YELLOW PAPER

The full technical specification of a payments-only Layer-1 blockchain. Technologies are described along the transaction lifecycle; every choice is argued and defended against contemporary alternatives — with formulas, diagrams, and honest caveats. For engineers deciding whether to join the team, and for analysts assessing the network's prospects.

Xync Network L1 · v1.0, July 2026

Contents

0. System model. Why "payments only" is a superpower

The network is a set of n validators, at most f of them Byzantine (arbitrarily malicious: they lie, stall, collude), n = 3f+1. The network is partially synchronous: messages arrive within an unknown but finite delay δ. Primitives: ed25519, SHA-256. Quorum q = 2f+1; any two quorums intersect in at least f+1 nodes — hence at least one honest one:

$$|Q_1 \cap Q_2| \geq q + q - n = (2f+1) + (2f+1) - (3f+1) = f+1 > f$$

The domain is deliberately reduced to money: **send**, **request**, **exchange** (255 currencies + XYNC) — no virtual machine, no arbitrary shared state. This is not a "platform-minus" — it is the source of every advantage we have, because it unlocks a theorem unavailable to general-purpose chains:

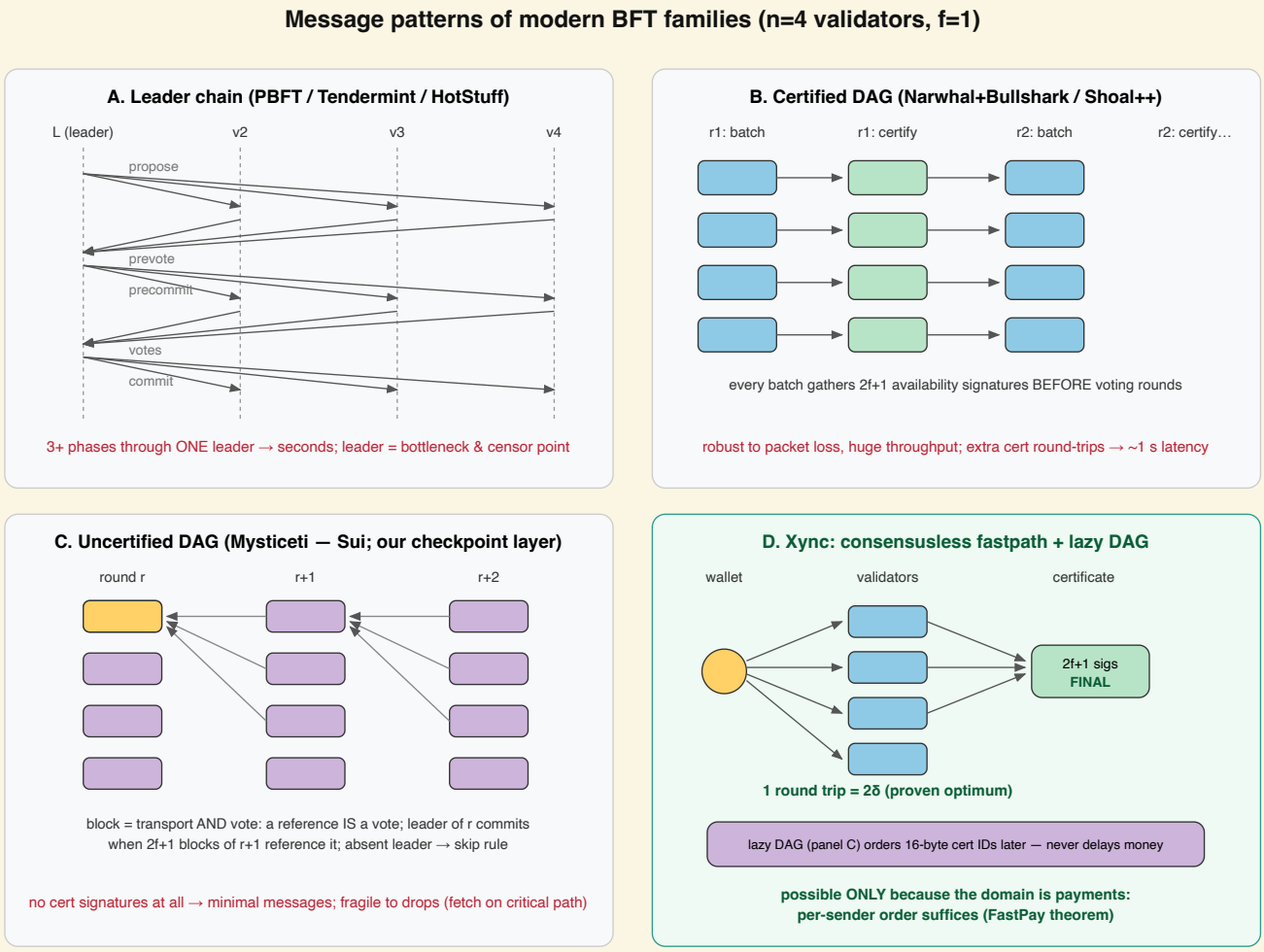
The domain theorem (FastPay, 2020). If every account is controlled by a single owner, a payment system needs no total order of transactions — a partial order per sender suffices. *Intuition:* only the spends of one account can conflict, and their order is set by the owner's own counter (seq). Transfers by different people commute: `apply(tx_A); apply(tx_B) ≡ apply(tx_B); apply(tx_A)` — the balances come out identical.

Corollary: consensus (choosing one global order) is **redundant on the payment critical path**. Finality is achievable in one round trip, and 2δ is the proven lower bound of any BFT finality (pod, 2025). General-purpose chains must order everything (smart contracts share state), so the theorem is closed to them — they compete at making consensus spin faster, while we simply removed it from the hot path.

Where the layer boundary runs. The theorem covers exactly those operations in which every account touched has an owner: a transfer, a money request, and a P2P swap (§14.2 — two legs, each signed by its owner; such swaps commute too and live in the fastpath). Anything that mutates **shared** state — pool reserves, liquidity-provider shares, issuer positions — has no owner: two independent swaps against one pool do not commute, because they read and write the same reserves. Such operations therefore travel exclusively through the lazy DAG, in a single order (§8, §14.4). That is the honest price of multi-currency: a pool swap finalizes in ~1–2 s, not 0.3 s. The invariant "shared mutable state changes only inside the commit, in DAG order" is machine checked (§14.7).

1. The grand consensus comparison 2020–2026

The choice of agreement algorithm is the single most consequential architectural decision in any L1: it fixes payment latency, the cost of a byte, and who is able to halt or censor the network. What follows is therefore not a list of fashionable names but a dissection of every living family as of July 2026 — how each one works, what it pays for its speed, and why in a payment network it loses to the pairing we chose. Proof-of-Work does not enter the comparison at all: its probabilistic finality is measured in tens of minutes and its energy cost is irreconcilable with a node running on a phone. Either property disqualifies it before the contest begins.



1.1. How each family works

A. Leader chains (PBFT 1999 → Tendermint/CometBFT → HotStuff → Jolteon/MonadBFT). One leader proposes a block; everyone votes in 2–3 phases (prevote/precommit); leader rotation on timeout. HotStuff made votes linear (O(n) instead of O(n²)) and pipelinable; MonadBFT (2025) added speculative finality and tail-fork resistance. The family's fundamental limits: every byte passes through the leader (a throughput bottleneck and a censorship/MEV point), and finality is several sequential phases across the globe: 1–6 s in practice. For a network promising an irreversible payment faster than a person can put the phone back in their pocket, this is a disqualifying price.

B. Certified DAGs (Narwhal+Tusk/Bullshark 2022 → Shoal 2023 → Shoal++/Sailfish 2024 → Raptr 2025). Narwhal's breakthrough: separate data DISSEMINATION from ORDERING. Every validator publishes batches in parallel, collects 2f+1 availability signatures per batch (a batch certificate), and consensus then orders the small certificates — hence the monstrous throughput (100k+ TPS across the family). Bullshark builds waves with anchors on top; Shoal++ squeezed latency to ~0.7–1 s with three parallel DAGs and leader reputation; Raptr (Aptos, 2025) added an optimistic path while keeping robustness: at 1% packet loss latency grows only ~15%. The family's price: every batch waits through a certification round trip (an extra RTT and signatures) BEFORE it can be ordered.

C. Uncertified DAGs (Mysticeti 2024 → Mahi-Mahi 2024). The bold move: drop certificates entirely. A validator's block is transport AND vote at once: referencing a block IS a vote for it (implicit voting). The leader of round r commits when 2f+1 blocks of round r+1 reference it; finality in 3 sub-rounds (~0.4–0.65 s in Sui production, v2 cut another 25–35%). An order of magnitude fewer signatures — ideal for compute. The weakness, independently confirmed (arXiv 2025/877, Autobahn): without availability certificates, data fetching lands on the critical path — 0.05% packet loss can inflate latency 10×, and a single slow validator adds two message delays per round.

D. Hashgraph (2016; opened as Hiero/Apache-2.0 in 2024). Gossip-about-gossip: nodes exchange the history of their communication, and each computes virtual votes locally without sending them. Mathematically elegant, aBFT. Why we pass: finality in seconds (2–5 s on Hedera), redundant metadata gossip, no separation of data from ordering, and the family has shown no fresh latency results — it loses to both B and C on both of our priorities.

E. Alpenglow (Solana, 2025–2026). The replacement of PoH+TowerBFT: Votor — voting in 1–2 rounds (with ≥80% stake online, one-round finality of 100–150 ms), Rotor — dissemination via erasure-coded shreds through sampled relays. SIMD-0326 passed with 98%, testnet since May 2026, mainnet ~Q4 2026. This is the current summit of TOTAL ordering. The price: leader-centricity (MEV/censorship), the hardest networking engineering in the business (erasure codes, weighted relay selection), and zero battle testing.

F. Consensusless (FastPay 2020 → Linera → Sui fastpath → pod 2025). When every account an operation touches has exactly one owner, there is nobody to agree with and nothing to agree about: validators verify the transaction independently and sign it independently, and 2f+1 signatures *are* finality — one network round trip, no rounds, no blocks, no leaders. pod.network turned the idea industrial: ~150 ms, 300k+ TPS, a continuous stream of attestations instead of a chain of blocks. Sui runs the same mechanism as its fastpath for owned objects. The weakness is precisely one: there is no total order, so arbitrary smart contracts are impossible (we do not want them) and the rare network-wide events need a separate mechanism — for us, the DAG layer.

1.2. Summary table

Protocol	Family	Finality	Messages per tx	Status 2026	Disqualifier for Xync
Tendermint/CometBFT	A	1–6 s	O(n) through leader ×3 phases	prod (Cosmos)	latency, leader
HotStuff/Jolteon	A	~1–2 s	O(n) ×3	prod (Aptos until '25)	latency
MonadBFT	A	~0.8 s speculative	O(n)	prod (Monad)	speculation + leader
Hashgraph	D	2–5 s	gossip²	prod (Hedera)	latency, traffic
DAG-Rider/Tusk	B	4+ rounds	per-batch certification	research	latency
Narwhal+Bullshark	B	~2–3 s	same	was prod (Sui)	Sui itself moved to C
Shoal++	B	~0.7–1 s	same	prod (Aptos)	the extra certification RTT
Sailfish	B/C	1 RBC + 1δ	–	research 2024	maturity
Mysticeti (v2)	C	0.4–0.65 s	minimal	**prod (Sui)**	fragility to drops
Mahi-Mahi	C	~3 sub-rounds	minimal	research	async complexity
Raptr	B+opt	sub-second	optimistic path	testnet/prod '26	no UX fastpath
Alpenglow	E	**0.1–0.15 s**	Rotor shreds	mainnet ~Q4'26	leader, unproven
FastPay/pod	**F**	**2δ ≈ 0.15–0.4 s**	**n attestations**	prod (pod)	no order (not a minus for us)

1.3. Xync's decision: F + C = fastpath + lazy DAG

No single family fits, because a payment network hosts events of two utterly different kinds, and we serve each with its own layer.

Money travels the fast lane — with no consensus at all (family F, panel D of the figure). The wallet broadcasts the transaction to all n validators; each verifies and signs it independently; and the first 2f+1 signatures, gathered together, form a certificate of finality. One network round trip. No rounds, no leaders.

Everything else travels the slow lane — through the lazy DAG (family C, Mysticeti-lite). Registering an account, punishing a cheat, changing the validator set, cutting a checkpoint, distributing fees: for these, order genuinely matters (who got index #1000, whose fraud proof landed first). They are few, none of them is urgent, and not one of them delays a payment. "Lazy" is literal: with no events to order, the layer emits not a single byte.

Why the hybrid beats every pure alternative:

- *Pure C (Mysticeti-only):* a payment would wait 3 sub-rounds (0.5–1 s instead of 0.15–0.4) and — worse — inherit the packet-loss fragility ON THE MONEY PATH. In our hybrid the DAG is off the critical path: its 10× degradation under drops slows registrations, not payments.

- *Pure E (the Alpenglow road):* comparable milliseconds, but at the price of a leader (MEV/censorship), erasure-coding engineering and a year of hardening. We get the same speed from the domain theorem rather than from heroic engineering.
- *Pure B (Shoal++/Raptr):* the best liveness under loss, but every batch pays the certification RTT — while for payment UX we already HAVE certification (fastpath attestations), and ours doubles as finality.
- *Pure F (the pod road):* nothing deterministically executes registrations, fines, validator-set changes, or produces checkpoints for light clients — pod outsources these; a lazy DAG is cheaper for us.
- *D (Hashgraph):* loses on latency and traffic to each hybrid layer taken separately.

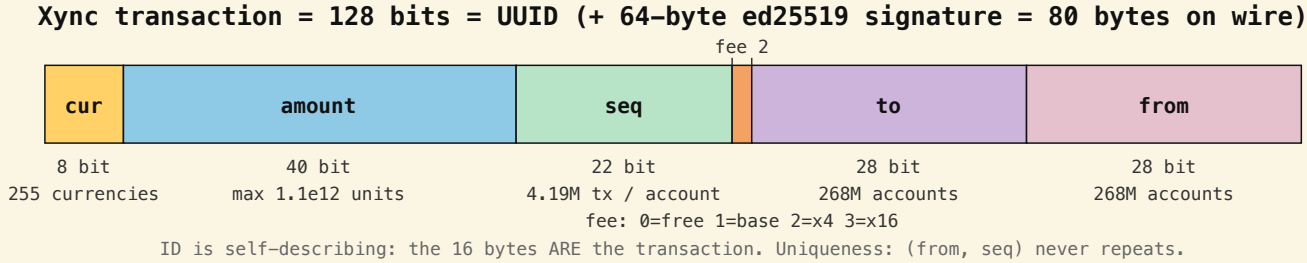
The honest price of the hybrid (also §17): (1) owner equivocation — even accidental, from a buggy wallet — is indistinguishable from an attack and gets punished; (2) any future shared-state feature must ride the slower DAG layer; (3) the fastpath demands seq discipline from wallets (the iron rule, §4).

2. Accounts and identity

An account: `{index, pubkey, seq, balances{cur: amount}}`. Indexes are compact (28 bits) and issued sequentially by a registration system transaction (~1 USD to the treasury — account-level Sybil resistance and funding, §11). #0 is the Xync operator: a legacy of the live XyncPay production; client ids are preserved 1:1 at migration (§12).

Defense of the choice. Hash addresses (BTC/ETH, 20–32 B) are free, but two of them wouldn't fit in half of a 16-byte transaction. A compact index = an expensive one-off registration + a cheap every single transaction; in a payment network traffic dominates. Side effect: short "beautiful" numbers — free gamification and status (remember ICQ). The $1 friction is softened by product means: a referral voucher (the treasury pays for invitees) and "the first incoming payment ≥$2 pays the recipient's registration back."

3. The transaction: 128 bits




```
currency 8 | amount 40 | seq 22 | fee 2 | to 28 | from 28  = 128 bits
ed25519 signature (DOMAIN_TX ‖ tx_id)                        = 64 bytes
total on the wire                                            = 80 bytes
```

The body **is** the identifier ("tx = UUID"): no separate hash, no serialization, zero overhead. Uniqueness is forever: the pair (from, seq) never repeats during the network's lifetime (the counter never resets).

Field	Rationale
currency 8	~100 fiats + ~150 coins; genesis registry with per-currency scale (USD:3, BTC:8, XYNC:6)
amount 40	up to $2^{40}-1 \approx 1.1 \cdot 10^{12}$ min. units: $1.1B per tx at scale 3
seq 22	4,194,303 outgoing per account for life: forever for individuals, ~11 years for a 1,000-tx/day pipeline; sub-accounts beyond (natural for bookkeeping)
fee 2	4 levels cover the real demand for priority: 0=free (§5), 1=base, 2=×4, 3=×16; multipliers are genesis parameters
to/from 28	268,435,455 accounts; v2 (33 bits, 8B+ people) activates by FRAME LENGTH (80 B = v1) – no version bit, no hard fork

Defense of the choice. (a) *timestamp inside the ID* (the early draft – and exactly how today's XyncPay production `Transaction.pack` works): identical transfers within a second collide; sender time is untrusted ("check the balance at the second of sending" is unsafe – order is set by the network, not by the sender's clock); no replay protection. seq solves all three at once and gifts us the gap rule (§5) and safe offline signing (§13). (b) *ID = hash(body)*: +16 B on every tx and the loss of self-description. (c) *UTXO*: worse than the account model for pure payments in size and wallet complexity. (d) *seq epochs* (an intermediate design): complicated signatures and allowed ID reuse across epochs – cut in favor of the 22-bit counter.

Comparison: Bitcoin ≈ 250 B, Ethereum ≈ 110 B, Solana ≈ 230 B, Sui ≈ 300 B, **Xync = 80 B**. At 4096 TPS (the ladder's ceiling, §8) that is a mere 320 KB/s of payload traffic.

4. Step 1: signing

The wallet knows its own seq (it is the sender's own counter — no network state is needed to sign) and signs 16 bytes. **The iron wallet rule:** signed bytes are persisted BEFORE the first send; every retry transmits the identical copy; the local seq counter increments on EVERY signature, even unsent ones. Re-signing the same seq with a different body is a cryptographic autograph of a double-spend (§10): the protocol cannot and must not distinguish a "retry bug" from an attack.

5. Step 2: admission, the gap rule, reservation, the free lane

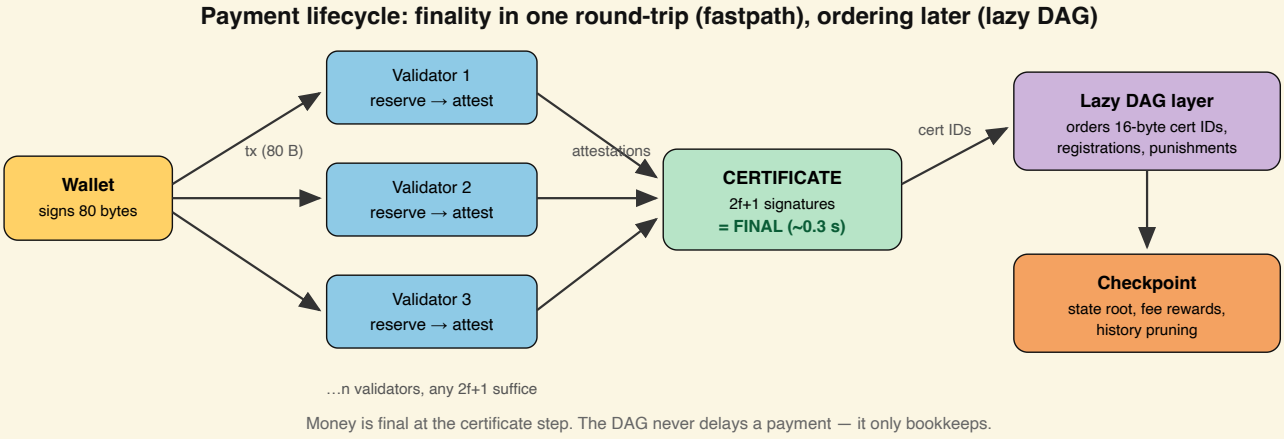
A validator, on an incoming tx (from a client or gossip):

1. codec ranges, account existence, ban flag;
2. **gap rule**: if tx.seq is ahead of expected, the validator does not know all of the sender's previous txs; the tx is PARKED until the missing ones arrive (never validate across a gap). Once the pipeline advances, the parking lot replays in order;
3. signature + **available** balance (settled − reserved) for amount and fee;
4. reserve the funds, and only then attest.

The reservation is a local guarantee: "this sender will not double-spend through me." The pending pipeline is capped (8 per account).

The free lane (fee=0): mechanics and pitfalls. Validating fee=0 is a protocol duty of ALL validators: finality needs a quorum, so "free" cannot be one node's setting. What IS per-node is **accepting them from users** (`ACCEPT_FREE` , altruistic mode): a regular node answers "use a charity node," an altruistic one accepts and gossips. Anti-spam: a hard protocol cap `free_tx_per_day` per account (farming accounts runs into the $1 registration: 20 free tx/day never pay back a dollar for a spammer), the free lane degrades first under load, and it contributes nothing to the fee pool — altruism doesn't dilute validator income. The living example: api.xync.net — the legacy XyncPay backend — remains the permanent free entry point for old-frontend clients (§12).

6. Step 3: attestation and finality (fastpath)



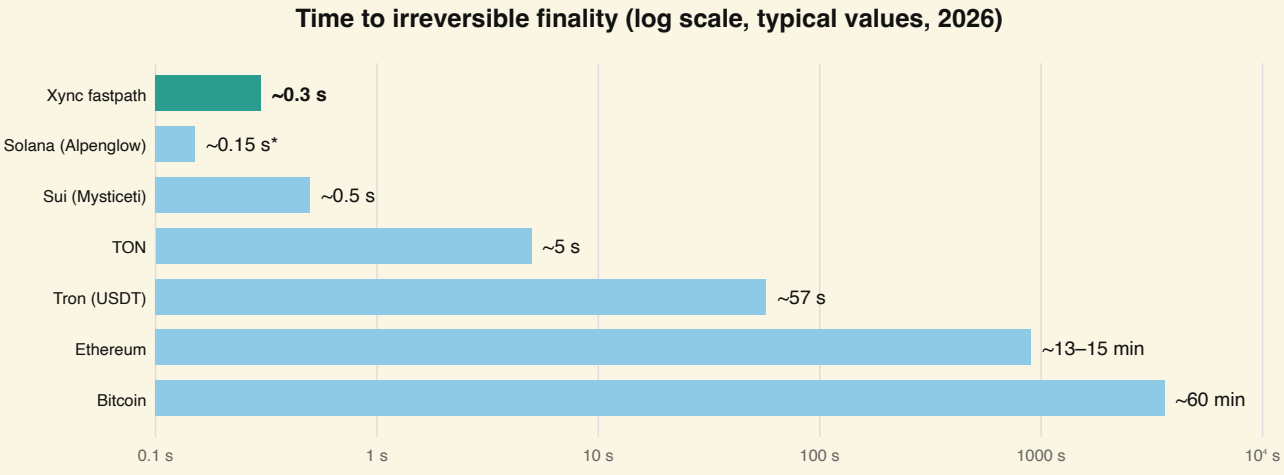
Money is final at the certificate step. The DAG never delays a payment — it only bookkeeps.

A successful reservation → an attestation `ed25519(DOMAIN_ATT ‖ tx_id)` → gossip. q = 2f+1 attestations = a **certificate** = finality. The certificate is self-contained (the 80-byte tx plus signatures; ~280 B at n=4), applied by any node immediately, and – crucial for §13 – **verifiable offline**.

Theorem (fastpath safety). Suppose certificates C_1, C_2 exist for different bodies with the same (from, seq). Their quorums intersect in ≥ f+1 validators, so at least one honest validator signed both bodies. But an honest validator, having reserved a seq, attests at most one body per (from, seq) – contradiction. ∎

Liveness: any 2f+1 live validators suffice for both attestations and the DAG – f crashes or traitors stop neither the money nor the bookkeeping.

Latency: $T_{final} = 2\delta + \varepsilon$, where ε is crypto (verify 30–60 µs per core; batch verification is ~2× faster). Target 150–400 ms globally – the proven optimum of the class. Fastpath throughput scales trivially: transfers of different senders are independent (embarrassingly parallel); the bottleneck is signatures – one core ≈ 15–30k verifies/s, an 8-core node exceeds 100k TPS before the network does.



* Alpenglow mainnet expected Q4 2026. Card networks authorize in ~2 s but settle in days and allow chargebacks.

State convergence. One sender's certificates apply strictly by seq (early arrivals buffer); different senders commute → all honest nodes converge byte-for-byte WITHOUT total order. Checked by a 46-invariant simulation: state roots of four independent Ledgers match to the bit.

7. Step 4: the network layer

The decisions come from the 07.2026 research and are defended against the alternatives. The prototype uses a WebSocket stand-in (for debuggability); the p2p service interface (`/broadcast` + type routing) survives the transport swap.

7.1. Transport: QUIC (neither raw UDP nor TCP)

Decision: **QUIC for all data traffic**; light discovery datagrams share the same UDP socket (demultiplexed by the first byte).

TLS 1.3 is built in (no separate Noise layer), 0-RTT resumption, stream multiplexing without head-of-line blocking, connection migration (a Wi-Fi→LTE handover doesn't drop the session — critical for the Telegram MiniApp). The decisive argument: **WebTransport = QUIC in the browser**; certhash lets a browser reach a validator without a CA certificate. The browser node gets the same transport for free — a direct road to the "full node inside the wallet" goal (WASM).

Alternative	Why not
Raw UDP + a homegrown reliability layer (Solana's road to QUIC)	We would have to implement retransmission, congestion control, encryption. Solana itself migrated to QUIC after the 2022 DoS incidents.
TCP (+TLS)	Head-of-line blocking across streams; no connection migration when the network changes (mobiles!); 2–3 RTT to establish against QUIC's 0–1.
The full libp2p stack	Drags multiformats, multiplexers, protocol negotiation — surplus compute and bytes. We take only the ideas (DCUtR-style hole punching).
WebRTC	A heavy stack (ICE/DTLS/SCTP) designed for media; for point-to-point data QUIC is simpler and faster.

The "raw UDP punches NAT better" myth is refuted by a 2025 measurement (4.4M attempts, 167 countries): hole-punch success is **QUIC ≈ TCP** — choosing QUIC sacrifices no reachability.

7.2. Node discovery: a Kademlia + PEX hybrid

Decision: **signed node records** (ENR style, ed25519) in a Kademlia DHT over UDP for global lookup; **Peer Exchange** over already-open QUIC links to densify the mesh cheaply; **DNS seeds** for bootstrap.

Alternative	Assessment
Kademlia DHT alone (Ethereum discv5)	Solid global O(log N) lookup, but every lookup is a series of RTTs; expensive for "find me a few more neighbours". Kept as the base.
Gossip/PEX alone (Bitcoin addr exchange)	Nearly free in traffic, but slow convergence and eclipse-vulnerable (neighbours dictate your list). Kept as a supplement, not the base.
UDP broadcast alone	Works on a LAN, does not exist on the internet. No.
A central registry/tracker	A point of failure and censorship. No — though DNS seeds as *bootstrap* are standard (Bitcoin, Ethereum EIP-1459).

Node records are signed with the node's key, so nobody can forge someone else's address. Anti-eclipse: peers are drawn from distinct /16 subnets plus random Kademlia buckets, not only through PEX.

7.3. NAT: three tiers of node instead of "everyone equal"

Measured reality: hole punching succeeds ~**70%** of the time (symmetric NATs don't yield). The protocol is therefore designed so that as few nodes as possible need inbound at all:

1. **Public validators/relays** — public IPs, accept everyone. The relay role is cheap: only punch coordination and the traffic of those who failed to punch flows through it.
2. **Home full nodes behind NAT** — outbound QUIC links to validators plus hole punching among themselves (DCUtR-style coordination via a relay; the observed-address in a discovery ping replaces STUN). The ~30% that fail get a circuit-relay fallback; the volume is small, since a node behind NAT need not accept inbound at all.
3. **Browser/mobile nodes** — outbound WebTransport only. Inbound is unnecessary by design: a subscription is an outbound stream, and gossip pushes arrive over already-open connections.

Alternative	Why not
UPnP/NAT-PMP	Works on some home routers, disabled by CGNAT carriers. As an opportunistic optimization yes; as the foundation no.
TURN/relay for everyone	Central cost grows linearly with the network; it kills decentralization. Fallback for the ~30% only.
"Every node needs a public IP"	Cuts off 95% of enthusiasts — direct damage to decentralization.

Traffic economy: non-validators do not relay consensus traffic and subscribe only to the streams they need (their own accounts + checkpoint headers) — node modes in §9.

7.4. Gossip discipline

A tx body crosses the network once; afterwards only 16-byte IDs travel. Priority lanes: fraud alarms > attestations/certificates > DAG blocks.

Layer	Prototype (this repository)	Production (Rust)
Transport	All-to-all WebSocket mesh	QUIC (quinn), WebTransport for browsers
Discovery	Static list from genesis	discv5-style DHT + PEX + DNS seeds
NAT	Not needed (docker network)	DCUtR-style punching + relay fallback
Gossip	Flood with `msg_id` dedup (optimal at n=4)	Push/pull hybrid, priority lanes, erasure block dissemination (Rotor idea) as the network grows

8. Step 5: the lazy DAG order

Payments are final without any total order (§6), yet a handful of event types still demand one: a registration hands out the next free account index, a fine debits a specific amount from a specific balance, `val_add` changes the quorum size for everyone at once, and a checkpoint must come out byte-identical on every node. Such events are rare and never urgent — which means the layer that orders them is allowed to be slow, so long as it is cheap and stays off the money path. The scheme is panel C above.

A block instead of a vote. A validator publishes a block `{author, round, refs, cert_ids, sys_txs, sig}`. One block referencing another says two things at once: "I received it" and "I vote for it" — there are no separate vote messages in the protocol at all, and this is family C's central economy. Finished fastpath certificates are not embedded in the block: they spread across the network long ago through their own gossip, so the block carries only their 16-byte identifiers. This is the same end-to-end principle at work: **every byte crosses the network exactly once.**

Where rounds come from. The network has no shared clock, so time is kept by quorum: round r+1 opens the moment q blocks of round r have been collected (a threshold clock). Nobody waits for a slow node. The round's leader is assigned round-robin (v2: several anchors at once plus uptime reputation, Shoal-style). A leader's block of round r counts as committed once q blocks of round r+1 reference it; if the leader stays silent, its round is simply skipped and the network never pauses.

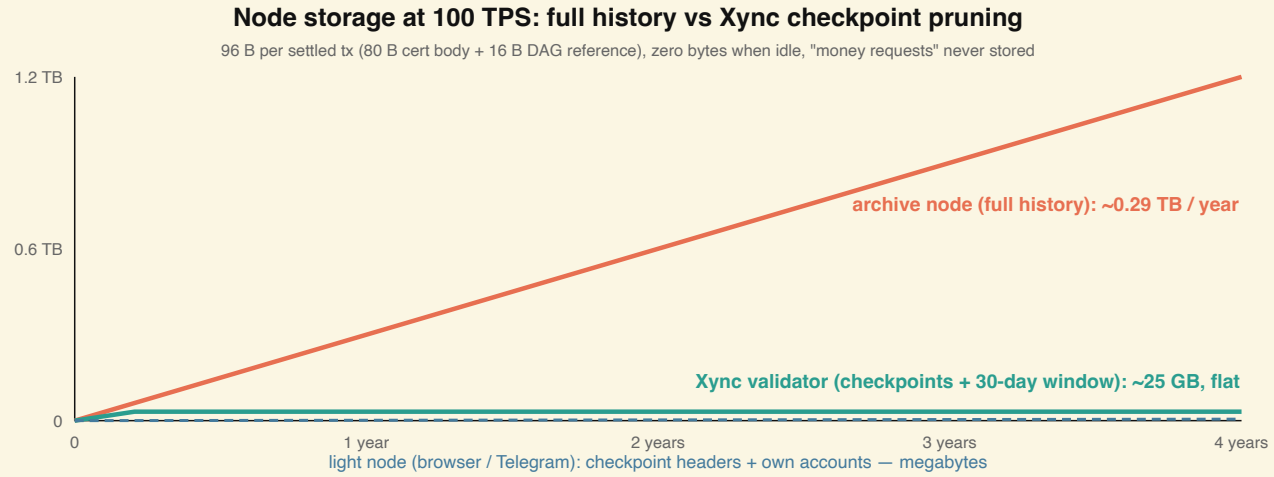
How a graph becomes a list. A committed leader block — the *anchor* — drags along its entire causal history: every block it references directly or through a chain of references. Each node walks that history by the same deterministic procedure and sorts the result by the pair (round, author). The resulting order is identical everywhere although no node agreed on it with anyone — the agreement follows from the shape of the graph, not from a vote. The payload of a round's non-leader blocks lands in the next anchor's commit; for money that delay is irrelevant, as it has been final since the fastpath.

Pace — the adaptive ladder. Every block costs traffic, and the DAG layer has nowhere to hurry. So a node waits longer before publishing the fewer certificates it has accumulated. The production parameters `[[4096, 1 s], [1024, 5 s], [64, 15 s], [1, 60 s]]` read as: 4096 certificates piled up — cut a block within a second; only one — wait a minute. The bottom rung is the point: **at zero activity there are no blocks whatsoever, and an idle network costs 0 bytes.** For comparison, Ethereum stamps an empty block every 12 seconds and Solana every 400 ms, and every node stores that emptiness forever.

9. Step 6: checkpoints, storage, light nodes

Every K commits — a checkpoint: the SHA-256 root of canonical state (plus quorum signatures in v2), the history-pruning boundary and the light-client entry point.

```
S(node) = 100 B · A  +  96 B · TPS · W  +  checkpoint headers
          state         history window
A — accounts, W — window in seconds; 96 = 80 (cert) + 16 (DAG ID)
```



Numbers: 1M accounts ≈ 100 MB of state; 100 TPS with a 30-day window ≈ 25 GB — a validator lives on a Raspberry Pi. A light node: checkpoint headers + its own accounts with proofs — megabytes, seconds to start, no "sync from genesis" ever. Full history is kept only by volunteer archive nodes (treasury grants). "Money requests" are ephemeral TTL objects and never enter the chain.

There are two node modes: **validator** (stake, quorum participation, rewards) and **light**. The customary third type — a full node without stake — is deliberately absent. Holding the full state while voluntarily forgoing validator rewards makes no sense when the hardware is identical either way. So "full" here is not a class of its own but the same validator binary launched without a key in the set: a mode for exchanges and legally cautious operators who want to see everything and vote on nothing.

Defense of the choice. Mina-style recursive SNARKs give constant history, but the prover is beyond browsers/phones; quorum checkpoints give the same practical benefits (instant start, pruning) with trivial verification. Classic container blocks would duplicate tx bodies.

10. Punishments: fraud proofs

Two differently-bodied signed txs with one (from, seq) form a self-contained proof (80+80 B + the reporter's signature). The catching node: (1) instantly freezes the sender locally and broadcasts a top-priority alarm — everyone freezes and releases the sender's reservations; (2) drops the proof into the DAG as a system transaction. Execution in commit order (determinism!): **permanent ban, fine `fraud_fine_xync` (or the entire XYNC balance), 50% to the reporter, 50% burned.** Reporter races are impossible: DAG order picks the same winner everywhere. The design intent: turn every node into a bounty hunter — catching a fraudster pays better than fraud.

11. Sybil resistance, stake and the dynamic validator set

11.1. Two different Sybil questions — two different answers

"Sybil" names two entirely unrelated threats, and conflating them is expensive. The first is cheap *accounts*; the second is cheap *power*.

Cheap accounts. If opening an account costs nothing, they get opened by the million: for spam, for farming the free lane, for hoarding short and pretty indexes. Our answer is a paid registration of about $1, paid into the network treasury. A dollar is invisible to a person opening one account and ruinous to a farm that needs a million. The side effects are pleasant: the treasury fills, and short indexes acquire a price and so are not squatted for free.

Alternative account-level anti-Sybil	Why not
PoW at registration	Grinding hashes on a phone means burning the user's battery to solve our problem.
KYC	Closes off permissionless entry — the very property the network exists to provide.
Social graphs, proof-of-personhood	Farms defeat them more cheaply than an honest person passes them, and they demand trust in the graph.

Cheap power. Were a quorum vote granted per account, that same million accounts would buy the whole network. So accounts do not vote at all: the right to sign attestations and DAG blocks comes **from stake alone**, quorum weight is proportional to its size, and signing two conflicting blocks is slashed. A million bought accounts grant not one gram of power over finality — they grant exactly one million empty wallets.

11.2. The dynamic validator set

The validator set is not frozen into genesis. The system transactions `val_add` / `val_remove`, signed by the governance key, ride the DAG layer, which means every node applies them at the same point of the same order — the set size n changes everywhere simultaneously and the quorum recomputes itself:

n	q = 2f+1	f	what it means
1	1	0	one node, one operator
2	2	0	both signatures required
3	3	0	all three required
4	**3**	**1**	**the first true BFT: the network survives the failure or malice of any single node**
7	5	2	survives two

Neither adding nor removing a node requires a network restart — an invariant that `selfcheck` verifies. A new node bootstraps its state from the latest checkpoint snapshot rather than replaying from genesis, and finds its peers over p2p by itself.

12. Migrating a live product: five phases, no downtime

This network is not being built on a blank slate: it already has a working predecessor — the XyncPay payment service, with live customers. Decentralization is therefore not the launch of a new chain but *surgery on a beating heart*: the product may not stop for a minute, and the user must notice nothing at all.

The starting point (07.2026): the `api.xync.net` backend validates everything single-handedly and the history lives in Postgres. An observation from the production code: its `Transaction.pack` already encodes 16 bytes (typ+ts | cur | receiver | amount | sender). The new network is the same format in spirit, with fixes (seq instead of ts, a fee level, 28-bit addresses, §3). The migration is therefore **a balance snapshot plus a history anchor**, not a rewrite of old transactions into the new format.

Three principles: the client notices nothing; every phase is reversible until declared final; no big bang.

Phase 1 — "the shadow" (1–2 weeks). A node of the new protocol comes up beside production: n=1, quorum 1 (the network works correctly with a single validator). After its own usual validation the backend duplicates every tx into the node asynchronously — failures never block production. Daily reconciliation of balances, Postgres ↔ `GET /account/{id}`. Production remains the sole source of truth. *Exit:* zero discrepancies for N consecutive days on live traffic.

Phase 2 — "switching the truth" (day X). Writes stop for 1–2 minutes; `scripts/migrate_from_postgres.py` builds genesis: balances by the production balance logic (verified-received − signed-sent), the currency registry from `cur`, keys from `user.pub`/`prv`, and `legacy_root` — a SHA-256 anchor of the entire centralized history. The past stays auditable without occupying a single byte of state: the full Postgres dump is published separately and checked against the anchor. The script's report flags keyless users and negative balances beforehand. Rollback: dual-write both ways for the first weeks, with Postgres as a complete mirror.

Phase 3 — "plus nodes" (months 1–3). The mechanics are described in §11.2 and need no network restart. What matters here is not the mechanics but how each added node changes the measure of trust placed in the Xync operator:

n	what changes in trust
1	as today, but already on the new protocol
2	a second operator gains a veto: nothing can be stolen without its signature
3	collusion of two out of three is impossible without leaving traces in the signatures
4	**the first real BFT: the network survives the failure or malice of any single node — Xync itself included**
7	Xync can be taken off the mandatory payment path entirely

Phase risk: fastpath latency grows with geography — the quorum is the fastest 2f+1 of n, so operators are picked for connectivity.

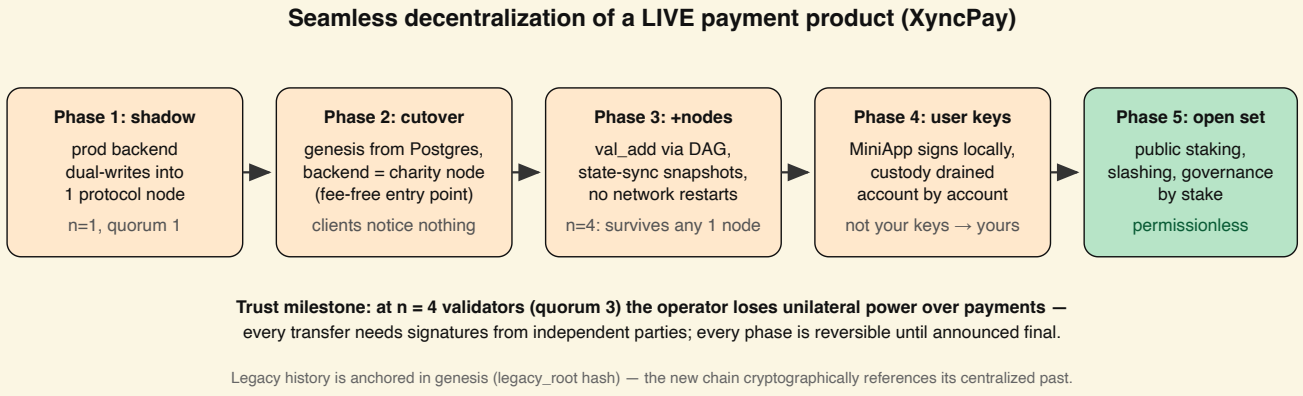
Phase 4 — "keys to the users" (parallel with phase 3). Keys are custodial today. The MiniApp generates a key locally, a `rekey` sys_tx (signed with the OLD key) replaces the account's pubkey, and

`User.prv` is wiped. "Not your keys" is solved one user at a time, with no common deadline; the gateway keeps serving non-custodial users as a relay.

Phase 5 — "the open set" (months 6+). A validator enters on a minimum XYNC stake, rewards come from fees (§15), block equivocation is slashed. The governance key: operator → team multisig → stake vote.

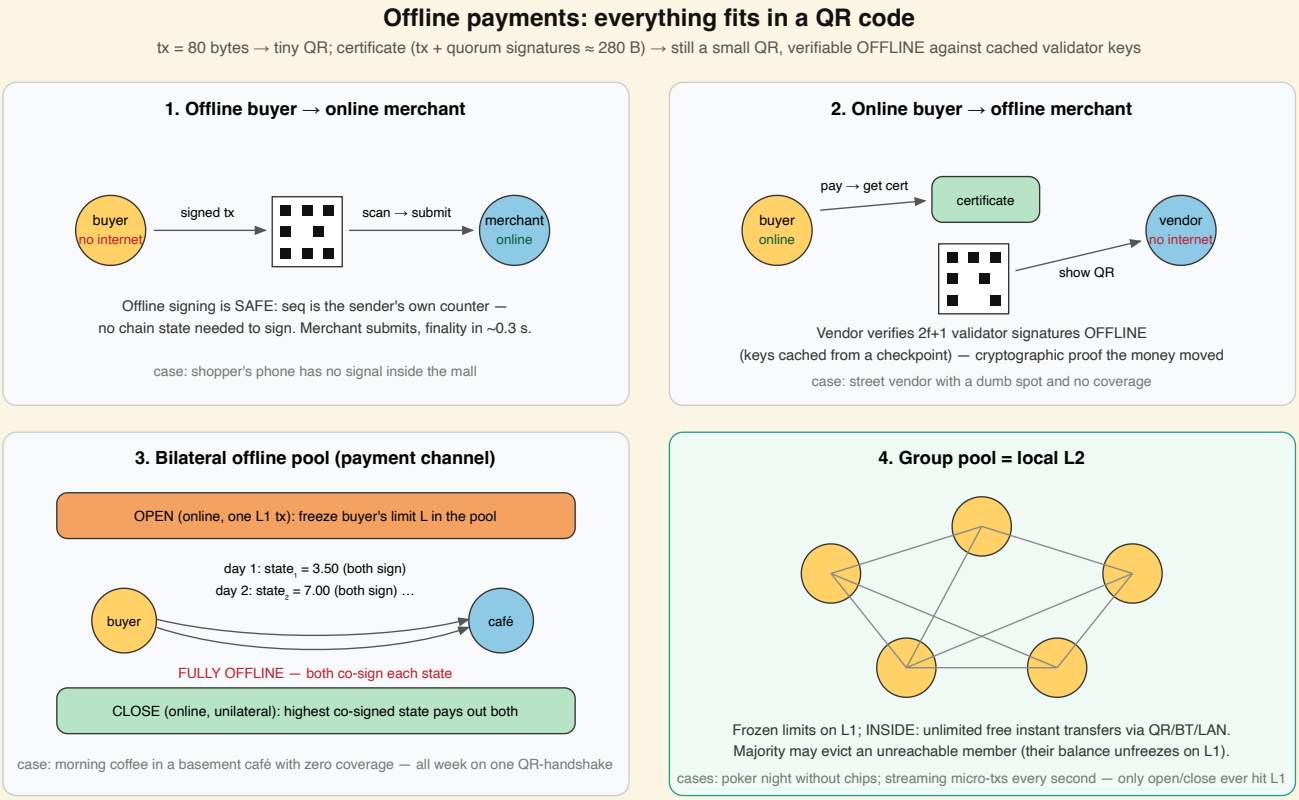
The legacy backend is never switched off: it becomes an **eternal charity node** (a validator in altruistic mode, `ACCEPT_FREE=1`) and remains the default entry point for old-frontend clients — free of charge, within the protocol daily limit (§5).

Migration question	Resolution
The old history	Not replayed; `legacy_root` in genesis is the crypto anchor; the Postgres dump is published for audit
Client ids	Preserved 1:1 (the codec allows #0 = the operator)
Money requests (`status=request` in production)	Never enter genesis — they are ephemeral; live requests are re-created by the gateway
signed-but-not-verified at snapshot time	The credit is already counted by the production balance formula; after launch the gateway resubmits them as ordinary txs
Currencies	ids from `cur` are preserved; code 0 is reserved for XYNC (unused in production — the script verifies this)



13. Offline payments and pools (v2)

Two protocol properties enable an offline class WITHOUT format changes: (1) **offline signing is safe** – seq is the sender's own counter, no network state is needed to sign; (2) **a certificate verifies offline** – 2f+1 validator signatures check out against the set cached from the latest checkpoint.



A signed tx is 80 bytes and a certificate at n=4 is ~280: both fit in an ordinary QR code.

Mode 1: offline buyer → online merchant. A buyer with no connectivity shows a QR holding the signed tx; the merchant scans it and submits it, watching finality land in ~0.3 s. Two subtleties: (a) the buyer's wallet must reserve seq locally – incrementing on every signature, even an unsent one, or the next offline signature creates a conflict, i.e. a self-denunciation; (b) the buyer learns nothing of the outcome until back online, and the wallet must honestly display "signed, awaiting delivery".

Mode 2: online buyer → offline merchant. The buyer pays from their phone, receives the certificate and shows its QR to the merchant. The merchant's offline verification: (a) the quorum of signatures against the cached validator set; (b) `to` is my account, currency and amount are right; (c) anti-replay: the pair `(from, seq)` has not been presented to me before (a local list of what was seen). A validator-set cache days or weeks old suffices: the set changes rarely and only through the DAG, and when it does, old keys stay valid for old certificates.

Mode 3: the bilateral pool – a payment channel. Opening (online, one L1 tx): `channel_open(payer, payee, cur, limit)` freezes the limit on the buyer's balance – the reserve machinery already exists in the Ledger. Offline payments: the buyer signs increasing states `state_k = (channel_id, k, spent_k)`, where `spent` grows monotonically and never exceeds the limit; on

scanning, the merchant CO-signs (a face-to-face meeting — two QRs exchanged), and both keep the state. Closing (online, unilateral): either party presents the largest co-signed state; payee receives `spent_k`, payer the remainder of the limit.

The defence against a "rollback" — presenting a stale state — is asymmetric, and worth spelling out. Payment to **the closer** is instant, in the amount of their share under the presented state; the **counterparty's** share is withheld for a short window (until the next checkpoint), during which they may present a state with a larger `k` and take more. Why the window protects the payee specifically: a stale state gives the payee less, and the remaining limit returns to the payer — so a rollback benefits the payer, and only the payer. It is pointless for the payee, whose share in older states is smaller. For a coffee shop all of this is invisible: it closes the channel itself in the evening and is paid instantly.

Mode 4: the group pool — a local L2. Members contribute limits `deposit_i` with a single L1 freeze tx. Inside lives a balance vector `B = (b_1..b_m)` under the invariant `Σb_i = Σdeposit_i`; inner transaction `k` is a signature from both counterparties of the transfer over `(pool_id, k, B_k)`, plus visibility for the rest on contact (local network, Bluetooth, QR). The whole group may be fully offline as long as its members stay in contact with each other. Evicting an absent member: a majority signs `evict(pool_id, member, B_last_seen)` — an online tx unfreezing their share at the last confirmed state; returning is possible only online, via `join`. Closing: the final vector under majority signatures → one L1 payout tx. Between opening and closing the network sees **nothing**: inner transfers are free and unlimited in frequency — "streaming every second". Use cases: poker without chips, a team on a hike, a market with no connectivity, per-second payment for content.

An escrow agent who never holds the money. The rule "a majority signs the close" yields escrow for free, without one new line of protocol. Let the pool hold three: a buyer, a seller, and a guarantor both of them chose. The buyer deposits the price, the seller deposits a bond, the guarantor deposits nothing. A majority here is two.

- *The deal goes well.* Buyer and seller sign the final vector between themselves — already a majority. The guarantor neither participates nor signs nor earns a cent. In ordinary life he might as well not exist.
- *A dispute.* The guarantor signs the vector together with whichever side he judges right — two of three again, and the pool closes.
- *The guarantor is crooked.* Stealing is physically impossible for him: his lone signature is not enough, and the second will only come from the person he did not rob. The money sits in frozen L1 deposits the whole time, never in an intermediary's account.

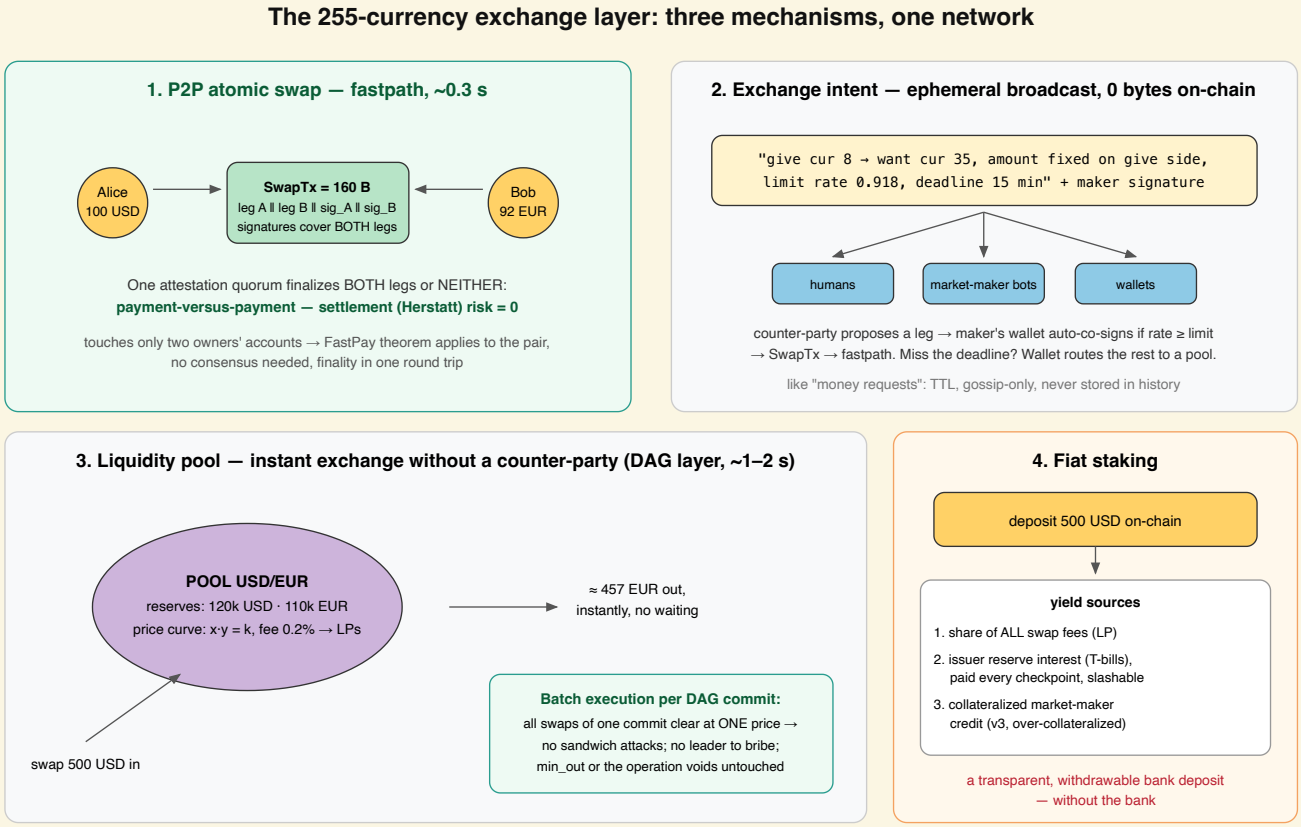
Hence "free": the network takes no percentage, and the guarantor charges nothing for deals he never touched. The internet is needed exactly twice — to open and to close; the deal itself happens offline, by exchanging QR codes. Honest limitation: a guarantor who colludes with one side robs the other. That is the ceiling of every "two of three" scheme, and the price of trust here is a single explicit one — choose the guarantor together.

Event	L1 transactions
Opening a pool (of either kind)	1
Thousands of inner payments	0
Evicting a member	1
Closing	1

Honest limitations. Modes 1–2 are asymmetric in awareness: whoever signed offline learns of finality later, and the UX must show it. Channels introduce a new class of state into the Ledger (open pools) and two new system tx kinds — that is v2; the prototype does not have them yet. The group pool demands local transport from wallets (BT/LAN/QR) and a gossip protocol inside the group — client complexity, not network complexity. A long full offline of both channel parties risks the counterparty having already closed it online; going online periodically to reconcile is basic hygiene, and the wallet reminds you.

14. The multi-currency layer: 255 currencies as a protocol primitive

The foundation of XyncNet's uniqueness: besides the native XYNC, **255 currencies/coins are wired into the protocol** — not contract tokens on top of a VM (like ERC-20), not bridge wrappers, but a `currency` field in every transaction and a registry in network state. The consequence: any operation in any currency inherits ALL protocol properties — 80 bytes, 2δ finality, offline certificates, the free lane.



14.1. The currency registry

`{code 1..255 -> name, scale, issuer_acct, listing_deposit}` — genesis + governance listing. Scale is per-currency (USD:3, BTC:8, JPY:0): the amount field is always interpreted in the currency's own minimal units. A fiat currency's issuer is a regulated entity with an XYNC listing deposit (slashed on misconduct) and mandatory oracle proof-of-reserves (§16). XYNC (code 0) is the only issuer-less currency.

Defense of the choice. The "tokens as contracts" alternative (ERC-20/SPL): every token is its own logic, its own audit, its own bugs, and a transfer costs VM execution. Here a currency is a NUMBER in a registry: zero attack surface, zero overhead, every currency behaves identically. The "bridges between chains" alternative: bridges are the hack champions of the industry (billions lost: Ronin $624M, Wormhole $326M, Nomad $190M…); our 255 currencies live inside ONE security domain of one validator set — the "bridge risk" class is absent by construction.

14.2. SwapTx: an atomic P2P exchange in the fastpath (160 bytes)

A new frame type (versioned by length: 80 B = transfer, 160 B = swap):

```
body_A(16) ‖ body_B(16) ‖ sig_A(64) ‖ sig_B(64)
body_A: {cur X, amount x, seq_A, fee, to=B, from=A}
body_B: {cur Y, amount y, seq_B, fee, to=A, from=B}
sig_i = ed25519(DOMAIN_SWP ‖ id_A ‖ id_B)   — a signature over the PAIR
pair_id = sha256_16(id_A ‖ id_B)            — object id for attestations
```

A validator reserves both legs atomically (both seqs, both balances — or nothing); one attestation over `pair_id`; one quorum finalizes both legs. **The pair safety theorem:** a conflict on either (from_A, seq_A) or (from_B, seq_B) cannot assemble a second quorum — the quorum intersection contains an honest validator who reserved the pair as a whole. This is PvP settlement (payment-versus-payment): "one leg executed, the other didn't" is impossible by construction — in traditional FX that is THE settlement risk (Herstatt risk) that has killed banks. The swap stays in the fastpath (0.3 s) because it touches only two owners' accounts — the FastPay theorem applies to the pair exactly as to a single transfer.

Atomic reservation (`reserve_swap`). Requirements are computed as a dictionary per SIDE and only over the OUTGOING legs — what is received from the counter-party does not count toward availability (otherwise a swap could "pay for itself"). Edge case: if a side gives XYNC, its principal amount and its XYNC fee are summed into one dictionary row — a naive implementation counting them separately would under-reserve. The reservation is a single entry keyed by `pair_id`; both sides' `pending_seqs` are extended together; releasing the reservation (quorum timeout, fraud freeze) rolls back both legs.

Finalization — the synchronization point of two seq chains (`settle_swap`). A swap is applied only when BOTH legs have reached their turn (`seq == side.seq + 1` on each). If either is "in the future," the certificate is buffered at BOTH sides and replayed as soon as the second leg arrives. **No deadlock:** each seq chain is monotone and independent, and the swap is their only shared point. Idempotency — a guard on `pair_id` BEFORE any mutations (gossip delivers the certificate 2–4 times). Both `pair_id` and both leg ids are placed into `settled` — so a leg cannot be replayed later as a lone 80-byte frame.

Domain separation = atomicity at the signature level. A leg's signature is over `DOMAIN_SWP ‖ id_A ‖ id_B`, not over the leg's body. Hence one leg's bytes cannot be presented as a valid standalone transfer (which is verified under `DOMAIN_TX`): neither participant can "extract" the leg favorable to them from the swap and execute it separately.

Generalized fraud proofs. A double-spend proof now accepts 80- OR 160-byte frames. From each, a set of spends `(from, seq, id)` is extracted (one for a transfer, two for a swap); the proof is valid if there is an intersection on `(from, seq)` with DIFFERENT bodies AND all signatures of both frames verify under their respective domains. Consequence: spending one seq both with a transfer and with a swap leg is an equally provable offense, leading to the same ban + fine (§10), with no separate machinery.

14.3. Exchange intents: the broadcast "I want to swap" request

The ephemeral layer (like "money requests" — zero bytes in history):

```
ExchangeIntent { maker, give_cur, want_cur, fixed: give|want, amount,
                 limit_rate, deadline_ms, sig(DOMAIN_XIN ‖ intent_id) }
```

The amount is specified either in the currency offered or in the currency desired (the `fixed` flag); the deadline sets both urgency and TTL. The intent spreads via gossip; a counter-party (a human or a market-maker bot) sends back a matching half-swap (a proposal); the maker's wallet auto-co-signs the SwapTx if the rate is no worse than limit_rate → fastpath. Deadline strategy: wait for a P2P match until `deadline − margin`, then execute the remainder through a pool (§14.4).

The rate is strictly integer, no float. The rate is represented as `rate = amount_want · 10⁹ // amount_give` (scale 1e9). Matched amounts are rounded AGAINST the taker: with `fixed = give` the desired `want` rounds up (`ceil`), with `fixed = want` the offered `give` rounds down (`floor`). The reason for strict integers is not only determinism: the rate is compared on DIFFERENT nodes and in DIFFERENT wallets, and `float` would diverge in the last bit — and with it come disputes over whether the trade clears limit_rate.

Where auto-co-signing lives. The SwapTx for an intent is co-signed by the maker's WALLET (`wallet watch`), not by a node: nodes deliberately never hold users' private keys (a security-surface invariant). Anti-spam: intents are signed, a TTL is mandatory, and there is a cap on active intents per maker (8 by default) — otherwise a free ephemeral announcement would become a flood vector.

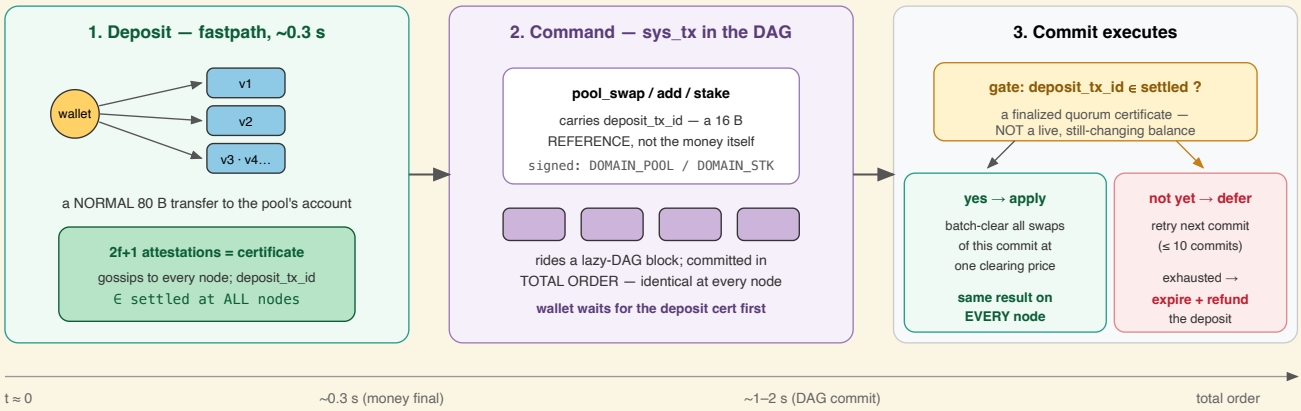
14.4. Liquidity pools: instant exchange with no counter-party

A pool is a typed state object (NOT a smart contract): `{cur_a, cur_b, ra, rb, total_shares, fee_bps, acct}`. The operations `pool_create/add/remove/swap` are **DAG-layer** system transactions: pool reserves are shared mutable state and need total order (the deliberate fastpath boundary from §1.3; pool-swap latency ~1–2 s, transfers unaffected).

Why the pool is in the DAG, not the fastpath. The FastPay theorem rests on an object having ONE owner: the quorum intersection catches its double spend. A pool has no owner — anyone swaps against the shared reserves, and the "live" reserve at the moment of processing DIFFERS across nodes (certificates arrive in different orders). The only way to obtain byte-for-byte identical state is to execute pool operations in the total order of the DAG.

Deposit-then-command (the determinism key). A pool's reserves and payouts change ONLY in `ledger.commit()`; the "live" balance is never read in the commit. Money enters a pool only as an ALREADY-finalized fastpath deposit (a plain 80-byte transfer to the pool's account), and the command carries a 16-byte REFERENCE `deposit_tx_id`. In the commit the command executes only if `deposit_tx_id ∈ settled`; otherwise it is deferred and retried (up to `POOL_CMD_MAX_RETRIES = 10` commits), then expires with a refund. Deposit and command dedup — the sets `used_deposits` and `pool_cmds`, both part of the checkpoint root. The pool account is created with an empty pubkey: `reserve()` answers `no_account` for it — spending from a pool via the fastpath is impossible in principle (there is no one to sign), while incoming transfers work.

deposit-then-command: deterministic pool state without reading live balances



Why the reference — not a live balance — is the only deterministic gate

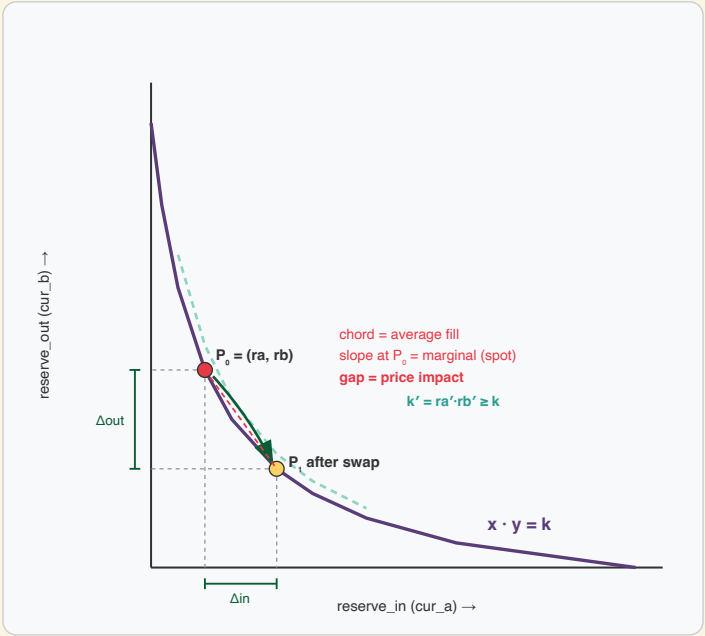
Fastpath certificates reach different nodes in DIFFERENT order. If commit read a "current" pool balance, two nodes could clear the same batch against different inputs and their state roots would diverge.

Gating on "deposit_tx_id ∈ settled" (a finalized quorum cert) makes the input a total-order fact; the retry queue absorbs jitter, so every honest node applies the same commit to the same reserves.

All pool / share / issuer state and the dedup sets enter the checkpoint root — otherwise node roots would drift.

Pricing – constant product, in integers (`xync/state/pool.py`). The fee is withheld on input: `eff = in · (10⁴ − fee_bps) // 10⁴`; the curve output is `curve_out = r_out · eff // (r_in + eff)` (the floor of `r_out − k/(r_in+eff)`, k = ra·rb). The FULL input (gross) is placed into the reserves, while the payout is computed from `eff` – the difference (the fee) settles into the reserves and accrues to the LPs. Shares: the first liquidity mints `isqrt(da·db)` (the geometric mean, as in Uniswap-v2: the first LP sets the starting rate, and the root makes shares invariant to the deposit's scale); an addition mints `min(S·da//ra, S·db//rb)` by the limiting side, with the excess refunded; a withdrawal takes both reserves down proportionally.

Constant product x·y = k: price impact, the fee wedge, and why k never falls



Integer curve, rounding toward the pool

The formula (Uniswap-v2 in integers)

```
eff = amount·(10000 − fee_bps) // 10000
out = reserve_out·eff // (reserve_in + eff)
```

both floor — never round in the trader's favour

The fee wedge = LP income

GROSS input enters the reserves; payouts are computed from EFF (< gross). The retained fee stays in the pool — that is the liquidity income.

Invariant: k never decreases

- spot netting of opposing flows leaves reserves unchanged (neutral);
- floor in curve_out ⇒ (ra+eff)·rb′ ≥ ra·rb;
- floor in pro-rata payout leaves dust in the pool;
- gross in, eff out ⇒ the fee is never paid out.

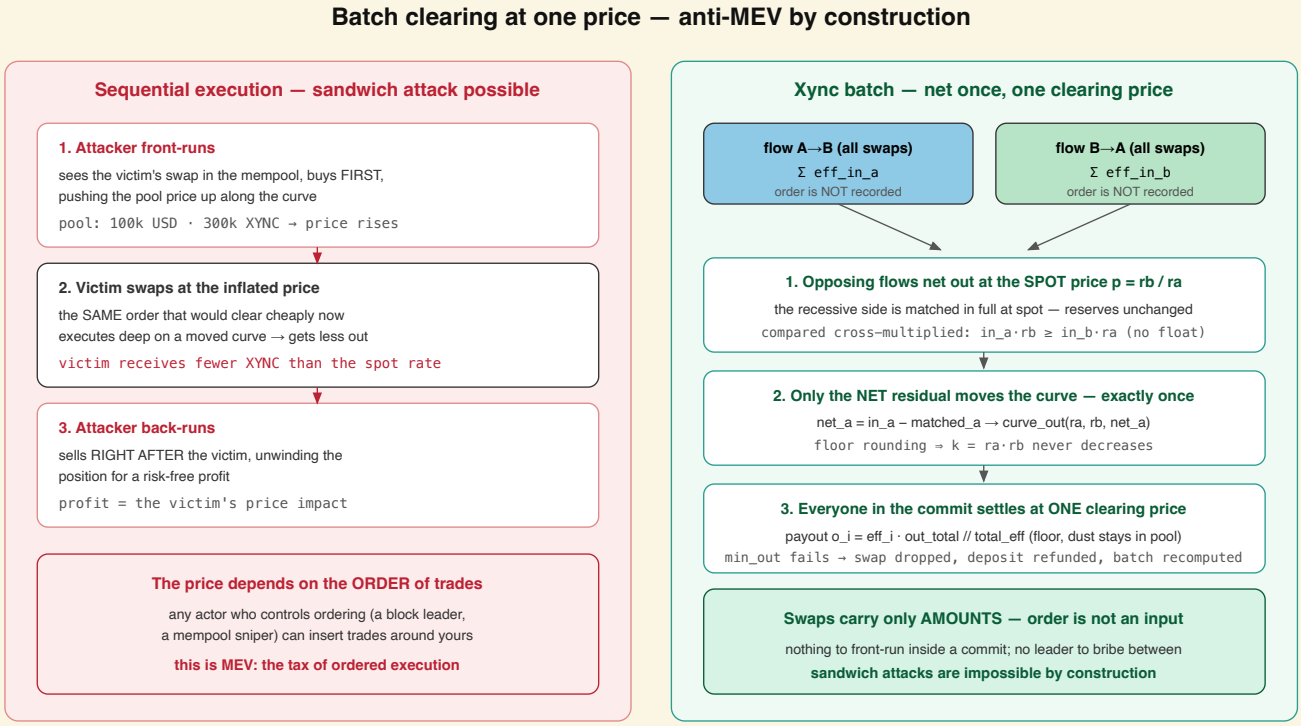
⇒ k′ = ra′·rb′ ≥ k, always

Why constant product (not an order book)

a pure function of two reserves — no VM, no matching engine, no per-trade code execution; always quotes a price, byte-identical on every node.

Anti-MEV: batch execution at a uniform clearing price. All `pool_swap`s of one DAG commit against a pool execute as ONE exchange:

1. opposing flows are netted at the pool's INSTANT rate p = rb/ra. The dominant direction is determined by cross-multiplication `in_a·rb ≥ in_b·ra` (no division, no float); the recessive side is matched in full at spot – an exchange at the instant price does not move the reserves;
2. the NET remainder of the dominant side passes through the curve `curve_out` EXACTLY ONCE – the only price move in the whole batch;
3. each swap gets a share of the total payout pro-rata to its input: `o_i = eff_i · out_total // total_eff` (floor);
4. `min_out` is checked against the FINAL individual payout; a swap that fails is excluded, its deposit refunded, and the batch is RECOMPUTED (to a fixpoint, ≤ n iterations). `expire_round < current` → refunded without participating.



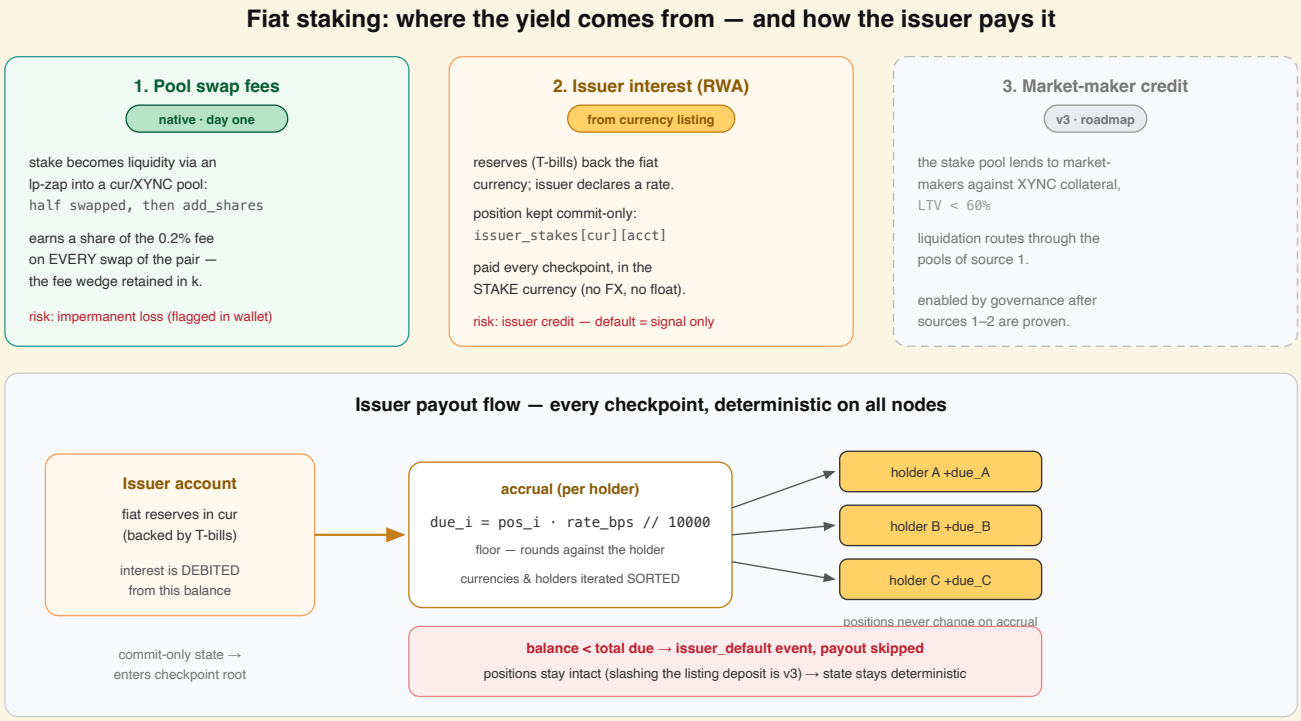
Why this is constructive anti-MEV. Only the per-direction SUMS enter the clearing – the order of swaps within a batch does not affect the result. Therefore a sandwich is mathematically impossible inside a commit: you cannot "stand in front of" a victim when everyone gets one price. Between commits there is no leader paid to insert a transaction: ordering is set by the deterministic DAG linearization (§8), not by an auction for a slot in a block.

The invariant k = ra·rb never decreases (checked by an `assert` in debug and by section 9 of selfcheck). Step by step: spot netting does not change the reserves → k is preserved; `curve_out` rounds the output DOWN → the net trade yields k′ ≥ k; the floor in pro-rata leaves "dust" in the pool → k grows a touch more. All rounding favors the pool; hence LPs cannot lose to rounding arithmetic, and the fee monotonically grows the reserves.

14.5. Fiat staking: a deposit without the bank

`stake(cur, amount, strategy)` — a DAG operation via the same deposit-then-command pattern (signed under `DOMAIN_STAKE`). Yield sources (honestly, by increasing maturity):

1. **Swap fees (native, day one)** — `strategy = "lp:<pid>"`. The stake works as pool liquidity: the depositor earns a share of the 0.2% fee on ALL exchanges of the pair. The prototype implements a one-sided "zap" — half the deposit is swapped along the curve into the paired XYNC, then liquidity is added on both sides (the remainder refunded); the pool's k invariant is not broken by this. Risk — impermanent loss, labeled in the wallet.
2. **Issuer interest (RWA)** — `strategy = "issuer"`. The funds move to the currency's issuer account (`issuer_acct` from the registry); the position is recorded in `issuer_stakes[cur][acct]`. At EVERY checkpoint holders are accrued `due = pos · rate_bps // 10⁴` (floor against the holder), debited from the issuer account. The rate is declared on-chain; payouts arrive automatically, and the reserves sit under oracle proof-of-reserves (§16). The "like a deposit, but transparent" profile.
3. **Market-maker credit (v3):** the staking pool lends to MMs against XYNC over-collateral (LTV<60%), with liquidation through the §14.4 pools.



Determinism of accrual. The interest payout runs in `_checkpoint` (the same place where the fee pool is distributed — the same class of operations), iterating over currencies and holders in SORTED order, with integer sums. If the issuer's funds fall short, an `issuer_default` signal is emitted (in the prototype — only an event; slashing the listing deposit is v3), while holders' positions stay untouched: the network does not punish a depositor for the issuer's default, it only records it on-chain.

Withdrawal (`unstake`) is the inverse operation: issuer → a payout from the issuer account, lp → `pool_remove`. The network wins three times: deeper exchange liquidity, more fees, and a real cash-flow product for non-technical users.

14.6. New operation types at a glance (implemented, v2)

Operation	Frame/layer	Latency	Signature domain
SwapTx	160 B, fastpath	~0.3 s	DOMAIN_SWP
ExchangeIntent	ephemeral gossip	0 bytes on-chain	DOMAIN_XIN
pool_create/add/remove/swap	sys_tx, DAG	~1–2 s	DOMAIN_POOL
stake/unstake	sys_tx, DAG	~1–2 s	DOMAIN_STAKE

14.7. Exchange-layer invariants and honest limitations

The exchange layer is implemented (4 PRs) and protected by the same principles as the payment core. The key invariants and how each is checked:

Invariant	How ensured	Check
Layer boundary: shared state changes only in commit, in DAG order	deposit-then-command; the "live" balance is never read in the commit	selfcheck §9
Commit determinism (byte-for-byte across all nodes)	integer math, sorted iterations, zero float	two runs of §9, matching roots
k = ra·rb never decreases	floor in curve_out and pro-rata, gross into reserves	`assert` + §9
Idempotency of an incoming cert	guard on pair_id; `used_deposits`, `pool_cmds`	§7, §9
Each type — its own signature domain	SWP / XIN / POOL / STAKE	§7, §10
All commit-only state — in the checkpoint root	pools, shares, issuer positions, dedup sets in `_state_dump` and the snapshot	§9, state-sync

In total, 46 `selfcheck` invariants (sections 7–10 are the exchange layer) plus e2e on a real network of 4 validators.

Honest limitations of the prototype. All of them are listed here; the journal in which each was decided and dated is `docs/decisions.md`:

- **The deposit gate is on local `settled`, not on "the certificate is committed to the DAG."** In practice it converges (the wallet waits ~0.3 s for the deposit to finalize before sending the command; the commit lands ~1–2 s later, by which time the certificate has spread to everyone; retries absorb the jitter). A strictly deterministic gate on a committed cert_id is a v2 plan.
- **A "seq burn" risk in the wallet's two-step P2P swap:** an expired half-swap can occupy a seq; the prototype wallet keeps no local seq-lock. The clean fix (a seq-lock with a TTL) is v2 / the Rust port.
- **The lp-zap is simplified** to `cur/XYNC` pairs, and zap swaps are not batched with the general pool; the issuer stake — the primary scenario of §14.5 — is unaffected by this.
- **Issuer interest** is paid in the stake's currency (no FX rate, to avoid introducing float) and pro-rata to CHECKPOINTS, not to real time.
- **`issuer_default` is only a signal**; slashing the issuer's listing deposit is v3.

15. Tokenomics

Emission: disinflation with a floor.

```
E(y) = max(E₀ · λʸ, E_floor),   E₀ = 4%/yr, λ = 0.85, E_floor = 0.75%/yr
```

In its first year the network prints 4% of supply; each following year prints 15% less than the one before, down to a floor of 0.75% a year below which emission never falls. The floor is not greed but a security budget: validators must still be paid twenty years from now, and a young network's fees will not feed them. The rejected alternatives: zero emission from day one (nothing to pay with until volume exists); a fixed percentage forever (chronic downward pressure on price); Ethereum-style adaptive emission (a superfluous feedback loop between token price and security — a reflexivity a payment network can only suffer from).

The fee and its fate. Fees are paid in XYNC only: a base of ≈ 0.001 XYNC multiplied by the fee-level multiplier (§3). Every fee collected splits three ways.

Share	To whom	Purpose
50%	validators	payment for work: signatures, storage, traffic
25%	burned	makes the network deflationary once volume suffices
25%	treasury	archive-node grants, referral vouchers, development

Registration deposits go to the treasury in full. A caught fraudster's fine splits in half: one half to whoever presented the proof, one half burned.

How the validators' half is divided. Neither evenly nor by stake alone. A node's weight in the distribution is `stake × uptime × attestation speed`, plus a bonus to whoever's block became the commit anchor. Uptime sits in that formula on purpose: a validator that sleeps earns nothing however large its stake — this is the economic expression of the requirement of uninterrupted service.

When the network turns deflationary. Burned exceeds printed as soon as the annual fee volume crosses the threshold V*:

```
0.25 · fee · V* > E(y) · Supply
```

We promise no "deflation from day one": below the threshold the network is inflationary, and that is the honest price of validators surviving long enough to see the volume.

What the reward does not measure. It is proportional to the **number** of transactions, not to their amounts: moving a million costs exactly what moving a dollar costs. A payment network has no business fining large transfers — a user who needs priority buys it with a fee level, not with the size of the payment. The free lane (fee=0) mints no rewards and dilutes no supply: the altruist pays in traffic, not in someone else's share.

Honest caveat. Fees in XYNC only means friction at onboarding: to send your own dollars you need a foreign token. This is precisely what Ethereum's users complain about. Three mitigations: a starter XYNC grant issued with the paid registration; an automatic "from the change" swap inside the wallet (§14.4 makes it a single operation); and, finally, the fee share itself is a governance parameter, not a protocol constant.

16. Oracles: fiat events as cryptographic facts

A network in which dollars and euros live must be able to learn that a bank transfer really happened. Usually that job goes to trusted intermediaries. We need none, because the fact we want is already signed cryptographically — just not by us.

The key observation. Every email from a bank carries a DKIM signature: the bank's mail server signs the headers and body with its key, and the public key sits in the bank's DNS. So an email saying "we credited you €1000" is not a screenshot, forgeable in a minute, but a verifiable cryptographic fact that only the bank itself could have produced.

The maturity ladder. We climb it as the network grows, and each rung is strictly stronger than the one below:

1. **An M-of-N committee.** Validators receive the email, each verifies the DKIM signature itself and votes with an attestation through the DAG layer. Simple, but M validators see the whole email — there is no privacy.
2. **zkEmail.** The user builds a ZK proof of the statement "an email from domain `bank.example` with fields Y exists, signed by a valid DKIM key," revealing neither the rest of the content nor the email itself. Privacy arises by construction, not by promise.
3. **zkTLS.** The same, but straight from the bank's API, with no mail intermediary at all.

The class is not hypothetical: zkP2P v2 (2025) runs in production against Venmo, Revolut and Wise.

What it buys us. A non-custodial fiat on- and off-ramp: escrow unlocks on a proof of a bank transfer, not on an exchanger's word. Auto-settlement of a "money request" against a bank receipt. Verifiable reserves for the issuers of registry currencies (§14.1) — a proof on demand rather than an audit report once a quarter.

Risks we see, and what we do about them. The bank rotates its DKIM key — the registry keeps the key history, and old proofs stay valid. A phishing domain resembling the bank's — only domains whitelisted in the currency registry are accepted. The bank claws a completed payment back — for large sums the escrow window outlasts the bank's clawback period.

All oracle events are DAG-layer system transactions. Payment latency is untouched.

17. Summary comparison and limitations

	Xync	Solana	Sui	TON	Tron	Ethereum
Finality	**~0.3 s**	~0.15 s*	~0.5 s	~5 s	~57 s	~13 min
Tx size	**80 B**	~230 B	~300 B	~300 B	~200 B	~110 B
Idle cost	**0 B**	empty blocks	empty checkpoints	empty	empty	empty
Browser node	**by design**	no	no	no	no	light protocols
Free tier	**yes**	no	no	no	no	no
Offline payment acceptance	**certificate in a QR**	no	no	no	no	no
Smart contracts	**no (deliberately)**	yes	yes	yes	yes	yes

* Alpenglow, mainnet ~Q4 2026.

Limitations we state ourselves:

1. **A buggy wallet can ban itself.** The protocol cannot tell a retry bug from an attempted double spend, and it must not: the iron signing rule (§4) is mandatory for every implementation. Mainnet softens it with graduated fines and a governance appeal, not by dropping the rule.
2. **The DAG layer dislikes bad networks.** Packet loss inflates its latency several-fold. Money is untouched — it finalizes on the fastpath — and the plan B against chronic degradation is on file: an optimistic path in the spirit of Raptr.
3. **The domain is narrow.** Anything that is not "send, request, exchange" is not for us: no NFTs, no derivatives, no arbitrary logic. This is the deliberate choice from which every advantage above grows, and simultaneously the ceiling of the network's applicability.
4. **Payment regulation.** The least deterministic risk of all. The strategy: non-custodial by default, licensed gateway partners at the fiat boundary, geographically diverse validators.

18. Roadmap

Stage	Content	Criterion
0 (done)	prototype: 4 validators × 5 microservices, fastpath+DAG, exchange layer, fraud proofs, dynamic set, migration scripts	selfcheck 46/46, 113 tests (e2e on a real network included), `docker compose up`
1	production shadow: dual-write from xync-hybrid, reconciliation	0 mismatches for N days
2	cutover: genesis from Postgres, backend = charity node	production on-protocol, n=1
3	partner nodes	n=4: f=1, the operator loses unilateral power
4	Rust core (codec/ledger/dag are an isolated spec already), QUIC	Rust⇄Python cross-test on one testnet
5	WASM light node: browser extension, TG MiniApp	<3 s cold start on a phone
6	user keys (`rekey`), open staking, channels §13	permissionless validator entry

The Python prototype is an executable specification: `xync/common/codec.py`, `xync/state/ledger.py`, `xync/consensus/dag.py` port to Rust mechanically; test vectors included. Join us: the XyncPay team, t.me/XyncPayBot.