## MNQRSTest\_EntryLogic.cs (Strategy Core)

/// Module: MNQRSTest (Strategy Core)  
/// Purpose: Orchestrates entry signals, quality gating, order placement, and module coordination for the strategy.  
/// Inputs:  
/// - Market data: primary Bars series (OHLC prices, 1-min timeframe)  
/// - Feature toggles: UseQualityGate, UseVolumeProfile, ApplyRunnerManagement (bool, default false for observe-only mode)  
/// - Quality threshold: MinQTotal2 (double ∈ [0,1], e.g. 0.60) – minimum quality score to allow entry if gating enabled  
/// Outputs:  
/// - Trade orders executed (EnterLong/Short calls) when entry conditions and gates are satisfied  
/// - Position management actions (stop/target submits via other modules)  
/// Timing/State:  
/// - Called on each bar close (OnBarUpdate with BarsInProgress==0) for signal evaluation and module updates  
/// - Utilizes NinjaTrader strategy state events (e.g., OnStateChange for initialization, OnExecution for fills) as needed  
/// Assumptions/Invariants:  
/// - Indicators and quality metrics are warmed up (sufficient history) before signals are acted on  
/// - Strategy runs on a single instrument/timeframe; session trading hours are configured to filter out off-session bars  
/// - All quality scores (Q\_\*) are normalized to [0,1] and feature flags default to off (no gating or dynamic sizing by default)  
/// Side Effects:  
/// - Places entry and exit orders (modifies account position) when conditions meet criteria  
/// - Invokes module logic: QualityGate for entry permission, RunnerLogic for position split, VPManagement for trailing stops  
/// - Triggers telemetry logging for each decision (e.g., writes a CSV line on entry taken or skipped)  
/// Telemetry contract:  
/// - No unique fields beyond module outputs – ensures BatchTag is included in logs and that all computed metrics (Q\_Total2, etc.) are passed to telemetry  
/// Failure Modes:  
/// - If quality metrics are NaN (insufficient data) or gating threshold not met (when enabled), strategy will skip entry (no order placed)  
/// - If an entry order is rejected (e.g., due to quantity or margin), the strategy records the failure but continues on next signals  
/// - If strategy quantity is 1 and runner split is requested, it defaults to no runner (entire position taken at primary target)  
/// Example:  
/// <code>  
/// // On each bar close:  
/// double qTotal = ComputeQuality();   
/// bool allowTrade = !UseQualityGate || qTotal >= MinQTotal2;  
/// if (entrySignalLong && allowTrade) {  
/// // Split position if runner management is on  
/// if (ApplyRunnerManagement) { EnterLong(GetTargetQuantity(), "Primary"); EnterLong(GetRunnerQuantity(), "Runner"); }  
/// else { EnterLong(DefaultQuantity); }  
/// }  
/// // Telemetry logging of decision  
/// Telemetry.LogDecision(CurrentBar, allowTrade, qTotal);  
/// </code>

## MNQRSTest\_PosVolLogic.cs (PosVol logic)

/// Module: MNQRSTest.PosVolLogic (Directional Volume Quality)  
/// Purpose: Computes a directional volume quality score (Q\_PosVol) indicating how strongly recent volume flow aligns with the trade direction.  
/// Inputs:  
/// - VolBuy, VolSell (double, volume at ask and bid for the bar; units = contracts) – order-flow data per bar (if unavailable, uses total Volume and price change as proxy)  
/// - IsLong (bool) – context of evaluation (true for long-side evaluation, false for short)  
/// - Internal parameters: nSmooth=8 (EMA period), nDelta=14, lookbackZ=100 (bars for z-score baseline) – define smoothing and normalization window (constants in default mode)  
/// Outputs:  
/// - Q\_PosVol (double ∈ [0,1]) – directional volume quality score for the current bar (higher = volume confirms direction, 0.5 = neutral)  
/// Timing/State:  
/// - Calculated on each bar close (OnBarUpdate for primary series); uses latest bar’s volume and price data  
/// - Relies on an EMA and z-score calculation over a rolling window (requires warm-up of ~100 bars for stable values)  
/// Assumptions/Invariants:  
/// - Volume data is available for each bar (if true bid/ask breakdown is missing, Volume is split by price movement as an approximation)  
/// - All intermediate calculations (buy percentage, delta z-score) are clamped or squashed into [0,1] to produce a bounded Q\_PosVol  
/// - Small epsilon added to volume totals to avoid division by zero (ensures BuyPct is defined even on low-volume bars)  
/// Side Effects:  
/// - None (pure function: computes and returns Q\_PosVol; does not modify state outside its internal EMA buffers)  
/// - The strategy stores Q\_PosVol in a series/field each bar for use by other modules (e.g., momentum logic, quality gating)  
/// Telemetry contract:  
/// - Adds column \*\*Q\_PosVol\*\* (double [0,1]) to CSV output for each decision or trade, representing the volume flow quality on that bar  
/// - Q\_PosVol is an input to FavMomo/TrueMomo metrics (logged separately) but is itself recorded for analysis  
/// Failure Modes:  
/// - If VolBuy/VolSell data is absent or volume is zero, outputs Q\_PosVol ≈ 0.5 (neutral) by design to avoid skewing decisions  
/// - During initial bars (fewer than lookbackZ), the z-score normalization uses partial history (early Q\_PosVol may be less reliable until warm-up complete)  
/// - Extreme volume spikes or outliers are tempered by the tanh squash, preventing Q\_PosVol from exceeding [0,1]  
/// Example:  
/// <code>  
/// // Compute Q\_PosVol for current bar:  
/// double buyPct = Clamp01(VolBuy / Math.Max(VolBuy + VolSell, 1e-9));  
/// double dirScore = IsLong ? buyPct : (1.0 - buyPct);  
/// double dirSmoothed = EMA(dirScore, nSmooth);  
/// double deltaZ = ZScore(EMA(VolBuy - VolSell, nDelta), lookbackZ);  
/// double strength = 0.5 \* (Math.Tanh(deltaZ) + 1.0);  
/// double rawQ = 0.6 \* dirSmoothed + 0.4 \* (dirSmoothed \* strength);  
/// double qPosVol = Clamp01(rawQ);  
/// </code>

## MNQRSTest\_MomentumLogic.cs (MomentumCore logic)

/// Module: MNQRSTest.MomentumLogic (Momentum Quality Aggregation)  
/// Purpose: Produces a momentum quality metric that integrates pure price momentum with volume confirmation and regime context.  
/// Inputs:  
/// - Base momentum metric (PosMomo or Q\_Momo, double ∈ [0,1]) – baseline momentum quality from price action (e.g., breakout or trend strength)  
/// - Q\_PosVol (double ∈ [0,1]) – directional volume quality (to amplify or dampen momentum via FavMomo)  
/// - Regime context metrics: Q\_Space, Q\_Trend (double ∈ [0,1] each) – optional broader context signals (e.g., congestion vs trending) to blend momentum measures  
/// - Internal weight parameters: lambda=0.75 (volume influence for FavMomo), blend formula uses equal parts Q\_Space and Q\_Trend to determine blending factor  
/// Outputs:  
/// - TrueMomo (double ∈ [0,1]) – final momentum quality score adjusted by volume and regime (higher = strong volume-backed momentum)  
/// - FavMomo (double ∈ [0,1]) – intermediate volume-favored momentum (momentum score after volume bias applied)  
/// Timing/State:  
/// - Computed each bar close after Q\_PosVol is available (as part of quality metric update before gating or sizing decisions)  
/// - Uses current bar’s metrics (assumes base momentum and context signals are updated for the bar)  
/// Assumptions/Invariants:  
/// - All input metrics are normalized to [0,1] (baseline momentum, volume quality, and context signals)  
/// - If context metrics (Q\_Space, Q\_Trend) are not available or not implemented, a neutral blend is assumed (e.g., treat 0.5 as neutral congestion/trend mix)  
/// - TrueMomo and FavMomo are clamped to [0,1] after calculation to prevent any overshoot due to weighting  
/// Side Effects:  
/// - None (pure calculations); stores the resulting FavMomo/TrueMomo in strategy fields for later use (e.g., quality gating, runner logic)  
/// Telemetry contract:  
/// - Adds columns \*\*FavMomo\*\* and \*\*TrueMomo\*\* (double [0,1]) to the CSV output for each trade/decision point  
/// - May log the baseline momentum (PosMomo or Q\_Momo) as well for reference, ensuring Python parity in momentum calculation  
/// Failure Modes:  
/// - If the baseline momentum metric is NaN or not yet initialized, defaults to 0.5 (neutral momentum) which yields FavMomo ≈ 0.5  
/// - If volume data (Q\_PosVol) is missing, FavMomo = baseline momentum (no amplification/attenuation)  
/// - In absence of any regime context, TrueMomo defaults to FavMomo (alpha = 0.0 if Q\_Space/Q\_Trend undefined, meaning favor volume-weighted momentum fully)  
/// Example:  
/// <code>  
/// // Calculate volume-favored momentum and true momentum:  
/// double baseMomo = PosMomo; // baseline momentum quality from price  
/// double favMomo = Clamp01(baseMomo \* (1.0 + 0.75 \* (Q\_PosVol - 0.5)));  
/// double regimeBlend = Clamp01(0.5 \* Q\_Space + 0.5 \* Q\_Trend);  
/// double alpha = 1.0 - regimeBlend; // weight for baseMomo vs vol-favored  
/// double trueMomo = Clamp01(alpha \* baseMomo + (1 - alpha) \* favMomo);  
/// </code>

## MNQRSTest\_VPLogic.cs (VPManagement logic)

/// Module: MNQRSTest.VPManagement (Volume Profile Trail & Runner Bias)  
/// Purpose: Decides the trailing stop mode (ATR vs SR-based) and adjusts runner position bias based on volume profile tailwind/headwind signals, with hysteresis to avoid oscillation.  
/// Inputs:  
/// - lastVP\_Congestion (int or bool ∈ {0,1}) – whether the previous state was in a congestion regime (1 = congested, 0 = trending)  
/// - VP\_TW, VP\_HW (double ∈ [0,1]) – volume profile tailwind and headwind scores (e.g., 1 if strong tailwind/support behind price, 1 if strong headwind/resistance ahead; typically one or both 0)  
/// - lastTrailType (enum {ATR, SR}) – the last applied trailing stop type  
/// - HysteresisBars (int, e.g., 10) – number of consecutive bars a regime signal (congestion or trend) must persist before switching trail type  
/// Outputs:  
/// - TrailType (enum {ATR, SR}) – chosen trailing stop mode for current conditions (ATR = volatility-based, SR = support/resistance level-based)  
/// - RunnerPct (double ∈ [0,1]) – adjusted fraction of position to keep as runner (1 = full runner, 0 = no runner) based on updated profile bias  
/// Timing/State:  
/// - Invoked on each bar close (BarsInProgress==0) when a position with a runner is active, to evaluate if trailing stop mode or runner portion needs updating  
/// - Employs hysteresis: state must remain congested or trending for HysteresisBars before a switch occurs, preventing rapid flip-flops bar-to-bar  
/// Assumptions/Invariants:  
/// - Volume profile metrics (VP\_TW/HW) are precomputed each bar (e.g., via an indicator) and normalized to [0,1]  
/// - At most one trailing mode switch can occur within a HysteresisBars window; initial TrailType defaults to ATR (trend mode) unless congestion persists  
/// - All quality inputs (Q\_\* metrics influencing tailwind/headwind) are clamped [0,1], ensuring consistent switching thresholds  
/// Side Effects:  
/// - Updates internal strategy state: sets the current TrailType for stop management and possibly modifies RunnerPct used for remaining position  
/// - Triggers an order update if TrailType changes (e.g., cancels existing stop and submits new stop according to the selected mode on the next tick)  
/// - Logs the volume profile state and decisions to telemetry (no direct file I/O here, deferred to telemetry module)  
/// Telemetry contract:  
/// - Adds columns \*\*VP\_TW\*\*, \*\*VP\_HW\*\*, \*\*VP\_Cong\*\* (congestion flag/score), \*\*TrailType\*\*, \*\*RunnerPct\*\* to the output  
/// - These fields are recorded each time a decision is made (typically at entry and when a switch condition is met) for debugging and parity checks  
/// Failure Modes:  
/// - If VP indicators fail to update or return NaN (e.g., not enough data for profile), the logic falls back to maintaining the last TrailType and RunnerPct (no change)  
/// - In extremely choppy conditions (congestion signal flickers), the hysteresis may delay needed switches or cause minor trailing inefficiency (accepted trade-off for stability)  
/// - If no position or no runner portion is active, the function exits without action (no outputs to apply)  
/// Example:  
/// <code>  
/// // Example trailing decision within strategy OnBarUpdate:  
/// if (Position.MarketPosition != MarketPosition.Flat && BarsSinceEntry > 0) {  
/// (TrailType newType, double newRunnerPct) = VPManagement.DecideTrailing(lastTrailType, VP\_TW, VP\_HW, lastVP\_Congestion);  
/// if (newType != lastTrailType) UpdateStopsToType(newType);  
/// AdjustRunnerPosition(newRunnerPct);  
/// }  
/// </code>

## MNQRSTest\_QualityLogic.cs (QualityGate logic)

/// Module: MNQRSTest.QualityGate (Q\_Total2 Gating)  
/// Purpose: Computes an overall quality score (Q\_Total2) for the trade setup and optionally gates trade entry based on a minimum quality threshold.  
/// Inputs:  
/// - Component quality metrics (double ∈ [0,1] each): e.g., momentum quality, volume quality (Q\_PosVol), trend quality, space/runway quality, support/resistance quality  
/// - MinQTotal2 (double ∈ [0,1]) – configured minimum composite quality required to allow a trade (if gating is enabled)  
/// - UseQualityGate (bool) – flag indicating whether to enforce gating (when false, Q\_Total2 is still calculated but does not block trades)  
/// Outputs:  
/// - Q\_Total2 (double ∈ [0,1]) – composite quality score aggregating multiple factors (higher = better overall trade quality)  
/// - AllowTrade (bool) – gating decision (true if trade is permitted given quality, or if gating is off; false if quality below threshold when gating on)  
/// Timing/State:  
/// - Evaluated on each bar close, prior to order entry; specifically when a potential entry signal is present, to decide if it should proceed  
/// - Q\_Total2 may be continuously updated each bar for logging even if no signal, but gating decision is only relevant at entry moments  
/// Assumptions/Invariants:  
/// - All contributing quality metrics are normalized and up-to-date for the current bar (calculated in other modules or earlier in this module)  
/// - The weighting of each component in Q\_Total2 is fixed or configurable but remains constant during a run (no dynamic re-weighting mid-run except via parameter tweaks between runs)  
/// - Default operation is observe-only (UseQualityGate=false), meaning Q\_Total2 is calculated and logged but does not inhibit trades  
/// Side Effects:  
/// - When UseQualityGate=true and Q\_Total2 < MinQTotal2, the strategy will skip placing the entry order (effectively filtering out low-quality trade signals)  
/// - Logs the quality score (and gate pass/fail) to telemetry for each evaluated entry decision  
/// - May influence position sizing logic if quality-weighted sizing is implemented (in future, high quality could increase trade size – not active in default mode)  
/// Telemetry contract:  
/// - Adds column \*\*Q\_Total2\*\* (double [0,1]) to CSV output for each trade decision, representing the overall quality score of that setup  
/// - The gating outcome (pass/fail) is reflected implicitly (e.g., a skipped trade will be noted with its Q\_Total2 in the log and no corresponding entry execution)  
/// Failure Modes:  
/// - If one or more component metrics are NaN or missing (e.g. due to insufficient data), Q\_Total2 is calculated with available terms; missing terms default to neutral (0.5) to avoid bias, or the composite may be marked invalid (leading to a skip)  
/// - Extremely high or low values in components are handled by normalization/clamping in their modules, preventing Q\_Total2 from being out of [0,1] range  
/// - Misconfigured MinQTotal2 (e.g., >1) is clamped internally or results in no trades if threshold is unattainable  
/// Example:  
/// <code>  
/// double qTotal = ComputeCompositeQuality(); // aggregate momentum, volume, space, etc.  
/// bool allow = !UseQualityGate || qTotal >= MinQTotal2;  
/// if (!allow) { /\* Skip trade due to low quality \*/ }  
/// else { /\* Place order \*/ }  
/// // Log the quality score regardless of gating  
/// telemetry.WriteValue("Q\_Total2", qTotal);  
/// </code>

## MNQRSTest\_RunnerLogic.cs

/// Module: MNQRSTest.RunnerLogic (Runner Eligibility & Scaling)  
/// Purpose: Determines if a trade should have a runner (secondary position to ride extended move) and what fraction of the position to allocate to that runner, based on quality metrics and context.  
/// Inputs:  
/// - Q\_ResRunner (double ∈ [0,1]) – runway quality for runner (e.g., quality of nearby support/resistance allowing further run; higher means more room to run)  
/// - Q\_PosMomo (double ∈ [0,1]) – momentum quality at entry (volume-positioned momentum, indicating strength of move initiation)  
/// - Volume profile context: VP\_Tailwind, VP\_Headwind (double ∈ [0,1]) – profile-based bias (tailwind support or headwind resistance) to adjust runner size (used as modifiers)  
/// - ApplyRunnerManagement (bool) – flag to enable actual runner split; default false (no runner placed, though logic still calculates for telemetry)  
/// Outputs:  
/// - RunnerEligible (bool) – whether conditions meet the criteria to employ a runner (true if runner should be used, false if take full profit at primary target)  
/// - RunnerPct (double ∈ [0,1]) – the fraction of the total position to allocate to the runner if eligible (remaining portion goes to primary target leg)  
/// Timing/State:  
/// - Evaluated at entry signal time (during order placement in OnBarUpdate) to decide order splitting before trade execution  
/// - Not typically recalculated during the trade (initial decision remains, subsequent adjustments handled by VPManagement if any)  
/// Assumptions/Invariants:  
/// - Runner criteria thresholds are moderate (e.g., requires Q\_ResRunner >= 0.5 and Q\_PosMomo >= 0.5 in default logic) to ensure runner only when quality is decent  
/// - Base runner fraction is fixed (e.g., 50% of position) when criteria just meet; profile tailwind/headwind can increase or decrease this fraction by a set amount (e.g., ±0.2) but final RunnerPct is clamped [0,1]  
/// - If total order size is 1 (unit), runner is automatically deemed ineligible (cannot split a single unit; RunnerPct forced to 0)  
/// Side Effects:  
/// - If RunnerEligible is true (and ApplyRunnerManagement=true), strategy will place orders in two parts: primary target and runner portion, each with its own exit strategy  
/// - If false or runner management disabled, entire position goes to the primary target (no second leg); still logs what RunnerPct \*would\* have been for analysis  
/// - Updates internal fields like lastRunnerPct for use by other modules (e.g., VPManagement might reference this to adjust runner during trade)  
/// Telemetry contract:  
/// - Adds columns \*\*RunnerEligible\*\* and \*\*RunnerPct\*\* to the CSV logs for each trade entry decision  
/// - These reflect the initial runner decision at entry (subsequent changes due to VPManagement will also be logged via that module’s RunnerPct updates)  
/// Failure Modes:  
/// - If required input metrics are missing (e.g., no prior bar data for Q\_ResRunner or Q\_PosMomo), defaults to RunnerEligible=false (no runner) to be safe  
/// - If profile bias inputs (VP\_Tailwind/Headwind) are not available at entry, no bias adjustment is applied (uses base RunnerPct)  
/// - Rounding issues: for small lot sizes, RunnerPct is effectively quantized (e.g., if total size=2 and RunnerPct=0.6, actual runner gets 1 unit = 50%)  
/// Example:  
/// <code>  
/// // Decide runner allocation at entry:  
/// bool runnerOK = (Q\_ResRunner >= 0.5 && Q\_PosMomo >= 0.5);  
/// double runnerFrac = runnerOK ? 0.5 : 0.0;  
/// // Apply volume profile bias:  
/// if (VP\_Tailwind > 0.5) runnerFrac += 0.2;  
/// if (VP\_Headwind > 0.5) runnerFrac -= 0.2;  
/// runnerFrac = Clamp01(runnerFrac);  
/// RunnerEligible = runnerOK;  
/// RunnerPct = runnerFrac;  
/// </code>

## MNQRSTest\_Logging.cs (Telemetry logic)

/// Module: MNQRSTest.Telemetry (CSV & HardLock Logging)  
/// Purpose: Collects and outputs key strategy metrics and events to external logs (CSV file for batch analysis, HardLock for live trade audit) to facilitate cross-engine parity and performance tracking.  
/// Inputs:  
/// - Strategy metrics and state: all quality scores (Q\_\*), decision outcomes (trade taken/skipped), trade details (entry/exit prices, PnL), etc., passed in from various modules when events occur  
/// - BatchTag (string) – identifier for the current batch run or configuration (appended to output for grouping results)  
/// - Export configuration: file path/name pattern (from Exporter\_AddOn settings) for CSV output; HardLock endpoint or object for live logging  
/// Outputs:  
/// - CSV log entries: each significant event (typically each trade entry/exit or decision point) is recorded as a line in a CSV file with all relevant fields (see TELEMETRY\_SCHEMA)  
/// - HardLock log records: real-time trade and performance data recorded to HardLock (in-memory or external service) for live monitoring (no file output, used for audit trail)  
/// Timing/State:  
/// - CSV logging is typically buffered during strategy run and flushed on termination (OnStateChange->State.Terminated), or written incrementally after each event depending on AddOn design  
/// - Logging calls occur on events: after an entry decision (taken or skipped), after a trade exit (to log outcome), or when trailing/runner adjustments happen, ensuring a chronological record  
/// Assumptions/Invariants:  
/// - The set of telemetry fields is consistent across runs and matches between C# and Python implementations (schema is fixed; both engines output identical column names for comparison)  
/// - BatchTag is unique per run or scenario to distinguish aggregated results (and is included as a column or in the file name as needed)  
/// - Writing to files is permitted and the configured output directory exists; if not, the module will attempt to create it or fail gracefully  
/// Side Effects:  
/// - Performs file I/O (writes to CSV on disk), which can impact performance if logging too frequently (in practice, limited to key points to keep overhead low)  
/// - Maintains static or persistent data structures for accumulating log entries (e.g., a list of records) which are cleared on strategy start and flushed on end  
/// - HardLock logging may send data to an external system or static memory that is read by other components (ensures no sensitive data leaks beyond intended scope)  
/// Telemetry contract:  
/// - Defines and requires the full set of CSV columns (see TELEMETRY\_SCHEMA.md) – including timestamp, trade outcomes, and all quality metrics – so that the Python engine can parse and match them one-to-one  
/// - Uses BatchTag in each record (or file name) to label the dataset, enabling grouping of results by experimental run  
/// - No A/B test or experimental fields are included in logs by default (only default-mode metrics are recorded to keep schema stable)  
/// Failure Modes:  
/// - File write failures (e.g., permission issues, disk full) will cause logging to be skipped or halted; the strategy itself continues running but without full telemetry (an error is typically reported to user)  
/// - If an expected metric is NaN or not set, the output will record it as blank or a default value (ensuring the CSV remains parseable); any mismatch in schema between C# and Python will be caught in CI  
/// - HardLock logging failures (e.g., disconnected target) do not stop strategy execution; data may be lost or partially recorded in such cases  
/// Example:  
/// <code>  
/// // Pseudocode for logging a trade decision:  
/// string record = $"{Time[0]},{BatchTag},{(isLong?\"Long\":\"Short\")},{entrySignal},{Q\_Total2},{Q\_PosVol},{TrueMomo},...,{tradeTaken}";  
/// CsvExporter.WriteLine(record);  
/// HardLockLogger.LogEvent(Time[0], "ENTRY", tradeTaken, Q\_Total2);  
/// </code>

## DEPENDENCIES.md

* **Strategy core** – Calls *QualityGate* to evaluate overall quality (uses Q\_Total2 from QualityLogic), calls *RunnerLogic* to determine order split, calls *VPManagement* for trailing adjustments during trades, and invokes *Telemetry* logging at key events. Uses standard NinjaTrader indicators (e.g., ATR for stops) and Volume Profile data (via QVP indicator) indirectly through VPManagement.
* **QualityGate** – Calls *PosVolLogic* to include volume flow quality in Q\_Total2 calculation and calls *MomentumLogic* (MomentumCore) to incorporate momentum metrics. Also relies on any pre-computed quality sub-metrics (trend, space, etc.) but does not call other modules beyond those providing input metrics.
* **PosVolLogic** – Self-contained; uses low-level utilities (EMA, z-score calculations) on volume data. No calls to other strategy modules (inputs provided by data feed or strategy context).
* **MomentumLogic** – Uses output from *PosVolLogic* (Q\_PosVol) and reads other quality metrics (Q\_Space, Q\_Trend) if available. No direct calls to other modules; computations are internal using provided inputs.
* **VPManagement** – Uses Volume Profile information (from an indicator or internal calc) to derive tailwind/headwind metrics. Reads the last state (trail type, congestion status) and *may update RunnerLogic’s output* (RunnerPct) during a trade. No direct calls to other modules, but relies on data from quality metrics (for congestion signal) and influences runner management.
* **RunnerLogic** – Reads quality metrics from *QualityGate/MomentumLogic* (e.g., Q\_ResRunner, Q\_PosMomo) and Volume Profile bias from *VPManagement* (tailwind/headwind values) to decide runner allocation. Calls NinjaTrader order methods to place primary/runner orders. No direct invocation of other modules (inputs are fed in from strategy core).
* **Telemetry** – Does not call strategy modules; instead, it is called by them. Receives data from all above modules (as parameters to log functions). Interfaces with the external *CSV Exporter AddOn* and *HardLock logger* to output data. No internal dependencies on strategy logic (focus is output only).

## TELEMETRY\_SCHEMA.md

The CSV telemetry output includes the following fields for each logged event (typically one row per trade or decision point):

* **timestamp** (DateTime): Date/time of the event (e.g., entry bar time)
* **side** (string): "Long" or "Short" indicating trade direction
* **pnl** (double): Profit or loss of the trade (in currency units)
* **mfe** (double): Maximum favorable excursion – peak unrealized profit during the trade (e.g., ticks or currency)
* **mae** (double): Maximum adverse excursion – largest unrealized drawdown during the trade
* **VolBuy** (double): Volume traded at the ask (buy volume) on the entry bar
* **VolSell** (double): Volume traded at the bid (sell volume) on the entry bar
* **Delta** (double): VolBuy minus VolSell (net aggressive volume on entry bar)
* **BuyPct** (double ∈ [0,1]): Proportion of volume that was buyer-initiated on the entry bar
* **Q\_PosVol** (double ∈ [0,1]): Directional volume quality score for the entry context
* **FavMomo** (double ∈ [0,1]): Volume-favored momentum (momentum quality after volume bias)
* **TrueMomo** (double ∈ [0,1]): Final momentum quality accounting for market regime
* **PosVolVP\_short** (double ∈ [0,1]): Short-horizon volume-at-price quality (volume at favorable price on shorter profile)
* **PosVolVP\_long** (double ∈ [0,1]): Long-horizon volume-at-price quality (on a longer-term profile)
* **PosVolVPmix** (double ∈ [0,1]): Combined volume-at-price score (blend of short/long horizon)
* **Q\_Res** (double ∈ [0,1]): Base support/resistance level quality (structural quality of nearest level)
* **Q\_PosRes** (double ∈ [0,1]): Support/resistance quality adjusted for volume flow at that level
* **Q\_Momo** (double ∈ [0,1]): Baseline momentum quality (price-action momentum score)
* **PosMomo** (double ∈ [0,1]): Alternate representation of momentum (e.g., positional momentum without volume influence)
* **Q\_Space** (double ∈ [0,1]): Runway/space quality (distance to nearest obstacle or room for price to move)
* **Q\_Trend** (double ∈ [0,1]): Higher-level trend quality metric (trend strength confirmation)
* **QTotal2\_Old** (double ∈ [0,1]): Original composite quality score (prior to including new PosVol/Momo factors)
* **QTotal2\_New** (double ∈ [0,1]): Updated composite quality score (with PosVol, FavMomo, TrueMomo, etc. integrated)
* **VP\_TW** (double ∈ [0,1]): Volume Profile tailwind indicator (degree of support from below for long / above for short)
* **VP\_HW** (double ∈ [0,1]): Volume Profile headwind indicator (degree of resistance ahead for long / support below for short)
* **VP\_Cong** (int/bool ∈ {0,1}): Congestion regime flag (1 if market was in congestion around entry, 0 if in clear trend)
* **TrailType** (string): Trailing stop mode used for the trade ("ATR" or "SR")
* **RunnerEligible** (int/bool ∈ {0,1}): 1 if a runner portion was enabled for the trade, 0 if not
* **RunnerPct** (double ∈ [0,1]): Fraction of the position allocated to the runner (e.g., 0.5 means 50% of position was left to run)
* **BatchTag** (string): Label for the test batch or configuration of this run (helps group and identify runs in analysis)

Each field is recorded for both the C# strategy and the Python replica to ensure parity. All numeric fields are within the ranges specified above, and boolean flags are output as 0/1 for consistency. The CSV header will include all column names exactly as listed, and each row will contain the values for a single trade or decision event.