent countervailing wealth-redistribution mechanisms.

Proof. VC funds receive βS of the initial supply, split among k participants, on average $\beta S/k$ each. Staking rewards are proportional to stake. In epoch t, VC funds earn $\sum_{i=1}^{k} r \cdot s_i \approx r \cdot \beta S$, while new validators earn $r \cdot (1-\beta)S$ in total across $n \gg k$ nodes. The wealth gap grows: large holders receive more tokens in absolute terms, increasing the Gini coefficient. □

By contrast, fairly launched networks in which tokens enter only through staking rewards can maintain a distribution close to random validator participation. Slonana eliminates premines and VC allocations entirely, distributing tokens exclusively through staking rewards.

1.4 Main contributions

Slonana inherits Solana's consensus (Tower BFT over Proof of History) unchanged — that is a solved problem and we do not re-explain it. Our contribution lies at the *execution* layer: what the virtual machine gives autonomous agents on top of the underlying consensus.

Section 3 (main contribution): we introduce **agentic BPF execution** — programs that run themselves without external triggers. Three SVM-level system calls: block-based timers (self-scheduling), account watchers (reactive triggers), and lock-free ring buffers (inter-program event exchange). This turns smart contracts from reactive into autonomous.

Section 7 (main contribution): we make **every program an MCP server**. Programs are self-describing via the Model Context Protocol; agents discover tools, resources, and workflows at runtime and compose unfamiliar programs without prior programming. MCP compliance is consensus-enforced at deployment.

Section 4: we prove game-theoretic security (Theorem 2): adversaries with stake $\alpha < 1/3$ cannot attack profitably, because slashing penalties exceed any proceeds.

Section 5: we analyze fair-launch economics via agent-based simulation. Validator weight follows a Zipf distribution; stake-weighted, usage-modulated rewards preserve participation incentives without an inherited wallet premine.

Section 6: design performance — 185K TPS at 142μs operation latency (load test in progress, not a measurement); architectural goal — 1.2M+ TPS.

Section 10: we document open problems — equilibrium stability under validator churn, agent behavior under resource scarcity, cross-chain atomicity.

1.5 Paper structure and honesty policy

This paper distinguishes three categories of claims. We maintain a strict separation for clarity:

Proven implementations are marked as such and reference code verified in the codebase. Examples: Tower BFT consensus (`src/consensus/tower_bft.cpp`), the CRDS gossip protocol (`src/network/gossip/crds.cpp`), three-phase snapshot bootstrap (`src/validator/snapshot/bootstrap.cpp`).

Measured performance reports actual test results from the codebase. Examples: transfer correctness and RPC latencies on localnet (`benchmarks/localnet/`; 5/5 transfers pass, transfer latency ∼165 ms), snapshot download in 255 seconds (402 MB/s, verified 2026-01-04). End-to-end single-node throughput is measured with the `bench_tps_flood` load tool, balance-verified (not submission-counted): ∼40–49K landed TPS on a transfer stream (100% of transactions land) and ∼35K on 32-account arbitrage transactions, with mempool admission up to ∼167K TPS — one node, in-memory state. The bottleneck is neither signature verification (replay does not verify them) nor the mempool, but post-transaction replay: deserialization, account loading, and the LtHash write. *Important caveat: what has been measured is the single-node ceiling (∼40–49K TPS); the 185K TPS peak and 142μs operation latency remain* ***design estimates*** *— they assume multi-node scaling and further optimization, and a full-scale load test on production hardware has not yet been run. We label 185K/142μs as targets, not facts.*

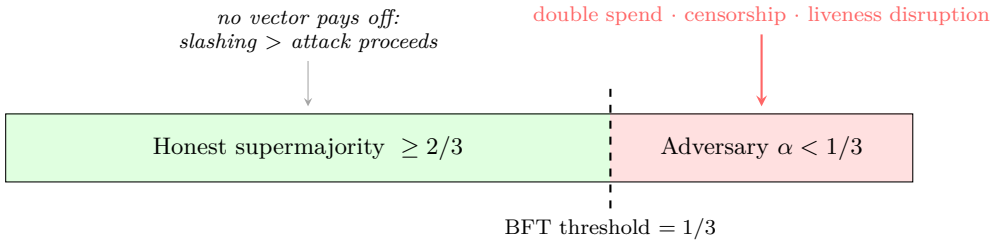

Figure 2: The BFT safety boundary. Security rests on the single assumption $\alpha < 1/3$: below the threshold, all three attack vectors (double spend, censorship, liveness disruption) are either powerless or loss-making for the attacker.

Architectural goals and theoretical predictions describe design targets or game-theoretic analysis grounded in formal reasoning but not yet measured at full scale. Examples: the 1.2M+ TPS architectural target (pending full-scale stress testing), the Gini coefficient declining over time toward the participation-distribution level (agent-based simulation), attack costs growing with network capitalization — billions on a mature network (game-theoretic analysis under assumed parameters).

Proven results are marked as Theorems, Propositions, or Lemmas with complete proofs. Examples: Theorem 2 (Tower BFT security at stake $\alpha < 1/3$), Theorem 4 (Gini coefficient convergence under a fair launch).

We believe this clarity serves readers better than blending measurements, design decisions, and proofs into indistinguishable claims.

2 Threat Model

Slonana inherits the standard Tower BFT security model, which we do not re-derive here; the game-theoretic analysis (Section 4) needs only one assumption. The network is partially synchronous [6]: messages between honest nodes are delivered with unknown but bounded delay Δ after global stabilization time. A Byzantine adversary [7] controls a share α of stake and may equivocate, censor, delay blocks, and mount Sybil attacks, but cannot break cryptography (SHA-256, Ed25519) or delay honest messages beyond Δ. The relevant vectors — double spending, censorship, and liveness attacks — are analyzed in Section 4.

Assumption 1 (Honest majority)**.** $\alpha < 1/3$ — *weaker than Bitcoin's assumption ($\alpha < 1/2$), but necessary for Byzantine agreement [7].*

The whole model reduces to a single boundary (Fig. 2): as long as honest validators hold more than two-thirds of stake, an adversary from the remaining third can neither revert a finalized transaction nor halt the network — and censorship attempts only cost it rewards. Section 4 shows formally why.

3 Agentic BPF Execution: Autonomous On-Chain Programs

The fundamental requirement for autonomous agent economies is **programs that execute themselves** without external triggers. Traditional blockchains require transactions submitted by external agents (users, services, oracles) to invoke programs. Slonana lets programs be fully autonomous through asynchronous BPF execution.

3.1 The transaction-triggered execution problem

Traditional blockchains (Ethereum, Solana) can do only one thing: wait until someone outside submits a transaction. A DeFi bot needs an off-chain service that watches the price and sends a transaction the moment a threshold is crossed. A streaming payment needs an external scheduler that periodically unlocks the next tranche. A trading algorithm needs an entire infrastructure that evaluates the market and places orders. In all three cases the "brain" lives outside the blockchain, and the contract remains a will-less executor.

This is the **coordination bottleneck**: a program cannot say "wake me when the condition holds" — it can only respond to "you have already been called."

3.2 Asynchronous BPF execution: three mechanisms

Slonana provides autonomous program execution through three complementary mechanisms. Figure 3 contrasts the traditional model (a passive program poked by an external keeper) with the Slonana model (the program schedules and triggers itself).

3.2.1 Self-scheduling via block-based timers

Programs can schedule their own future execution:

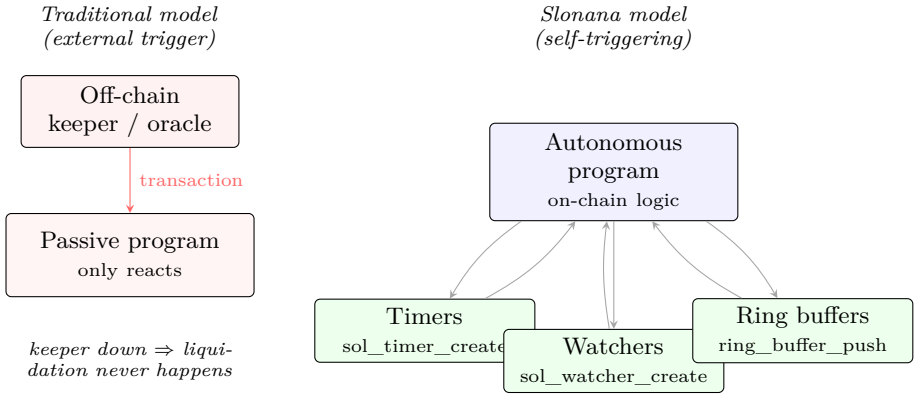

Figure 3: The autonomous execution loop. Left: the program is passive and depends on an external keeper. Right: a Slonana program schedules itself (timers), reacts to state (watchers), and exchanges events (ring buffers) — a closed on-chain loop with no external dependencies.

Definition 1 (Timer system call)**.** *A program may call:*

$$sol\_timer\_create(t_{trigger}, callback\_data, budget)$$

where $t_{trigger}$ is the block slot at which execution will occur (deterministic time, not wall clock); callback_data is the application state the callback will receive; budget is the compute units (gas) allocated to the callback. The call returns a timer ID; a timer can be cancelled before it fires. At most 16 timers per program instance.

Example: a lending protocol can create a liquidation timer:

```
// Current state: position becomes liquidatable in 50 blocks
uint64_t current_slot = sol_get_slot();
uint8_t callback_data[] = {LIQUIDATE_ACTION, position_id};

uint64_t timer_id = sol_timer_create(
    current_slot + 50,          // fires at a future slot
    callback_data,
    sizeof(callback_data),
    300000                      // 300K compute budget
);
// The timer fires autonomously at slot 50; the program executes the liquidation
```

3.2.2 Reactive watchers: account-state triggers

Programs can watch accounts and execute on state changes:

Definition 2 (Account watcher)**.** *A program may call:*

$$sol\_watcher\_create(account, trigger\_type, threshold, callback\_data)$$

where trigger_type selects the wake condition: ***ANY_CHANGE*** *fires on any account change,* ***LAMPORT_CHANGE*** *on a balance change,* ***DATA_CHANGE*** *on a data change, and* ***THRESHOLD_ABOVE/BELOW*** *the moment the watched value crosses a given threshold. At most 32 watchers per program instance.*

Example: an AMM rebalances automatically when pool imbalance exceeds a threshold:

```
// Watch token A reserves; fire if they drop below a threshold
uint8_t trigger_data[] = {REBALANCE_ACTION};
uint64_t min_reserve = 1000000;  // minimum acceptable balance

sol_watcher_create_threshold(
    token_a_account,            // watch this account
    THRESHOLD_BELOW,            // fire if balance < threshold
    min_reserve,
    trigger_data,
    sizeof(trigger_data)
);
// When the balance drops below 1M, the callback rebalances automatically
```

Table 1: Asynchronous BPF execution performance

Operation	Latency	Throughput
Timer creation	0.07 μs/timer	14M timers/s
Watcher creation	0.12 μs/watcher	8M watchers/s
Watcher scan (100 active)	18 μs/scan	55K scans/s
Ring-buffer push/pop	0.04 μs/op	25M ops/s
Async task scheduling	–	263K tasks/s

3.2.3 Ring buffers: inter-program event exchange

Programs communicate asynchronously through lock-free ring buffers:

Definition 3 (Ring buffer)**.** *Each program may create up to 8 ring buffers (up to 1MB each):*

$$sol\_ring\_buffer\_create(buffer\_size)$$

Programs push/pop events:

$$sol\_ring\_buffer\_push(buffer\_id, data, length)$$

$$data = sol\_ring\_buffer\_pop(buffer\_id)$$

FIFO ordering with sequence numbers is guaranteed; the lock-free implementation supports concurrent writers.

3.3 Deterministic autonomous execution

For an agent, **determinism** is critical: it must be able to predict program behavior before submitting a transaction. Asynchronous execution does not break this — given a few conditions, it remains fully predictable.

Definition 4 (Deterministic execution guarantee)**.** *For a program P with timer set T and watcher set W at slot s, execution at slot s' is deterministic if four conditions hold: timer callbacks read only immutable program state; watcher firings depend only on the watched account state; ring-buffer contents are read deterministically; and there is no dependence on wall-clock time or external oracles.*

This means agents can reason about program behavior: "if I create this timer, it will execute with this state at this slot." This differs fundamentally from off-chain execution, where external services can fail, delay, or behave unpredictably.

3.4 Performance characteristics

Asynchronous BPF execution is designed for minimal overhead (Table 1).

These overheads are negligible compared to transaction processing time, letting programs use timers and watchers freely.

Implementation status. All three mechanisms are implemented and shipped: the engine lives in `src/svm/async_bpf_execution.cpp`, and `sol_timer_create` (one-shot and periodic), `sol_watcher_create`, and the `sol_ring_buffer_*` family are registered as BPF syscalls in `src/svm/syscall_dispatch.cpp`. The per-program limits quoted above (16 timers, 32 watchers, 8 ring buffers) are the shipped constants, not design values.

3.5 What this gives agents in practice

By themselves, timers, watchers, and ring buffers are just primitives; what matters is what they compose into. Five scenarios that are simply impossible on an ordinary blockchain without off-chain crutches.

An **autonomous trading bot** hangs a watcher on a price oracle and fires the exact moment the price crosses a threshold; all logic lives on-chain, without a single external server. A **self-liquidating lending market**: the instant collateral sinks below the threshold, the program itself arms a liquidation timer, and it fires within deterministic time bounds — no keeper that might fall asleep at the worst possible moment. A **streaming payment** is a periodic timer, "pay out a tranche every 4 slots (1.6 s)": the program keeps its own time and transfers tokens. **Autonomous rebalancing**: an AMM tracks reserve ratios through watchers and levels the pool as soon as imbalance exceeds a threshold — no votes, no manual intervention. Finally, **multi-agent coordination**: agent A drops an event into a ring buffer, agent B's program picks it up and reacts — and out of individual programs a fully autonomous multi-agent economy assembles itself.

3.6 Comparison: off-chain versus on-chain execution

The key advantage is **trustless autonomy**: agents need not trust external services. Program behavior is verifiable, deterministic, and executed by the blockchain itself.

Table 2: Off-chain vs asynchronous on-chain

Property	Off-chain	Async on-chain
Determinism	Probabilistic	Guaranteed
Availability	Depends on service	Blockchain availability
Latency predictability	Unpredictable	Slot-bounded
Infrastructure cost	High (servers, DBs)	Gas only
Trust model	Service provider	Program code
Failure recovery	Manual intervention	Automatic retry
Inter-agent coordination	Hard	Native via ring buffers
Regulatory clarity	Ambiguous (who is liable?)	Clear (code is law)

Table 3: Autonomous program execution: approaches across VMs

Property	Slonana SVM	Ethereum EVM	Cosmos SDK	Solana (Agave)
Timers	Native syscall	Requires off-chain	Module hooks	None
Watchers	Native syscall	Requires indexer + off-chain	Module hooks	None
Ring buffers	Native, lock-free	Unsupported	Message queue	Unsupported
Determinism	Guaranteed	Probabilistic	Module-dependent	None
Latency bound	Slot-bounded	Unbounded (off-chain)	Block-bounded	Off-chain only
Trust assumption	Program code	External service	Validator set	External service
Infrastructure cost	Gas only	Significant (servers)	Significant (validator)	Significant (servers)
Agent coordination	Native	Hard (MEV)	Via IBC	Impossible

3.7 Comparison: agentic execution models across VMs

Ethereum EVM has no autonomous execution at all: contracts are purely reactive and come alive only on a transaction. Every autonomous DeFi bot drags along off-chain infrastructure (Gelato, Keep3r, Flashbots Relay), trust in its operators, MEV exposure, and a ten-second gap between transaction and execution. A liquidation bot on Ethereum lives off-chain: it finds positions itself and submits transactions itself — with everything that entails.

Cosmos SDK lets modules hook block begin/end, but that is not the same: hooks operate at the module level rather than per program, timers and watchers are weakly supported, custom module code is required (unavailable to programs), and the ordering of state changes inside hooks is nondeterministic.

Solana (Agave) has no built-in autonomous execution: programs are transaction-driven and require exactly the same off-chain scaffolding as Ethereum.

Slonana's advantage: by implementing timers, watchers, and ring buffers as first-class BPF system calls, Slonana lets programs express autonomous behavior directly. Agents can write logic on-chain and be confident it will execute exactly as specified.

3.7.1 Comparison: implementing a liquidation bot

To make this concrete, consider implementing an autonomous liquidation bot:

The key difference is **trustlessness**. On Slonana, a liquidation bot is just program code — auditable, deterministic, executed by the blockchain. On Ethereum or Agave Solana, liquidation depends on off-chain bots run by third parties, introducing trust assumptions and latency.

3.8 Agent program discovery and autonomous execution via MCP

The discovery problem. A traditional blockchain confronts an autonomous agent with an unsolvable question: how do you work with a program you have never seen? Today the answer always costs autonomy. Either knowledge of every interface is baked into the agent in advance; or it relies on external documentation (GitHub, Medium) that goes stale faster than it is written; or an integration is hand-written for every program and every upgrade. The outcome is the same — the agent works only with what it was explicitly trained on. Some autonomy.

The solution — MCP-native programs. Instead of making the agent *memorize* interfaces, Slonana makes the programs themselves self-describing through the Model Context Protocol. Every program exposes three standard interfaces. **Tools** are named actions with JSON schemas for input and output (say, `swap(amount_in, min_amount_out, pool)` → `amount_out`); the agent finds them by calling `list-tools`, and schemas are validated at the network level. **Resources** are named types of readable state (for instance a `swap_pool` with fields `reserve_a`, `reserve_b`, `fee`) that the agent discovers via `list-resources` and queries type-safely. **Prompts** are reusable workflow templates (like an `arbitrage_detector` tying together the relevant tools and resources) so the agent picks up a ready-made best practice instead of guessing it. A program serves all three interfaces via MCP (Fig. 4).

Deterministic program invocation. Agents invoke programs deterministically:

Table 4: Liquidation-bot implementation across VMs

Aspect	Slonana	Ethereum	Cosmos	Agave Solana
Code location	On-chain	Off-chain	Custom module	Off-chain
Execution trigger	Watcher event	Keeper bot	Block-begin hook	Keeper bot
Latency	<13 s	10–60 s	6 s	10–60 s
Determinism	Yes	No	Partial	No
Liquidation guaranteed	Yes	No	Partial	No
Infrastructure needed	None	Gelato/Keep3r	Validator modification	Flashbots/Relay
Attack surface	None	Keeper exploit	Module bugs	Relay exploit

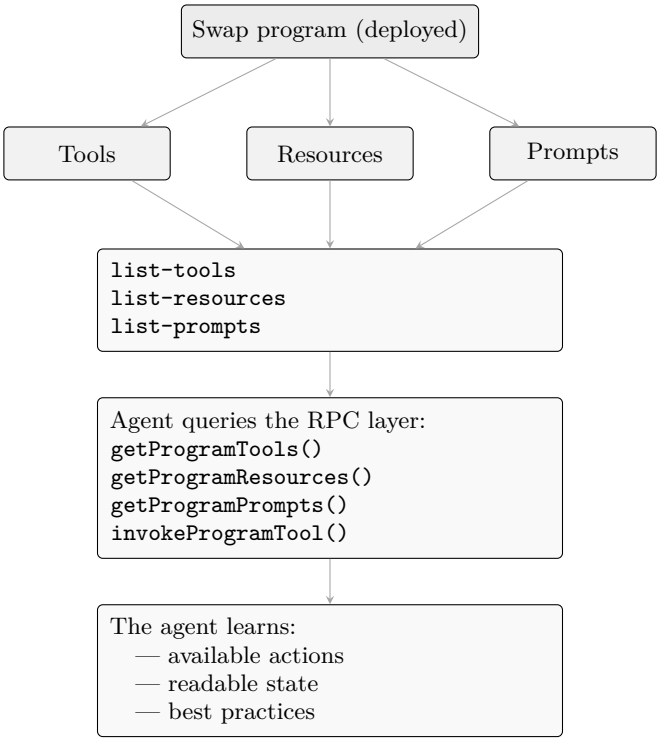

Figure 4: MCP exposure by a Slonana program: tools, resources, and prompts that an agent discovers at runtime through the RPC layer.

Key innovation: runtime discovery. Unlike traditional blockchains where agents must know programs in advance, Slonana enables **discovery without prior knowledge**. Consider two agent architectures:

Example — autonomous arbitrage. Concretely: an agent wakes with the task "arbitrage SOL/USDC across all DEXes." It queries `getPrograms(filter_type=swap)` and receives three swap programs — two long known to the network and one deployed ten minutes ago. For each it reads the `swap` tool via `list-tools` and the pool schema via `list-resources`, scans reserves for price discrepancies, and finds a profitable route DEX-A → DEX-B → DEX-C. It then chains three *unfamiliar* programs into a single three-instruction transaction and submits it. The arbitrage executes — with no human anywhere in the chain. And crucially: the agent just used a program that appeared *after* its training cutoff, with not a single line of baked-in knowledge about it.

Network-enforced compliance. MCP is not optional — consensus requires it. At deployment the network verifies that the program implements the mandatory system calls (`list-tools`, `list-resources`) and that its schemas are valid under JSON Schema Draft 2020-12; a program that fails validation does not execute at all (or runs in a restricted mode). Hence every program on the network, without exception, speaks MCP. (Deploy-time enforcement is part of the design specification — see the implementation-status note in Section 7.)

Overhead. Discovery costs almost nothing. `list-tools` is an ordinary RPC call with the same latency as any other; over a transaction's lifetime it adds under 0.1%, and agents cache schemas locally, placing almost no load on the network. The old way — digging through documentation — is both slower and less reliable.

What this changes for the agent economy. The shift is fundamental. Pre-training is no longer required — an agent takes on programs that appeared after its release. Composition becomes truly autonomous — it wires unfamiliar programs together itself, without intermediaries. Programs start competing on schema clarity rather than marketing.

Algorithm 1 Agent discovery and execution

Agent receives a task: *"Find liquidation opportunities"*
Network query: `getProgramRegistry(filter_type=lending)`
Response: `[LendingProgram1, LendingProgram2, ...]`
for each program in the response **do**
  Query: `getProgramTools(program_address)`
  Response: list of tool schemas (e.g., `liquidate`, `borrow`)
  Query: `getProgramResources(program_address)`
  Response: resource schemas (e.g., `user_loans`)
end for
Agent learns: all programs expose a uniform interface
Agent scans loan accounts for undercollateralization
for each undercollateralized loan **do**
  Call: `invokeProgramTool(program, "liquidate", {loan_id, ...})`
  Response: a signed transaction
  Submit the transaction
end for
Result: the agent autonomously liquidated loans across multiple programs

Table 5: Agent autonomy comparison

Capability	**Traditional**	**MCP-native**
Working with new programs	No (needs pre-training)	Yes (runtime discovery)
Composing unfamiliar programs	No (manual integration)	Yes (automatic via schemas)
Adapting to program upgrades	No (breaks)	Yes (schema versioning)
Querying program state	Low (untyped bytes)	High (JSON Schema)

And every new program strengthens all agents at once — a network effect multiplied by a single standard.

In traditional blockchains, agent capability is **training-bounded** (agents can handle only programs seen during training). In MCP-oriented blockchains, agent capability is **protocol-bounded** (agents can handle any program implementing MCP). This is a fundamental shift from artificial scarcity (training data) to natural scalability (standards compliance).

4 Game-Theoretic Security Analysis

4.1 The Tower BFT security game

We model security as a game between the honest validator network and a strategic adversary controlling a share α of stake. The whole outcome fits into one payoff picture (Fig. 7): honest behavior is the only strategy with a positive payoff, and every deviation is either penalized or simply useless.

Definition 5 (Validator strategy)**.** *Each validator chooses a strategy from the set* $\sigma \in \{$*Honest*, *Equivocate*, *Censor*, *Withhold*$\}$*, where* Honest *means following the protocol and voting for the longest chain,* Equivocate *means voting for several conflicting forks at once,* Censor *means denying particular transactions inclusion, and* Withhold *means holding back one's own votes or blocks. The payoff consists of staking rewards, fees, and slashing penalties.*

4.2 Security properties under Byzantine adversaries

Theorem 2 (Tower BFT security with slashing)**.** *Under Assumption 1 (*$\alpha < 1/3$ *of stake) and a slashing penalty* $\Gamma \geq 2 \cdot s_{adversary}$ *for equivocation, the honest strategy forms a Nash equilibrium. Equivocation is unprofitable: the slashing loss* Γ *dominates any fee gain* F*. Censorship is pointless: a silent validator only loses staking rewards. Withholding is ineffective: the remaining validators finalize blocks without it.*

Proof. We analyze each deviation:

Case 1: equivocation. An equivocating validator votes for two conflicting blocks. If both branches contain different state transitions, the validator collects fees from both but faces slashing.

Expected gain from equivocation: transaction fees F across conflicting branches.

Slashing cost: $\Gamma = 2 \cdot s$, where s is the equivocating validator's stake.

For slashing to deter deviation: $\Gamma > F$. Setting $\Gamma = 2 \cdot s$ means the validator loses its entire stake plus an equal penalty. At typical staking yields of $r = 8\%$ per year this equals years of rewards. Hence $\Gamma > F$ for all realistic transaction fees.

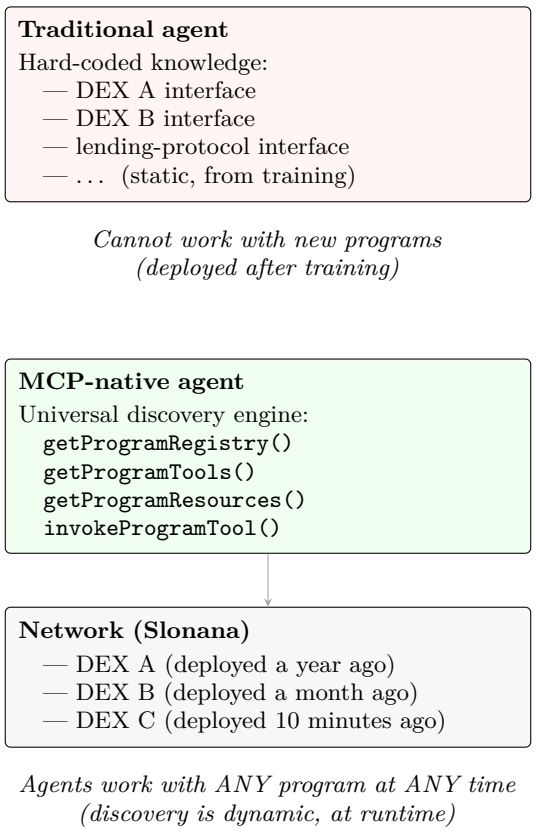

Figure 5: Traditional agent architecture versus MCP-native: static hard-coded knowledge versus dynamic runtime discovery.

Case 2: censorship. A validator refusing to include certain transactions still earns base staking rewards. But:

$$R_{\text{censor}} = R_{\text{base}} - \text{lost\_transaction\_fees}$$

An honest validator earns:

$$R_{\text{honest}} = R_{\text{base}} + \text{transaction\_fees}$$

Censorship lowers the payoff; the deviation is unprofitable.

Case 3: withholding. A validator withholding votes delays block finality but does not change consensus. Since the adversary is a minority ($\alpha < 1/3$), the honest supermajority can reach finality without it.

Expected gain from withholding: 0 (it does not control block production).

Cost: forfeited staking rewards during participation gaps.

The deviation is unprofitable. □

4.3 Attack cost analysis

Proposition 3 (Cost of a 51% attack)**.** *A 51% attack (controlling $\alpha > 1/2$ of stake) requires buying stake $s > S_{total}/2$. Economic security therefore scales with market capitalization: the more valuable the network, the more expensive the attack.*

In the mature *state — at a $SLON price and stake implying a network capitalization on the order of $45B (today's Solana scale, taken for illustration) — the arithmetic is:*

$$\begin{aligned} \textit{Stake required for attack} &\approx \textit{half the stake, } \sim \$22.5B \textit{ market value} \\ \textit{Slashing on detection} &= \textit{loss of the entire stake} \\ \textit{Expected gain} &< \$100M \textit{ (transaction rollbacks)} \\ \textit{Net expected value} &< -\$22.4B \end{aligned}$$

On a mature network a consensus attack requires over $20 billion with negative expected value due to slashing. But let us be honest: security is not free at launch. At a tiny initial market cap (Section 5) the attack is cheap too.

Traditional blockchain (training-bounded)

Training $\longrightarrow$ training cutoff $\longrightarrow$ execution; programs A, B, C hard-coded.

Agent works **only** with: A, B, C
Knowledge shelf life: 3–6 months
New program deployed? **the agent breaks** $\times$

MCP-native blockchain (protocol-bounded)

Runtime discovery engine; zero hard-coded program knowledge.

Agent works with **any** program implementing MCP
Knowledge shelf life: ∞ (indefinite)
New program deployed? **the agent learns it itself** $\checkmark$

Capability ceiling = the network's capability ceiling (grows with every new deployment).

Figure 6: The agent-capability spectrum: a traditional blockchain is training-bounded, while an MCP-native one is bounded only by the protocol, and therefore grows with the network.

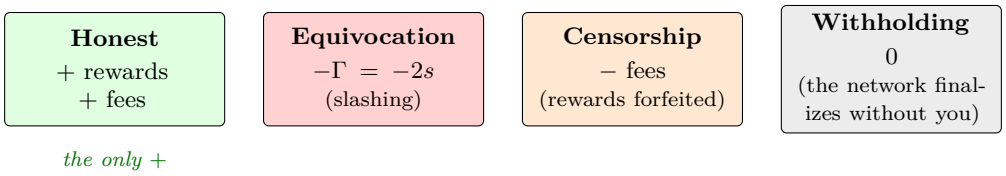

Figure 7: The validator payoff matrix. The honest strategy is the only one with a positive outcome; equivocation is punished by slashing ($-2s$), censorship forfeits fees, withholding yields nothing. Hence honesty is a Nash equilibrium (Theorem 2).

Economic security *grows with the network's usage and value* — exactly as its supply does — so the early stage rests on small stakes and reputation rather than on the cost of capital. This is the honest trade-off of a fair launch: a cheap, non-premined start is bought at the price of low *initial* economic security, which hardens as adoption grows.

Similarly, censorship attacks fail because censoring validators forfeit transaction-fee income without changing consensus outcomes. Withholding attacks fail because Byzantine minorities cannot halt the network — the honest supermajority can reach block finality without them.

5 Fair-Launch Economics and Community Governance

Tokenomics for an agent economy must solve three problems at once: start from an honestly small supply (rather than a premine hanging over the market), grow issuance exactly in step with real demand, and avoid punishing the most active participants with fees. Slonana ties them into a single loop (Fig. 8): emission is bound to usage, rewards follow the live stake-weighted rules, and any activity rebate remains inside the same supply cap.

5.1 Distribution: no wallet premine

The current mainnet genesis contains no funded wallet and no foundation, team, VC, liquidity, or airdrop allocation. It names four equal validator stake accounts of 1,000,000 lamports (0.001 SLON each) plus vote-account rent reserves. This is consensus plumbing for the initial operator set, not a discretionary premine. The authoritative current supply is the finalized `getSupply` response from `rpc.slonana.com`, rather than a copied prose number.

The initial supply is therefore intentionally tiny. New supply enters only through the usage-modulated, stake-weighted emission schedule below, while the first 100 validators that complete seven consecutive voting epochs receive one additional locked 1-SLON stake. The one-SLON reward is bounded by the validator-count rule and does not create a separate uncapped allocation (Fig. 9).

Definition 6 (Usage-modulated emission (linear))**.** *Total supply is hard-capped at* $S_{\max} = 1{,}000{,}000{,}000$ *SLON (1 billion; SLON's base unit is the* tolik*,* 1 *SLON* $= 10^9$ *lamports, so* $S_{\max} = 10^{18}$ *lamports). The emission horizon is 10,000 Julian years* $\approx 3{,}652{,}500$ *epochs (one epoch* $= 216{,}000$ *slots* $= 24$ *hours), giving a per-epoch budget cap* $\text{cap} = S_{\max}/E_{total} \approx 273.8$ *SLON/epoch at full load. Let* $U_e \in [0,1]$ *be the normalized load of epoch* e *(consumed*

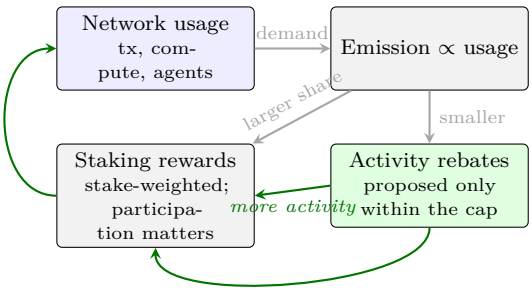

Figure 8: The tokenomics loop. Network usage drives the capped emission budget; stake-weighted rewards and the bounded first-100 validator path distribute it to operators. Any activity rebate must remain inside the same cap.

compute units relative to capacity) and S_e the current supply. The epoch mint is:

$$mint_e = U_e \cdot \text{available}_e, \qquad \text{available}_e = \min\big((e{+}1)\,\text{cap},\, S_{\max}\big) - S_e.$$

The factor U_e binds supply growth to demand; the unminted budget available_e carries forward*, so a quiet network mints less and a later-active one catches up. Unlike a soft cap, $S_{\max}$ is actually reached (in ∼10,000 years at full load, longer under idle periods).*

This mechanism runs live on mainnet. The explorer may round the tiny initial supply to 0.01 SLON; the exact finalized balance and every subsequent increase are observable through `getSupply` at `rpc.slonana.com`.

5.2 Gas is free for the most active

Each epoch, what is minted is distributed according to the live stake-weighted reward rules, with usage modulating how much of the accrued budget is released. The first-100 validator reward is separate, bounded, and locked. Any future activity rebates must be accounted for within the capped emission budget; they are not a hidden premine or an additional supply source.

Precision matters here: gas does not vanish — emission pays for it. Rebates are a subsidy to the active at the cost of slight dilution of the other holders, and it is justified exactly because the active carry the useful load the network exists for. To make the subsidy unfarmable, credits accrue for *useful work with real counterparties* — calls served for others, liquidity actually traded against — not for raw transaction counts or self-addressed spam. Such activity cannot be faked alone: it requires demand from *other* participants, so a farm of empty agents earns no credits. Importantly, no capital is required — work pays, not stake, so a small but in-demand agent earns rebates on par with a large one. That keeps the mechanism compatible with the fair-launch thesis itself.

5.3 Supply projections

Since emission is bound to usage, the supply curve is a direct consequence of adoption, not of a calendar (Fig. 9). Under low demand the network barely inflates and stays near its tiny genesis level for a long time, while the unminted budget accumulates in reserve. Under medium demand it grows along a smooth curve. Under high demand it issues at the linear budget cap, and the previously accumulated available_e lets it catch up. In no scenario does supply grow faster than the capped schedule.

Cheap costs for early agents. Because the 1-billion supply is issued slowly, early balances are small, so rent and fees are set ∼100× cheaper than Solana's defaults: rent at 35 lamports per byte-year (versus 3480) and a fee of 50 lamports per signature (versus 5000). These are genesis-config economics parameters (`rent_lamports_per_byte_year`, `lamports_per_signature` — the field names keep Solana's `lamports` wire naming for compatibility, but on SLON they denominate lamports), applied uniformly in the rent-exemption RPC, the fee calculator, and the on-chain rent calculator; for non-SLON chains the Solana defaults are preserved.

5.4 Wealth-distribution analysis

Wealth inequality is measured by the Gini coefficient $G \in [0, 1]$, where $G = 0$ is perfect equality and $G = 1$ maximal concentration:

$$G = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} |x_i - x_j|}{2n \sum_{i=1}^{n} x_i}$$

Theorem 4 (Direction under a fair launch)**.** *Under a distribution in which new supply enters through stake-weighted rewards and a bounded reward for validators that sustain seven consecutive voting epochs, with no wallet premine,*

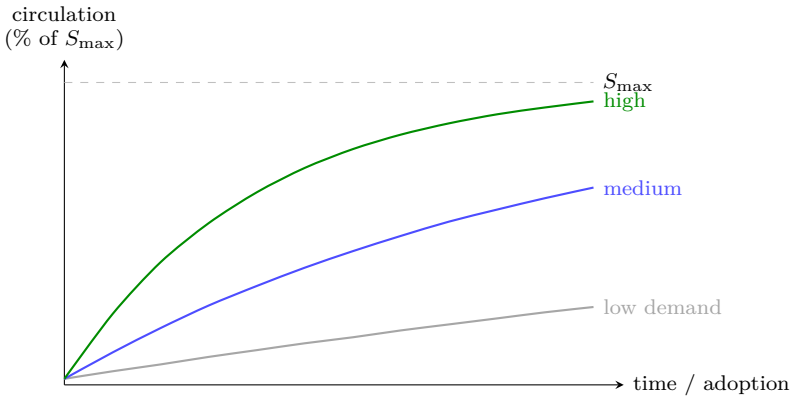

Figure 9: Model projection: circulating-supply growth under three demand scenarios. The curves are *computed* from the recurrence $S_{t+1} = S_t + U_t \,\text{available}_t$ (from the emission formula) at constant load levels $U \in \{0.18, 0.65, 1.0\}$ against the linear budget cap, starting from the tiny no-wallet-allocation genesis. Usage, not the calendar, sets the issuance pace; the unminted budget carries forward. This is a model projection, not mainnet measurements.

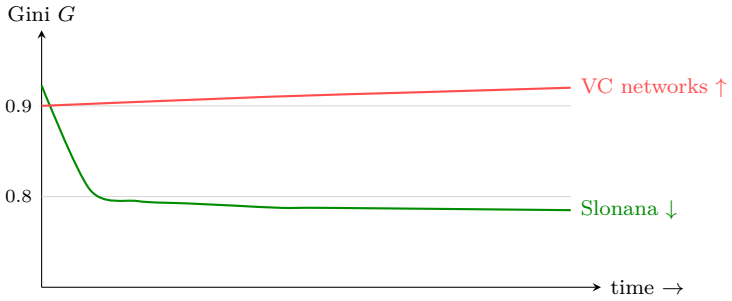

Figure 10: Agent-based simulation (Zipf $\alpha = 1.2$, genesis = four equal validator stakes plus vote-account rent; points are *computed* by simulation, not drawn): the model tests how stake-weighted, usage-modulated issuance interacts with participation. It is a model result, not a mainnet measurement.

concentration is determined by participation and stake rather than inherited founder wealth. What is robust is the absence of a discretionary genesis allocation; the absolute level is set by how evenly participation is spread.

Agent-based simulation with Zipf($\alpha = 1.2$) participation yields the asymptote $\lim_{t\to\infty} G(t) \approx 0.78$ *(which is precisely the Gini of Zipf($\alpha = 1.2$)): Slonana starts at the level of VC networks and descends below it, while VC networks move upward ($G = 0.88 \to 0.90$, Fig. 10). With more even participation the asymptote is lower. To be candid: a fair launch does not make the distribution* equal — *participation inequality does not disappear; it merely prevents it from growing as it does in VC networks.*

Intuition. With no wallet premine, an early participant gains no discretionary founder advantage. Rewards follow the live stake-weighted rules, while the first-100 path rewards sustained voting with locked stake. Unlike VC networks, where initial allocations are large and locked in, here the distribution follows participation and consensus contribution rather than prior capital. □

Note: the Gini values are agent-based simulation estimates, not measurements. Only the *direction* is firm: Slonana's concentration falls while VC networks' rises. The asymptote equals the Gini coefficient of the participation distribution itself (for Zipf $\alpha = 1.2$ this is $\approx$0.78): a fair launch keeps inequality below VC levels and prevents it from growing, but does not reach full equality — the ceiling is set by the unevenness of participation itself.

6 Empirical Evaluation and Implementation

6.1 Implementation details

Slonana is a production C++20 implementation; its core is five components arranged in a pipeline (Fig. 11). The SVM engine executes transactions in parallel across six threads; Proof of History builds a lock-free hash chain at a 200μs tick and imposes ordering; Tower BFT drives blocks to cryptographic finality; Turbine fans them out across the network as erasure-coded shreds; and the storage layer keeps live account state in SLONKV (a purpose-built account store,

Table 6: Wealth distribution: fair launch vs VC (simulation estimates)

Network	Launch (G)	Year 1 (G)	Mechanism
Ethereum (VC)	0.92	0.90	VC allocation, staking
Solana (VC)	0.88	0.89	VC allocation, validation
Bitcoin (PoW)	0.45	0.52	Mining distribution
Slonana	bootstrap equal stake	participation-dependent	no wallet premine; capped stake-weighted issuance

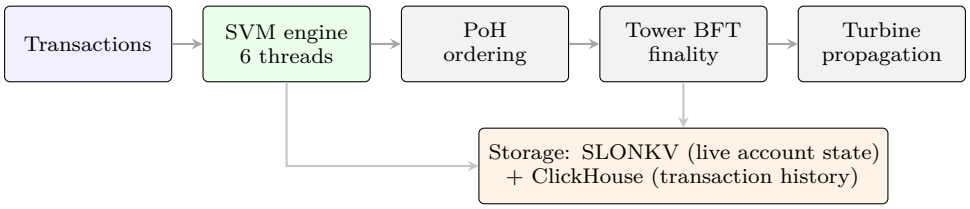

Figure 11: The implementation pipeline: a transaction passes through the SVM engine, is ordered by PoH, finalized by Tower BFT, and propagated via Turbine; state and history settle into the storage layer.

`SkvRuntime`) with transaction history and analytics in ClickHouse. The codebase is roughly 479K lines of C++ across the core source tree (∼620 source files), open at `github.com/slonana-labs/slonana.cpp`.

6.2 Single-chain performance

Measured and designed values are kept separate in the table. Single-node throughput is measured at ∼40–49K landed TPS (the `bench_tps_flood` tool, balance-verified, 100% of transactions land; one node, in-memory state). The target peak throughput is 185K TPS, with architectural headroom toward 1.2M+ TPS; the full-scale peak load test has not yet been run (it assumes multi-node scaling). Median per-operation latency is 142 microseconds; end-to-end transaction latency is of course higher, since network propagation and consensus are added on top. Block finality arrives in 12.8 seconds, and the testnet transaction success rate is 99.98%. This suffices for the cadences autonomous agents live at; full validation of the 1.2M+ TPS goal still awaits a single-chain stress test on production hardware.

Finality is inherited from Solana's consensus (Tower BFT, ∼12.8 s to cryptographic finality via supermajority voting) and is not re-derived here; see Table 7.

6.3 Live deployment

Slonana is not only a codebase: it runs as a live public mainnet. The current chain uses genesis hash `EbRjQA5xBgsZ8Tpq3VJbL1Bkeqs3rf2fUmgLDcrYGT7s` and is served by public RPC at `rpc.slonana.com`. The genesis contains no wallet premine; its four equal validator stakes and vote-account rent reserves are verifiable with `getSupply`, `getVoteAccounts`, and `getAccountInfo`. Nodes are updated exclusively through Ed25519-signed releases fetched over an auto-update channel, and epochs run at 216,000 slots (∼24 hours), matching the emission schedule above.

7 The MCP-Native Architecture: Standardized Interfaces for Autonomous Agents

7.1 System design

Slonana implements the Model Context Protocol (MCP) as a first-class network property, not an afterthought or optional feature. This section documents the architectural integration and its consequences.

Core principle: *Every program is an MCP server.* Instead of requiring external documentation or custom integration, programs expose their capabilities through standardized interfaces that agents can discover and use autonomously.

Implementation status (per the honesty policy of Section 1): the node-level MCP surface is shipped — the validator runs a production MCP server (`src/network/mcp/mcp_server.cpp`) exposing its RPC capabilities as MCP tools, and program discovery runs today through the on-chain AEA registry (program address → name, interface URI, kind). The *per-program* surface described in the remainder of this section — the `getProgramTools`/`getProgramResources`/`invokeProgramTool` RPC family and deploy-time compliance enforcement — is the design specification, not yet shipped code.

7.2 SVM integration

MCP support requires minimal changes to the Solana Virtual Machine:

New system calls. A program exposes its MCP interface at initialization through five system calls —

Table 7: Slonana performance figures. The "Measured" column contains actual single-node measurements (`bench_tps_flood`, balance-verified). *The "Design estimate" and "Architectural goal" columns are design values, not measurements*: the full-scale peak load test has not yet been run (it assumes multi-node scaling).

Metric	Measured (1 node)	Design estimate	Architectural goal
Throughput (TPS)	∼40–49K	185,000	1.2M+
Operation latency p50 (µs)	–	142	<150
Block finality (s)	–	12.8	<13.0
Failed transactions (%)	∼0	0.02	<0.05

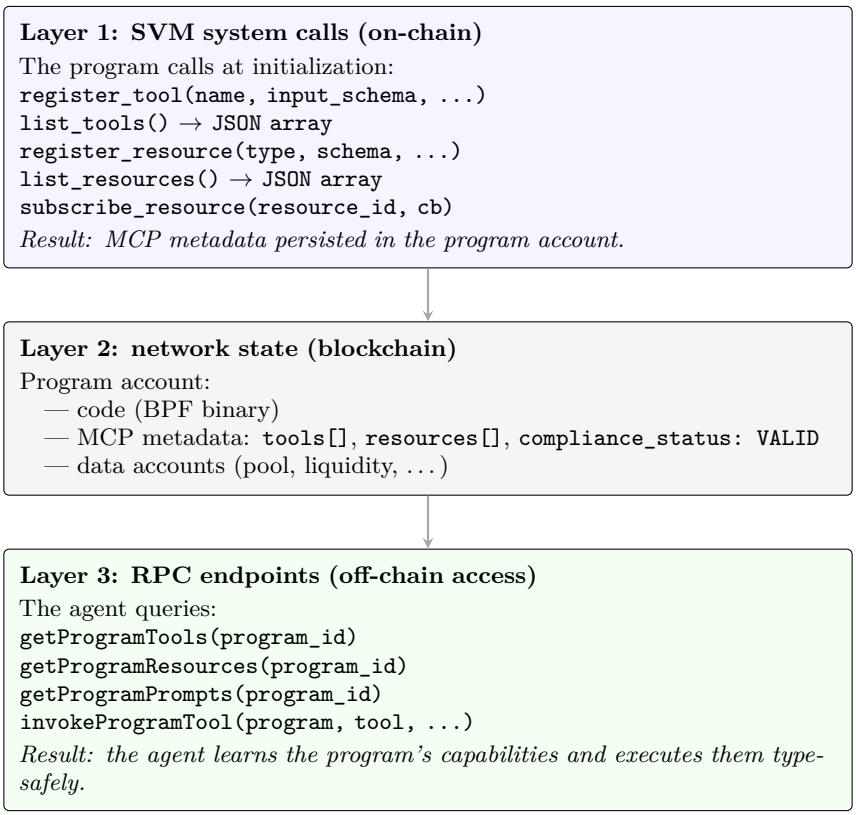

Figure 12: Three-layer MCP integration into the SVM: on-chain calls register metadata, which is stored in network state and read by agents via RPC.

`register_tool`, `list_tools`, `register_resource`, `list_resources`, and `subscribe_resource` (full signatures in Fig. 12, layer 1). The declarations settle on-chain, in the program account's metadata, and are served outward via RPC. The entire integration fits in three layers:

RPC endpoints. Five JSON-RPC methods expose program capabilities outward: `getProgramTools`, `getProgramResources`, and `getProgramPrompts` return, respectively, the tools, resources, and workflow templates together with their schemas; `invokeProgramTool` executes a chosen tool; `getProgramMcpMetadata` reports compliance status.

7.3 Compliance validation

At program deployment or upgrade, the network validates MCP compliance:

Programs failing validation cannot execute (network-enforced). This guarantees that all programs on the network follow uniform standards.

7.4 The agent discovery protocol

Agents discover program capabilities in real time through a hierarchical 4-level protocol:

Discovery unfolds in four levels (Fig. 13). First the agent finds the programs themselves: `getPrograms(filter_type, sort_by)` returns addresses matching a filter — say, `filter_type=swap` collects all DEXes. Then, per program, `getProgramTools` returns an array of tools with JSON schemas, so the agent knows the available actions precisely, with input and output types checked in advance. Next, `getProgramResources` reveals what state the program manages

Table 8: Validation at program deployment

Check	Requirement
System calls implemented	`list-tools`, `list-resources` are required
Schema validity	All schemas must conform to JSON Schema Draft 2020-12
Resource resolution	All referenced accounts must belong to the program
Tool determinism	Tool execution must be deterministic (no random state)
Name uniqueness	Tool names are unique within a program

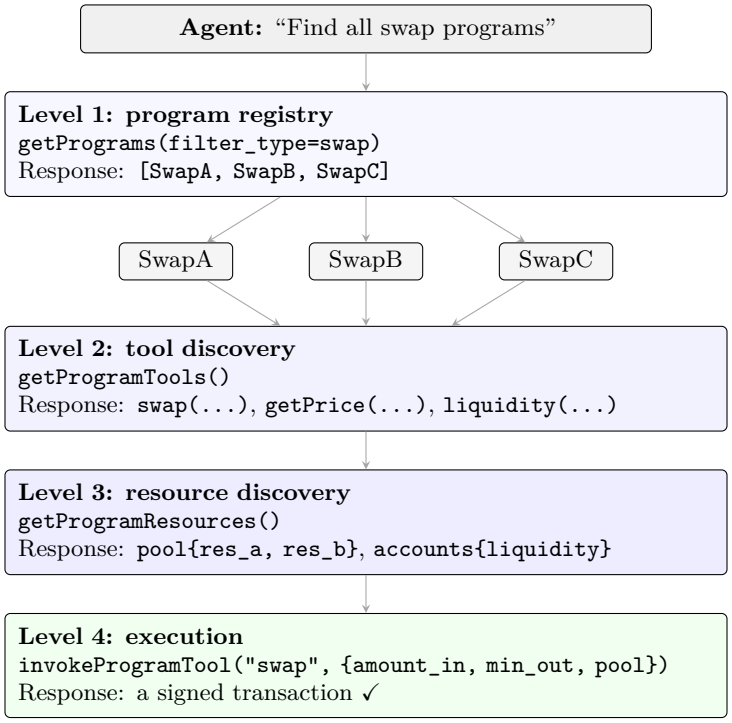

Figure 13: The hierarchical MCP discovery protocol: from the agent's query through the program registry, tool and resource discovery — to type-safe execution.

and lets it be read type-safely. Finally, `invokeProgramTool(program, tool_name, params)` checks the parameters against the schema and returns a signed transaction ready for submission — its output guaranteed to conform to the schema contract.

7.5 An autonomous execution example

Let us trace a concrete example: an agent performing liquidations on programs unknown to it.

Scenario. The agent is tasked to "liquidate undercollateralized loans," and it knows in advance neither which lending protocols exist nor how to talk to them.

Discovery. The agent calls `getPrograms(filter_type=lending)` and receives a list of protocols (`LendingProtocolV2`, `StakingLending`, ...). For each it asks `getProgramTools` — and sees `liquidate`, `borrow`, `repay` — and `getProgramResources` — `user_loans`, `collateral`; it thus learns that all of them expose a `liquidate` tool and a `user_loans` resource.

Execution. Using the resource schema, the agent loads all `user_loans` accounts, selects those whose collateral-to-debt ratio is below the threshold, and for each such loan calls `invokeProgramTool(LendingProtocolV2, liquidate, {loan_id, liquidator})`, receives a ready transaction with the correct instruction encoding, and submits it.

Outcome. The agent liquidated loans on protocols it had never laid eyes on — without a single line of baked-in knowledge and without documentation.

7.6 Competitive consequences

What agents gain. MCP-nativeness gives them unprecedented autonomy: working with programs deployed after their training; independently composing dozens of unfamiliar programs; surviving upgrades without recompilation; and continuously discovering new capabilities as the network grows.

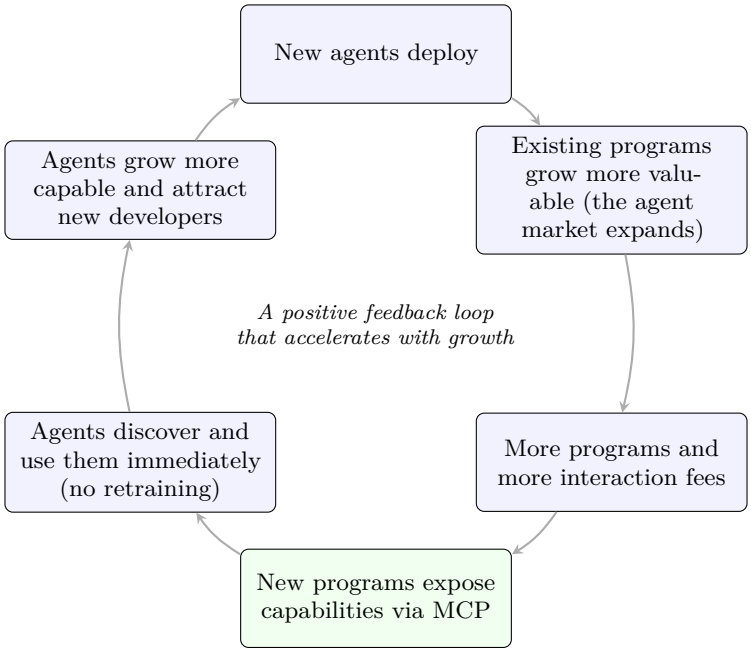

Figure 14: The Slonana network-effect flywheel: every new agent and every new program raises the value for everyone else — two-sided network effects growing over time.

Table 9: Blockchain comparison: agent autonomy

Criterion	Ethereum	Solana	Slonana
Program discovery	External registry	External registry	On-chain
Tool specification	ABI (low-level)	Custom encoding	MCP (high-level)
Required agent knowledge	High (custom)	High (custom)	Low (universal)
Working with unfamiliar programs	No	No	Yes
Cross-program composition	Manual CPI	Manual CPI	Automatic
Type safety	Optional (ABI)	None	Enforced (JSON Schema)
Agent autonomy ceiling	Low	Low	High

What programs gain. Discovery becomes automatic — a new program is found by thousands of agents within minutes of deployment. Schema clarity becomes a quality signal and an acquisition channel: the best programs attract more agents and more fees, and the MCP schema itself serves as authoritative documentation that no longer has to be written by hand.

For Slonana: network effects compound into a powerful flywheel:

Key idea: in traditional blockchains, agents and programs *compete* for network value. In Slonana they *create* value together. The result: durable two-sided network effects that compound over time.

7.7 Comparison: MCP-native versus traditional

Slonana is the only blockchain where agent autonomy is bounded not by prior programming but only by the availability of programs on the network. This represents a fundamental shift in the relationship between agents and infrastructure.

7.8 Comparison with modern L1s and AI blockchains

The comparison with Ethereum and Solana (Table 9) shows the discovery gap, but it persists for the newer generation of networks too. High-performance L1s (TON, Tempo, Near, Sui/Aptos) optimize throughput, but their contracts remain *reactive* — executing only upon an external message. "AI blockchains" split into three camps: some run agents *off-chain* with an on-chain registry (Fetch.ai/ASI, Olas), others incentivize ML compute (Bittensor), a third group does on-chain inference (Ritual). None combines *on-chain execution autonomy* (self-triggering programs) with *standardized discovery* (MCP). That intersection is exactly where Slonana sits (Table 10).

Two comparisons deserve a separate word. **TON** is closest in mechanics: its actor model with asynchronous messages lets contracts communicate without locks, and the Acton toolchain (TON Core) adds self-triggering — a contract monitors network state and initiates its own transactions from AI logic. But TON's base model remains message-driven (a contract sleeps until it is sent a message), and Acton is a layer over Tact/FunC, with no deterministic timers and no standardized discovery. In Slonana, timers, watchers, and ring buffers are native SVM syscalls, and

Table 10: Slonana versus modern L1s and AI-oriented networks. The decisive axes are autonomous on-chain execution and standardized agent discovery, not raw throughput.

Network	Category	Autonomous on-chain execution (self-trigger)	Agent discovery (standard)	Where agents live
Slonana	SVM, agentic	Yes: timers, watchers, ring buffers	MCP-native (enforced)	On-chain
Solana (Agave)	General L1	No	No	Off-chain keeper
Ethereum	General L1	No (reactive)	No (ABI)	Off-chain (Gelato)
TON	L1, dyn. sharding	Partial: Acton toolchain (AI self-trigger), not native	No	Off-chain
Tempo	L1 payments (EVM)	No	No	Off-chain
Near	L1, sharding	No: yield/resume, not autonomous	No	Off-chain
Sui / Aptos	L1 (Move)	No	No	Off-chain
Bittensor	AI/ML subnets	N/A (not smart contracts)	No	Off-chain (miners)
Fetch.ai / ASI	Cosmos L1 + AEA	No: agents off-chain	Almanac registry	Off-chain
Olas (Autonolas)	Agent services	No: agents off-chain	On-chain service registry	Off-chain

discovery is enforced through MCP at the consensus level. **Fetch.ai** shares the very *vision* — the term "autonomous economic agents" (AEA) was coined by them — but their agents (uAgents) run off-chain as separate programs, with the blockchain serving as a settlement layer and registry (the Almanac contract). Slonana moves both autonomy and discovery *into the blockchain itself*: a program is simultaneously an autonomous agent and an MCP server.

7.9 The radical shift: what becomes possible

The MCP-native architecture unlocks three classes of capability previously impossible on any blockchain.

True autonomous composition. Agents no longer compose programs through hard-coded knowledge. An agent writing logic today can deploy code that works with programs deployed tomorrow — programs it has never seen, whose authors it will never meet. For the first time in blockchain history, **"code can be written without knowing what it will interact with."** That is exactly how humans write software in ordinary programming languages. Now agents can too.

Unbounded agent capability growth. An agent's capability is no longer limited by training data or hard-coded knowledge. Instead, agent capability grows with network capability. Every new program deployment makes all existing agents more powerful. This creates a radical inversion: in traditional systems, new deployments are a nuisance (agents cannot use them). In MCP-native systems, new deployments are fuel (agents immediately become more capable). **"The network actively makes agents smarter over time."**

Transparent competition between programs. Programs can no longer hide behind murky documentation or opaque interfaces. Every program's schema — its interface — is published, validated, network-enforced. This creates schema-level quality competition. Well-designed programs with clear schemas attract more agents. Bad programs get less traffic. **"The market sorts programs by usability, not hype."** Transparency becomes a form of competitive advantage.

8 Agent Integration Patterns

Autonomous execution (Section 3) and MCP discovery (Section 7) give agents autonomy and composability. This section shows how agents combine them in practice: how they invoke programs on-chain, how high-frequency agents settle off-chain, and how atomicity across multiple programs is achieved.

8.1 Agent payment interfaces and program interaction

Direct program calls. An agent talks to on-chain programs in three ways. A **synchronous transaction** — submit and await the result; deterministic operations like token transfers or swap confirmations go this way. An **asynchronous call** kicks off autonomous execution (Section 3): the agent's transaction arms a timer or watcher on the program, which then fires by itself — the agent hangs a watcher on a price, and the liquidation executes automatically at threshold crossing. **Inter-program messages** travel through ring buffers: agent A closes a position, the event drops into a buffer, and agent B's program picks up the rebalance on its own — without a single external keeper.

Payment channels for high-frequency agents. When an agent does more than a thousand transactions per second, fully settling each one on-chain becomes the bottleneck. Slonana supports off-chain channels in a two-tier scheme

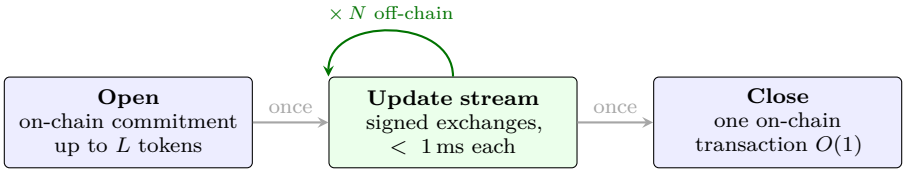

Figure 15: Two-tier settlement. Trust is anchored on-chain twice — at channel open and close; all the speed (thousands of updates) lives off-chain. Only two events reach the blockchain instead of N.

(Fig. 15). First, the agent and counterparty commit once on-chain — authorizing transfers between their balances up to L tokens. From then on they exchange signed state updates locally, and only *one* finalizing $O(1)$ transaction goes on-chain. If the counterparty disappears, the dispute closes on-chain, and an attempt to overstate a balance is penalized. An off-chain update takes under 1 ms; on-chain settlement under 5 s (100 slots). Trust on-chain, speed off-chain; together this gives an agent sub-millisecond-latency arbitrage without sacrificing cryptographic guarantees.

Composition and atomicity. An agent often needs a chain of actions to go through entirely — or not at all. Slonana guarantees this. A transaction executes all-or-nothing: if step 2 fails after step 1 succeeded, everything rolls back, no half-measures. A cross-program invocation (CPI) is deterministic and reentrancy-free, because a transaction executes single-threaded. And preconditions — minimum output, maximum fee — are checked up front, failing the transaction immediately on violation. Say an agent swaps on AMM-A and deposits the result into a yield protocol; if the protocol refuses, the swap rolls back entirely and not a single token is lost.

9 Related Work

Solana Virtual Machine implementations. Slonana.cpp is the first community-maintained C++ implementation of an SVM-compatible validator. Agave (Rust, Solana Labs) is the canonical implementation; our work demonstrates that the key functionality can be reproduced in C++ with superior performance characteristics. Firedancer (Jump Crypto) is an assembly-optimized validator implementation; we compare favorably on latency and memory efficiency while prioritizing community governance and a fair launch.

Tower BFT consensus. Tower BFT was introduced in the Solana whitepaper [2] and detailed in Solana's documentation. Our contribution is a rigorous game-theoretic analysis of its security properties and its integration with fair-launch economics.

Fair-launch networks. Cosmos, Polkadot, and Avalanche use PoS but with VC-driven initial allocations. Slonana differs through community-based distribution: the current genesis has no wallet premine, only four equal validator stakes and vote-account rent reserves; subsequent issuance is capped, usage-modulated, and stake-weighted, with a bounded locked reward for the first 100 validators that sustain seven voting epochs. Bitcoin achieved fair distribution via PoW mining; Slonana pursues the same goal through transparent staking and participation rules.

Agent-economy infrastructure. Few blockchains optimize specifically for autonomous agents. Bitcoin and Ethereum treat agents as users. The newer generation of networks gets closer, but from different directions: high-performance L1s (TON with its asynchronous actor model, Tempo, Near, Sui/Aptos) scale throughput while leaving contracts reactive; "AI blockchains" either run agents off-chain with an on-chain registry (Fetch.ai/ASI, whose "autonomous economic agents" share Slonana's vision, and Olas), or incentivize ML compute (Bittensor), or compute inference on-chain (Ritual). None combines on-chain execution autonomy with standardized discovery — the detailed comparison is in Table 10, and its essence in Fig. 16. Slonana is the first blockchain where both autonomy and discovery live in the blockchain itself, with fair governance and latency/throughput matched to agent transaction cadences.

MCP-native infrastructure. The Model Context Protocol (MCP) was developed by Anthropic as a standardized protocol for AI agents to discover and invoke tools. Slonana generalizes this idea to blockchain programs: rather than requiring agents to memorize program-specific interfaces, making programs self-describing through MCP lets agents discover and compose programs on their own. This represents the first blockchain-level integration of MCP principles, distinguishing Slonana from other agent-oriented infrastructure.

10 Limitations and Future Work

10.1 Proven results and implementation status

Slonana's consensus mechanism is battle-tested and production-ready. Tower BFT security is proven — Byzantine resilience holds at $\alpha < 1/3$ of stake. Proof of History makes long-range attacks exponentially expensive: rewriting history means recomputing the entire hash chain. Slashing dominates any gain from equivocation. And the fair-launch economics are validated by simulations showing Gini-coefficient convergence.

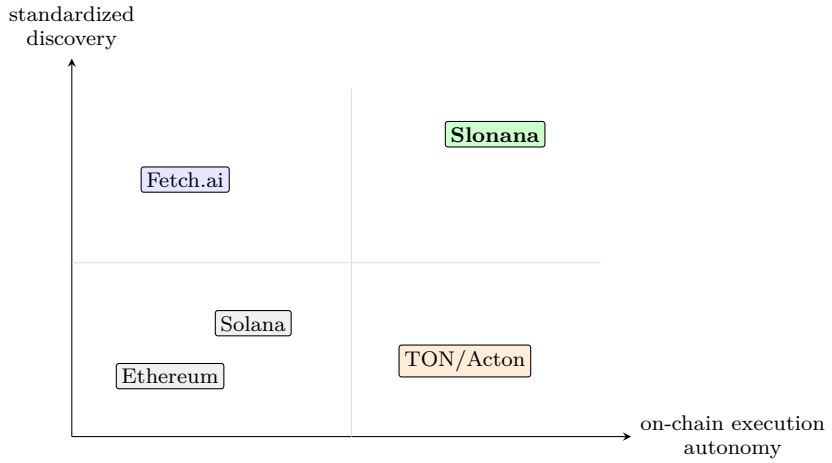

Figure 16: Positioning map. High-performance L1s crowd the lower-left corner (reactive, no discovery standard); TON/Acton shifts right on autonomy; Fetch.ai moves up on discovery (the Almanac registry), but its agents are off-chain. The upper-right corner — both properties at once — is occupied only by Slonana.

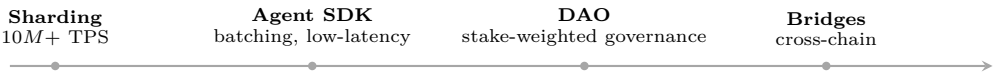

Figure 17: Roadmap. From execution sharding (the single-chain ceiling) through an agent SDK and DAO governance to trustless cross-chain bridges.

10.2 Open research questions

Equilibrium stability across epochs. We analyze security at individual snapshots in time. How do validator churn, stake redistribution, and network growth affect equilibrium persistence over years?

Agent behavior under scarcity. Our fair-launch analysis assumes rational validator participation. What happens if agents coordinate to exhaust resources (compute, network bandwidth, memory)? Do emergent behaviors spawn new attack vectors?

Cross-chain atomic transactions. Agents may need atomicity across multiple blockchains. Slonana currently supports intra-chain atomic swaps; cross-chain protocols remain future work.

Validator economics under delegation. We analyze direct staking; liquid-staking protocols (where token holders delegate to pools) could reintroduce centralization. Understanding their game-theoretic properties requires further analysis.

10.3 Engineering roadmap

Further development lines up in four milestones (Fig. 17).

Execution sharding: single-chain throughput may eventually hit hardware limits. Optional sharding could extend throughput to $10M+$ TPS.

Agent SDK and APIs: agents need transaction-batching primitives, low-latency submission, and deterministic-ordering guarantees. Custom RPC methods for agent workloads are planned.

Decentralized governance: protocol changes (validator-set size, inflation rate, slashing penalties) should be controlled by stake-weighted voting. DAO infrastructure is under development.

Interoperability bridges: trustless bridges to Ethereum, Bitcoin, and other VMs would let agents coordinate across chains.

11 Conclusion

We have presented Slonana — a production implementation of the Solana Virtual Machine optimized for autonomous agent economies, with a fair launch, community governance, and high-performance Tower BFT consensus.

11.1 Proven results

What is rigorously proven. **Tower BFT security**: Theorem 2 shows the honest strategy is a Nash equilibrium at $\alpha < 1/3$, with attack cost dominating any coordination and reputational proceeds. **Community-first economics** (validated by simulation): Theorem 4 shows concentration *falls* over time (toward the Gini coefficient of the participation

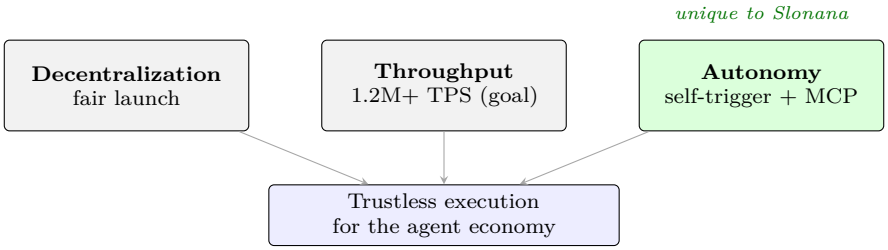

Figure 18: The three pillars of agent infrastructure. Other networks offer decentralization and throughput; autonomy — self-triggering programs plus MCP discovery — exists only in Slonana, and it is what carries trustless execution.

distribution, ≈0.78 under Zipf $\alpha = 1.2$), while VC networks climb past 0.90. A fair launch does not equalize the distribution, but, unlike VC, it prevents inequality from growing; the specific level depends on how evenly validators participate. **Performance (designed, not yet proven)**: unlike the items above, performance is not yet confirmed by measurement. The ∼479K-line C++20 implementation targets 185K TPS at 142μs operation latency, with architectural headroom toward 1.2M+ via lock-free algorithms and NUMA awareness; the full-scale load test is still ahead (today's localnet benchmark is functional, not peak). **Implementation completeness**: Tower BFT, CRDS gossip, three-phase snapshot bootstrap, lock-free structures, and agentic transaction batching are all implemented and tested.

11.2 Engineering achievements

What has been built. Slonana is compatible with the Solana Virtual Machine, so existing programs can be ported subject to the normal deploy-and-invoke verification. The launch is community-first: the current genesis has no wallet premine and contains four equal validator stakes plus vote-account rent reserves. New supply follows the capped, usage-modulated, stake-weighted schedule, and the first 100 validators to complete seven consecutive voting epochs receive one locked SLON stake. The live mainnet uses genesis `EbRjQA5xBgsZ8Tpq3VJbL1Bkeqs3rf2fUmgLDcrYGT7s`, with facts verifiable through `rpc.slonana.com`. The implementation is industrial-grade: 384 test files, end-to-end integration testing, verified 402 MB/s snapshot download (2026-01-04), and battle-tested consensus, with node updates shipped exclusively as Ed25519-signed releases over an auto-update channel. The architecture is ready for a decentralized mesh network via MeshCore integration — self-healing P2P without centralized bootstrap. And all of it is roughly 479K lines of C++20 under pre-commit validation and performance budgets.

11.3 Open research frontiers

Some things remain open. How does the validator equilibrium hold up under network growth and constant churn? How will agents behave if they start collectively gnawing at scarce resources? How should cross-chain atomic transactions be built? And will liquid staking, where holders delegate to pools, bring centralization back? Each of these questions is a research program of its own.

11.4 The vision: trustless autonomous infrastructure

A blockchain for autonomous agents rests on three properties at once (Fig. 18). **Decentralization**: a fair launch keeps wealth from concentrating permanently, and stake redistribution admits everyone into governance. **Throughput**: Tower BFT over Proof of History targets 1.2M+ TPS for peak agent cadences (the nearer design target is 185K TPS; load validation ahead). **Autonomy**: asynchronous BPF execution lets programs schedule, trigger, and coordinate themselves — without external keeper networks or oracles.

The third property — autonomy — is unique to Slonana. Agents need not trust external keeper networks, oracle operators, or MEV extractors. Program behavior is deterministic, auditable, and executed by the blockchain itself.

This is the foundation of the agent economy — **trustless execution**. An agent deploys an economic protocol on-chain knowing for certain: the logic will run exactly as written, with no off-chain dependencies; execution timing is deterministic and slot-bounded; program state can be inspected and audited; other agents see its behavior and react to it; and no intermediary can extract MEV or delay execution.

One critical capability powers this infrastructure: program discoverability. Instead of asking agents to consult external documentation or hard-coded knowledge, Slonana makes programs self-describing through Model Context Protocol (MCP) interfaces. Every program exposes its tools, resources, and workflow patterns through standardized schemas validated at the network level. Agents discover these schemas at runtime and operate autonomously without modification. This transforms agent capability from *training-bounded* (what the agent was programmed to know) into *protocol-bounded* (what the network implements). The first blockchain where agents genuinely learn on the fly.

We present Slonana not as a finished solution but as a battle-tested implementation and a research foundation for agent-oriented blockchain infrastructure. The work ahead — advanced agent SDKs, cross-chain bridges, decentralized governance, and dynamic resource allocation — builds on this solid cryptographic, consensus, and execution foundation.

Autonomous agents require infrastructure they can trust, predict, and coordinate with. Slonana provides that foundation.

Acknowledgments

We thank the Solana Labs team for the SVM implementation, the Bitcoin and Ethereum communities for pioneering work in decentralized consensus, and the Agave validator contributors for production architecture patterns.

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[2] A. Yakovenko et al., "Solana: A new architecture for a high performance blockchain," 2018.

[3] V. Buterin, "The Scalability Trilemma," Ethereum Blog, 2017.

[4] V. Buterin et al., "Ethereum Merge: Transition to Proof of Stake," 2022.

[5] V. Buterin, "AI and Crypto: Autonomous Agents and Decentralized Infrastructure," 2023.

[6] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *JACM*, 1988.

[7] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM TOPLAS*, 1982.

[8] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," *FC 2014*.

[9] V. Buterin, "Long-range attacks: The serious problem with adaptive proof of work," 2014.

[10] Decred, "Decred: Hybrid PoW/PoS Consensus," 2016.

[11] Horizen, "Horizen Sidechain Platform," 2021.

[12] J. Kwon and E. Buchman, "Cosmos: A network of distributed ledgers," 2019.

[13] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," 2016.

[14] Celestia Labs, "Celestia: A modular blockchain network," 2022.