PROTOCOL RESEARCH WHITEPAPER

Sour: Flow Perpetuals and Price-Space Batch Clearing for Synthetic Perpetual Markets on Solana

A research characterization of a curve-bracket perpetual DEX whose positions are demand curves, whose clears are event-triggered batches, and whose funding, settlement, and cascade logic are integrated into a deterministic on-chain clearing instruction.

Sour Research, prepared from the Sour repository and protocol documents

Repository state inspected: May 1, 2026

ABSTRACT

Sour is a Solana perpetuals protocol that replaces the usual scalar market-order or limit-order representation with two-breakpoint price-space demand curves. Each open position is a curve over price; a permissionless clearing instruction aggregates active curves, includes the oracle index price as a virtual breakpoint, computes an effective LP absorption slope, solves a monotone piecewise-linear equilibrium, writes per-position PnL, accrues fees, and executes margin cascade when required. This paper formalizes the implemented mechanism, derives its fixed-point arithmetic, compares it with major perpetual DEX designs, and states the engineering tradeoffs visible in the inspected repository. The central design claim is not that Sour eliminates all market risk or all extractable value; rather, it relocates execution from serial priority queues into uniform-price event batches, embeds funding-like mark drift in the clearing equation, and makes liquidation a consequence of clearing rather than a separate keeper race.

Keywords: perpetual futures, batch auction, flow trading, Solana, DeFi, on-chain clearing, synthetic derivatives, price-space curves, margin cascade.

This document is a technical whitepaper and repository characterization. It is not a security audit, legal advice, investment advice, or a statement that the protocol is mainnet-ready. External protocol descriptions cite official or primary sources available on May 1, 2026.

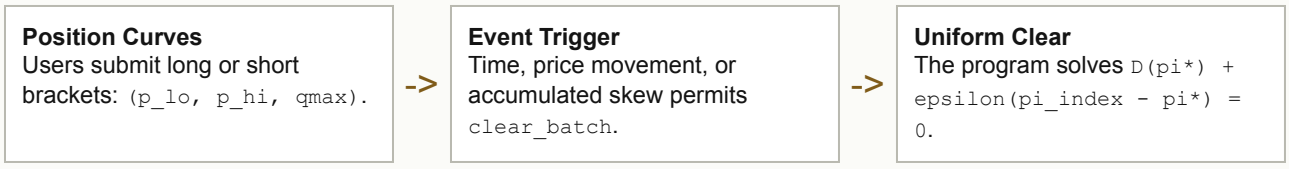
1. Introduction

Perpetual DEXs have converged around three broad mechanisms: central-limit-order-book protocols with priority queues, virtual-AMM or hybrid AMM systems, and trader-to-pool oracle-priced venues. These models have produced deep markets, but they retain familiar microstructure problems: adverse selection at oracle boundaries, queue-position races, separate funding and liquidation processes, and a product split between sophisticated order books and simplified retail interfaces.

The Sour repository describes and implements a different primitive: a flow-perpetual market on Solana. Its README states the mechanism as a synthesis of price-space perpetuals and the Budish-Cramton-Kyle-Lee-Malec flow-trading program: events fire on price-line crossings or other triggers, each position is a two-breakpoint demand curve, and batches clear by solving the aggregate demand equation. The on-chain program is organized around five account classes: `Protocol`, `SourVault`, `Market`, `TraderAccount`, and `Position`.

The distinctive idea is that a trader does not merely submit "buy X contracts" or "long with Y leverage". The trader defines a bracket in price space, and the bracket maps clearing price to exposure. The market then solves for a common clearing price that balances trader curves against an LP absorption term anchored to the current index price. This gives Sour the character of a batch-cleared synthetic market rather than a per-order matching venue.

FIGURE 1. SOUR CLEARING LOOP



2. Repository and System Context

Sour is a monorepo. The canonical protocol implementation lives in Rust under `programs/sour` and `programs/sour-math`. TypeScript packages provide the SDK, account decoders, configuration, and shared UI components. Frontend surfaces include a trading app, a marketing site, and an Expo-based iOS app. Services include a keeper, market maker, and bot-core layer. The repository's product documents define three public surfaces: `sour.finance`, `app.sour.finance`, and a native-feeling iOS app, all sharing the same chain effects through the same program instructions.

The code should be treated as canonical where documentation diverges. Several older documents use Q64-style names for price fields; the math modules explicitly state that prices are raw integer micro-USDC per base unit, widened only for safe arithmetic. Likewise, v0.4 introduced a fixed-point storage convention for `qmax` so that small BTC-size bets do not round to zero. The paper follows the implementation rather than older prose when specifying arithmetic.

LAYER	REPOSITORY EVIDENCE	WHITEPAPER INTERPRETATION
Protocol	`programs/sour/src/state.rs`, `instructions/`	Zero-copy account model, deterministic instruction set, single global LP vault, per-market state.
Math	`programs/sour-math/src/{curve,solver,epsilon,clear,fees,margin}.rs`	Curve evaluation, equilibrium solving, fixed-point PnL, fee, and cascade primitives.
Trader UX	`sites/app`, `packages/sour-sdk`, v0.4 product specs	Trader Mode and Degen Mode both compile to the same `upsert_position` primitive.
Operations	`services/sour-keeper`, `services/sour-mm`, v0.4i RPC hotfix docs	Keepers crank clear batches; market making and account state are moving toward WebSocket-backed live state.

3. Formal Model

3.1 Units and fixed-point convention

Let prices be stored as micro-USDC per base unit. Let `F = 2^32`. In the v0.4 storage convention, a position stores `qmax_storage = qmax_underlying * F`. Curve fill ratios are also scaled by `F`; therefore the raw signed demand returned by the curve module carries two factors of `F`. Notional and PnL routines remove both factors by division by `F^2 = 2^64`.

```
F = 2^32
q_i = qmax_storage_i = qmax_underlying_i * F
effective_size_i(pi) = s_i * q_i * x_i(pi)
notional_i(pi) = |effective_size_i(pi)| * pi / F^2
```

3.2 Two-breakpoint position curves

Each position has a lower price `p_lo`, upper price `p_hi`, maximum size `qmax`, and sign. For a long, exposure is maximal below the lower breakpoint and decays linearly to zero at the upper breakpoint. For a short, the sign is reversed and the fill ratio increases with price.

```
Long fill:
x_L(pi) = clamp_01((p_hi - pi) / (p_hi - p_lo))

Short fill:
x_S(pi) = clamp_01((pi - p_lo) / (p_hi - p_lo))

Signed demand:
D_i(pi) = +q_i * x_L(pi)  for longs
D_i(pi) = -q_i * x_S(pi)  for shorts
D(pi)   = sum_i D_i(pi)
```

This representation is more expressive than a scalar order because it lets a trader encode how desired exposure changes as price traverses a band. It also creates a natural UI primitive: dragging a demand band on

a chart and moving a leverage or bet-size slider both update the same underlying instruction arguments.

3.3 Event-triggered batch eligibility

A clear is not continuously scheduled by a funding clock. The `clear_batch` instruction first checks whether one of three triggers is satisfied: elapsed slots since the prior clear, price movement beyond a basis-point threshold, or accumulated market skew beyond a cap. The instruction also has a stale-market pause threshold, so a long period without clearing can pause the market.

```
clearable =
    (slot_now - last_clear_slot > trigger_max_slots)
 or (|price_smoothed - last_clear_price| > last_clear_price * trigger_delta_bps /
10000)
 or (|cum_skew| > trigger_skew_cap)
```

3.4 Oracle anchoring and gap-velocity control

The inspected implementation supports multiple price-source kinds, with active paths such as Pyth pull feeds and AMM-derived sources. Price reads are owner-checked and freshness-validated. Before the solver receives the index price, the gap-velocity module can bound the per-clear price step: a single extreme oracle print is clamped to a configured maximum movement from the previous clear.

```
max_step = last_clear_price * gap_velocity_cap_bps / 10000
pi_index = clamp(raw_oracle_price,
                 last_clear_price - max_step,
                 last_clear_price + max_step)
```

This is a liveness-preserving defense: the protocol does not necessarily hard-fail on a large print; it can continue clearing at a bounded index and ratchet toward the external price over successive clears.

3.5 LP absorption slope and equilibrium

The LP vault enters the clearing equation through an absorption slope `epsilon`. If an override is set, the market uses it. Otherwise the slope is derived from the vault's total assets and the market's allocated LP basis points, capped by the protocol's maximum epsilon. The solver scans sorted breakpoints, includes the index price as a virtual breakpoint, and interpolates where the monotone left-hand side crosses zero.

```
market_capital = SourVault.total_assets * allocated_lp_bps / 10000
eps_bps =
    epsilon_bps_override, if epsilon_bps_override > 0
    epsilon_bps_dynamic_scale * market_capital / SCALE_NORMALIZER, otherwise
eps_bps = min(eps_bps, max_epsilon_bps)
epsilon = eps_bps * pi_index / 10000

Clearing equation:
Phi(pi) = D(pi) + epsilon * (pi_index - pi)
Find pi* such that Phi(pi*) = 0
```

The term `epsilon * (pi_index - pi)` is positive when the candidate clearing price is below the index and negative when it is above. It therefore behaves as an LP-side absorption curve around the index. The larger the allocated LP capital and slope parameter, the more inventory the LP can absorb for a given displacement from the oracle index.

3.6 PnL, fees, and entry update

After `pi*` is solved, each position is re-evaluated at the clearing price. PnL is marked against the previous entry value. On first clear, if the stored entry is zero, the program records the new entry and emits zero PnL delta. In current v0.4 code, upsert also seeds the entry from the current oracle mark so fresh positions can close against a meaningful reference even before the next batch.

```
pnl_delta_i =
    0, if entry_old_i = 0
    effective_size_i(pi*) * (pi* - entry_old_i) / F^2, otherwise

entry_new_i = pi*
realized_pnl_new_i = realized_pnl_old_i + pnl_delta_i

fee_i = notional_i(pi*) * fee_micros / 1,000,000
```

Fee unit note: the implementation computes fees using `fee_micros / 1,000,000`. Public fee-rate claims should be audited against configured market parameters before publication.

3.7 Margin and cascade

Sour uses account-level health after a fresh clear. Health is collateral plus realized PnL over positions. Maintenance is a basis-point percentage of notional exposure. If health falls below maintenance, the cascade selector sorts positions by realized PnL ascending and zeroes losing positions until the residual account is solvent or no positions remain. Isolated mode reduces to a one-position version backed by `collateral_locked`; cross mode pools collateral across positions.

```
health = usdc_collateral + sum_i realized_pnl_i
exposure = sum_i notional_i(pi*)
maintenance_required = exposure * maintenance_margin_bps / 10000

If health < maintenance_required:
    sort positions by realized_pnl ascending
    zero worst positions until remaining health >= remaining maintenance
```

This is why the repository can truthfully describe liquidation as embedded in clearing rather than exposed as a standalone keeper instruction. A keeper or cranker is still required for liveness; the important distinction is that the keeper does not choose separate liquidation fills in a race against users. Cascade is a deterministic consequence of the batch state.

3.8 LP vault accounting

The global `SourVault` backs every market through allocation basis points. Its available assets are total assets net of outstanding positive realized PnL owed to winning positions. Deposits and withdrawals use available assets rather than raw total assets so new LPs cannot buy into an overstated NAV during the window between winner PnL accrual and close-out.

```
available_assets = total_assets - outstanding_winner_pnl

shares_for_deposit =
    amount, if total_shares = 0 or available_assets = 0
    amount * total_shares / available_assets, otherwise

usdc_for_shares =
    shares * available_assets / total_shares
```

4. Mechanism Advantages

Price-space intent

Users express conditional exposure over a price interval, not merely a direction and size. That makes risk shape a first-class on-chain object.

Uniform batch treatment

Positions included in a valid clear are settled at one common clearing price, reducing within-batch priority games that arise in serial matching.

Integrated funding-like drift

Deviations between `pi*` and `pi_index` are resolved inside the clearing equation rather than through a separate hourly funding transfer process.

Deterministic cascade

Liquidation is part of `clear_batch`. No external liquidator selects the execution moment after the clear state is known.

Shared vault capital

One global LP vault can allocate capital across markets, improving capital reuse relative to completely siloed pools.

One engine, multiple UXs

Trader Mode, Degen Mode, and future mobile flows produce the same program primitive, which limits protocol fragmentation across interfaces.

These advantages should be read precisely. Sour does not remove oracle dependence, market risk, LP inventory risk, or the need for keeper liveness. It does reduce a specific class of execution race by replacing serial ordering inside a batch with a curve aggregate and common solve. It also collapses several processes that are usually scheduled separately in other perps systems into one deterministic instruction.

5. Comparison with Major Perpetual DEX Designs

The table below compares Sour's mechanism with representative perps venues using primary protocol documentation. It focuses on mechanism design rather than token incentives, brand, or current liquidity. Latency and throughput claims for other protocols are reproduced from their official materials where cited.

PROTOCOL	EXECUTION MODEL	LIQUIDITY / COUNTERPARTY	FUNDING AND LIQUIDATION	SOUR CONTRAST
Hyperliquid	Application-specific L1 with on-chain price-time-priority order books.	Order-book makers, takers, and HLP/backstop mechanisms.	Hourly funding; liquidation interacts with book and backstop vault.	Sour does not maintain a serial order book. It clears curve demand in event batches.
dYdX Chain	Cosmos app-chain with validator-run in-memory order books and committed matched trades.	Order-book liquidity supplied by makers and takers.	Premium-based funding and protocol liquidation orders with insurance fund support.	Sour's execution object is a position curve; cascade is in the clear rather than a separate liquidation order.
GMX v2	Oracle-priced trader-to-pool perps with no traditional order book slippage.	GM/GLV liquidity pools act as trader counterparty.	Adaptive funding, borrowing fees, price impact, and collateral-threshold liquidations.	Sour is also LP-backed, but its price comes from a curve equilibrium around the index rather than direct oracle execution plus impact formulas.
Drift	Hybrid Solana venue with JIT auctions, DLOB keepers, and AMM backstop.	Makers, JIT auction participants, and AMM liquidity.	Funding from mark/oracle TWAPs; partial and progressive liquidation engine.	Sour avoids a DLOB/JIT auction path and uses one on-chain piecewise-linear solve for batch settlement.
Jupiter Perps	Solana trader-to-JLP model with oracle price, price impact, and keeper execution.	JLP pool is the main counterparty.	Borrowing, opening/closing fees, liquidation price dynamics, request-execute flow.	Sour's global vault resembles pool-backed risk, but exposure is mediated by curve brackets and uniform clears.
Synthetix Perps	Noncustodial perps with smart-contract custody, subaccounts, and maker/taker exchange UX.	SLP or protocol liquidity absorbs system risk depending on deployment architecture.	Hourly funding; liquidation can transfer or seize positions, with ADL as a backstop.	Sour integrates funding-like drift and cascade into the same event-clearing instruction.
Vertex	Hybrid off-chain sequencer order book plus on-chain AMM and risk engine.	Order-book liquidity and AMM liquidity, with cross-chain liquidity abstractions.	Funding from order-book mark and spot oracle TWAPs.	Sour sacrifices sequencer-style low latency for deterministic on-chain batch clearing semantics.

5.1 What Sour can credibly claim

Sour's comparative advantage is structural. It can credibly claim that fills are not allocated by price-time priority within a clear; that funding is not a separate hourly cron; that liquidation is not an independent liquidation keeper flow; and that the same position-curve instruction can power both professional and simplified interfaces. It should not claim deeper liquidity, lower fees, or superior realized execution quality until supported by live mainnet data.

5.2 Scientific relation to flow trading and batch auctions

Budish, Cramton, Kyle, Lee, and Malec propose replacing continuous-time trading races with flow trading: traders submit demand schedules and a market clears periodically or by event. Their earlier frequent-batch-auction work argues that discrete-time clearing can reduce arms-race incentives in latency-sensitive markets. Sour adapts that spirit to a perps setting: the demand schedules are two-breakpoint position curves, the clearing object is a synthetic perpetual exposure, and the LP slope supplies inventory absorption around the oracle index.

6. Security, Liveness, and Risk Considerations

- **Oracle risk remains fundamental.** Sour reads external prices and validates freshness, confidence, ownership, and gap velocity. These checks mitigate but do not eliminate oracle risk.
- **Keeper liveness is still required.** There is no standalone liquidation bot, but a valid `clear_batch` still has to be submitted. If no one cranks a clear, settlement and cascade do not progress.
- **LPs are real counterparties.** The global vault absorbs inventory and bad-debt risk. Cross-market capital efficiency also creates cross-market contagion if risk parameters are wrong.
- **Account-list scalability is an engineering bound.** The current design scans accounts supplied to `clear_batch`. Sharding and address lookup table management appear in repository fields and docs as future scalability work.
- **Implementation comments show active iteration.** v0.4 documents fixed precision for small bets, session keys, WebSocket state, and close-position accounting. These are healthy engineering details, but they also imply the system should be externally audited before value-bearing deployment.
- **Fees and public metrics must be parameter-checked.** The fee formula is clear, but any advertised bps rate must be derived from live `fee_micros` settings rather than copied from comments.

7. Product Implications

The same mechanism supports two user experiences. Trader Mode exposes chart-driven demand bands, market selection, portfolio state, and precision controls. Degen Mode compresses the same action into directional gestures and risk presets. Because both call the same `deriveBracket` and `upsert_position` path, UI simplification does not create a second execution engine.

Session-key delegation is another product-level implication of the architecture. The user can register a browser-held delegate for trade instructions, reducing repeated wallet popups. The delegate is not authorized to withdraw collateral or rotate itself. This is not a substitute for wallet security, but it creates a practical separation between trading ergonomics and custody rights.

The v0.4i operational documents also show why a live perps application cannot rely on naive polling. The project is moving market-maker and frontend state toward WebSocket subscriptions: bootstrap once, subscribe to account and program changes, and fall back to slower polling only when disconnected. This is an infrastructure detail, but for a high-frequency interface it is part of the protocol's practical viability.

8. Conclusion

Sour's research contribution is to recast perpetual DEX execution as event-triggered flow clearing. Position curves encode desired exposure as a function of price; a shared vault supplies absorption around the oracle index; a deterministic on-chain solver computes a common clearing price; and PnL, fees, margin checks, and cascade occur in the same instruction. This makes Sour materially different from CLOB-first perps, hybrid JIT venues, and oracle-priced pool venues.

The mechanism's strongest claim is microstructural: by aggregating curves and clearing uniformly, Sour reduces the importance of serial queue position for included flow. Its second strongest claim is architectural: funding-like drift and liquidation cascade are not separate clocked or competitive processes. The open research questions are equally clear: calibrating epsilon to market depth and volatility, proving robust behavior under sparse account inclusion, measuring LP adverse selection in live markets, auditing fee and margin units, and validating keeper liveness under network stress.

References

[S1] Sour repository README, inspected May 1, 2026. Local path: `README.md`.

[S2] Sour curve math implementation, inspected May 1, 2026. Local path: `programs/sour-math/src/curve.rs`.

[S3] Sour solver implementation, inspected May 1, 2026. Local path: `programs/sour-math/src/solver.rs`.

[S4] Sour clear-batch instruction, inspected May 1, 2026. Local path: `programs/sour/src/instructions/clear_batch.rs`.

[S5] Sour account state definitions, inspected May 1, 2026. Local path: `programs/sour/src/state.rs`.

[S6] Sour v0.4.0 combined spec and product docs, inspected May 1, 2026. Local paths: `claudedocs/sites-v0.4.0-combined-spec.md`, `docs/superpowers/specs/2026-04-27-sour-sites-design.md`.

[E1] Budish, E., Cramton, P., Kyle, A., Lee, J., and Malec, D. "Flow Trading." PDF: https://cramton.umd.edu/papers2020-2024/budish-cramton-kyle-lee-malec-flow-trading.pdf.

[E2] Budish, E., Cramton, P., and Shim, J. "The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response." *Quarterly Journal of Economics*. https://academic.oup.com/qje/article/130/4/1547/1916146.

[E3] Hyperliquid documentation, HyperCore overview, order book, oracle, funding, and liquidations. https://hyperliquid.gitbook.io/hyperliquid-docs/hypercore/overview.

[E4] dYdX Chain technical architecture, oracle, funding, and liquidation documentation. https://www.dydx.xyz/blog/v4-technical-architecture-overview.

[E5] GMX v2 documentation, introduction, liquidity, fees, and liquidations. https://docs.gmx.io/docs/intro/.

[E6] Drift protocol documentation, JIT auctions, DLOB, AMM, oracles, funding, and liquidation engine. https://docs.drift.trade/protocol/about-v3/jit-faq.

[E7] Jupiter Perps and JLP user documentation. https://docs.jup.ag/user-docs/trade/perps-and-jlp.

[E8] Synthetix documentation corpus, perps and exchange architecture. https://docs.synthetix.io/llms-full.txt.

[E9] Vertex Protocol documentation, overview, technical architecture, and oracles. https://docs.vertexprotocol.com/getting-started/overview.

[E10] Pyth Network whitepaper and price-feed documentation. https://docs.pyth.network/whitepaper.

[E11] Liu, B., Szalachowski, P., and Zhou, J. "A First Look into DeFi Oracles." https://arxiv.org/abs/2005.04377.

[E12] Qin, K., Zhou, L., Gamito, P., Jovanovic, P., and Gervais, A. "An Empirical Study of DeFi Liquidations: Incentives, Risks, and Instabilities." https://arxiv.org/abs/2106.06389.

[E13] Xu, J., Paruch, K., Cousaert, S., and Feng, Y. "SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) protocols." https://arxiv.org/abs/2103.12732.