ChatGPT

FINBOT: Comprehensive Blueprint for an Algorithmic Trading System

Part 1 — The Trading Industry (Masterclass)

Global Trading Ecosystem

The global trading ecosystem consists of interconnected institutions and venues that facilitate the buying and selling of financial securities. **Exchanges** (e.g. NSE, BSE, NYSE) provide electronic order books and matching engines where buyers and sellers transact. These matching engines are computer systems (often hundreds of servers with low-latency hardware) that continuously match incoming orders against existing limit orders [1] [2]. For example, NSE and BSE (India's main exchanges) both operate fully electronic limit order books managed by central matching engines [3]. In commodities, the MCX (Multi Commodity Exchange) in India – launched in 2003 – handles futures trading under SEBI regulation [4].

Market participants include **brokers and dealers**, who provide market access. Brokers use **Order Management Systems (OMS)** to manage the lifecycle of client orders (entry, monitoring, allocation) and **Execution Management Systems (EMS)** to route orders to markets [5] [6]. An OMS tracks orders end-to-end (including compliance and reporting), while an EMS is optimized for real-time order placement and market data handling [5] [6]. For example, an EMS often provides advanced order types (algos, basket orders) and direct market connectivity, whereas an OMS maintains full audit trails and allocations [5] [6].

Once trades are executed on exchanges, **clearing corporations (CCPs)** step in as central counterparties. The CCP guarantees settlement by interposing itself between buyer and seller, mitigating counterparty risk [7]. In India, CCIL clears fixed-income, while stock trades on NSE/BSE are settled via NSCCL and ICCL (NSE's clearing houses) under SEBI oversight [3]. Globally, entities like DTCC (U.S.) and LCH (Europe) play similar roles.

Liquidity providers and **market makers** help maintain orderly markets. A market maker continuously quotes two-sided prices (bid and ask) for a security, profiting from the spread. A market maker's commitment often comes with obligations (e.g. quoting within a certain spread). Other liquidity providers (large institutions, prop shops) may supply depth by posting large orders or participating in dark pools. For instance, modern market makers use algorithms to post staggered limit orders capturing tiny spread profits [8] [9]. In contrast, some liquidity providers underwrite big OTC trades – guaranteeing large block executions for institutional clients [9].

Retail vs. Institutional: Retail traders are individual investors trading relatively small amounts through brokerage platforms. They typically pay flat commissions, trade round lots, and have less market impact [10]. Institutional traders (hedge funds, mutual funds, banks) trade large blocks (often tens of thousands of shares) and use sophisticated tools like DMA. Institutions negotiate basis-point commissions and often split orders to avoid moving prices [11] [10]. Because institutional orders are large, they can influence prices, so institutions may hide or slice orders to minimize market impact.

Order routing and latency: An order from a trading strategy passes through multiple layers: the algorithm generates a signal, risk checks are applied, the order is sent (often via FIX) through an OMS/EMS, over the internet to the broker or exchange gateway, and finally to the exchange matching engine. **Latency** accumulates at each step (network hops, processing delays) [12] [13]. Low-latency trading firms often **co-locate** servers in exchange data centers to cut physical distance (fiber length) [13] and use **Direct Market Access (DMA)** to bypass traditional broker paths [14]. Smart order routers (SOR) may split orders across multiple venues to find best price, but this adds complexity and potential delays.

Fees, taxes, and regulation: Trading costs vary by market. In India, brokers charge 18% GST on their brokerage [15]. Exchanges levy transaction fees (often 0.001% of turnover) and SEBI charges a tiny turnover fee (~₹10 per crore, i.e. 0.0001%) [16]. The **Securities Transaction Tax (STT)** is 0.1% on delivery equity trades (both buy and sell) [17], and 0.025% on intraday. A stamp duty of ~0.015% is paid on delivery trades [18]. Globally, tax regimes differ: e.g., the U.S. has no STT, though a small SEC fee on sales was recently reduced to $0 [19]. The UK still has a stamp duty (0.5% on most equity transfers). **Regulations** are enforced by bodies like India's SEBI, the U.S. SEC/CFTC, and Europe's ESMA/MiFID II. SEBI mandates audits for trading algorithms (SAS requirements) and has tightened algo rules (e.g. requiring strategy approval, risk controls). Exchanges use price bands and circuit breakers to curb volatility. In short, trading must comply with local rules: trade reporting, position limits, best execution obligations, etc., which vary by jurisdiction.

Market Microstructure

Market microstructure studies how order flow and trading mechanics determine price formation. At its core is the **Limit Order Book (LOB)** – the collection of all outstanding limit orders at each price level [2]. Buy orders (bids) and sell orders (asks) populate the LOB. The **best bid** (highest buy price) and **best ask** (lowest sell price) define the **inside market** and the **spread** (ask – bid). The **mid-price** is (best bid+ask)/2.

LOB dynamics: When a market order arrives, it "hits" the LOB consuming liquidity at the best price(s), potentially moving the mid-price. Between trades, participants add or cancel orders, changing depth. Key microstructure concepts include:

- **Ticks and Spreads**: Every exchange has a minimum price increment (tick). Tight tick sizes allow fine price discovery. The **spread** reflects immediate liquidity: a wide spread suggests low liquidity or high uncertainty, whereas a narrow spread indicates a liquid, competitive market.

- **Imbalance and Flow**: **Order book imbalance** measures buy vs. sell side pressure. For example, the *static imbalance* over N levels is (Σ bid sizes – Σ ask sizes)/(Σ bid + Σ ask) [20]. A large positive imbalance (more bids) signals buying pressure. Dynamic imbalances (with time-weighting or volume adjustments like VAMP) can predict short-term price pressure [21]. Related is **Cumulative Volume Delta (CVD)**: tracking net buying vs. selling volume over time. If prices fall but CVD is strongly positive (many buyers), a reversal may be imminent.

- **Slippage and Impact**: **Slippage** is the difference between expected execution price and actual fill price. It arises because market orders walk the book, especially when liquidity is thin. **Market impact** risk grows with order size relative to book depth. Strategies must account for depth and implement limit or sliced orders to mitigate impact.

- **Latency Layers**: Several latency components matter. **Network latency** includes physical distance and hops. **Broker latency** includes order routing and checks. **Exchange latency** is the time an exchange takes to match and confirm an order. Internally, the trading system (signal generation, risk checks) also adds latency. HFT firms meticulously measure “tick-to-trade” time (market data to order) and round-trip times [12]. Reducing latency often requires specialized hardware (FPGA, kernel bypass), but for mid-frequency (seconds-scale) trading, well-designed software and fast networks suffice.

- **Price Formation & Liquidity Flow**: Price moves as new information or order imbalances shift expectations. Large buy orders deplete ask side and push prices up; conversely, large sells push prices down. **Liquidity flow** is also influenced by auctions: Many exchanges have periodic call auctions (e.g., opening/closing auctions) where orders accumulate and match at a single clearing price, concentrating volume. Outside auctions, continuous trading prevails. Flash events (e.g. 2010 “Flash Crash”) occur when microstructure breaks down (liquidity evaporates, orders cascade).

Ultimately, microstructure determines the edges available to high-frequency traders. Tiny inefficiencies (bid-ask bounce, spread capture, fleeting liquidity imbalances) are the sources of profit. HFT and MFT strategies must model the LOB dynamics intimately, since any edge is erased if execution speed or book understanding is inferior [22].

Types of Trading Strategies

Trading strategies span a wide spectrum of time horizons and logic. Below is an overview of key categories, with their core ideas, typical edges, and risks – noting where each might fit into a FINBOT multi-strategy system.

- **High-Frequency Trading (HFT)**: Ultra-fast trades held for milliseconds. Edges come from statistical order flow patterns, arbitrage across venues, market making (capturing spread), latency arbitrage. Requires co-location, direct feeds, and lean code. *Risks:* adverse selection, flash crashes, technological failures. *FINBOT:* Without co-location, HFT is out of scope, but we can borrow microstructure insights (like imbalance scalping) at slightly slower speeds.

- **Mid-Frequency Trading (MFT)**: Trade holding times from seconds to minutes [23]. Edges come from short-term trends, event-driven moves (like news or economic data bursts), momentum/patterns, and liquidity provision during known volatility windows. MFT blends quant models with moderate infrastructure – fast data but not nanosecond speed. *Risks:* higher slippage than HFT, competition from true HFT on very short timescales, and from slower traders on longer trends. *FINBOT:* The core mode, targeting microsecond-to-second data for signals but executing on human-scale latency.

- **Low-Frequency Quant (LFT)**: Trades held days to months. Edges from fundamental analysis, long-term statistical mispricings, momentum, seasonality. *Risks:* overnight risk (overnight gaps), macro shocks, large drawdowns. *FINBOT:* Can include some market-neutral quant factors or swing trades for diversification, but main focus is shorter.

- **Options Scalping/Market Making**: Quoting options and hedging the Greeks rapidly. Edges come from earning the implied spread between bid/ask, exploiting model errors, and gamma scalping. *Risks:* volatility spikes (gapping losses), hedging slippage, model breakdown. *FINBOT:* If planned for options, requires precise volatility models and risk mgmt, but likely out of core scope if focusing on equities/futures.

- **Momentum (Trend-Following)**: Buy when prices rise (or sell when falling). Edges: continuation of price moves, often driven by institutional flows or breakouts. Use indicators like moving averages or momentum oscillators. *Risks:* false breakouts, reversals, whipsaws in choppy markets. *FINBOT:* Implement momentum signals (e.g., price crossing VWAP or moving average) for short bursts.

- **Arbitrage**: Exploit price differences for the same or related assets. Examples: Statistical arbitrage (pairs trading), index arb (futures vs basket), exchange arbitrage (same stock dual-listed). *Risks:* execution risk, funding costs, model decay. *FINBOT:* Latency-sensitive; without co-lo, focus on pairs or futures vs spot where speed is less extreme. Use Kalman filters or Z-scores for pair signals.

- **Mean Reversion (Statistical Arbitrage)**: Assume prices revert to some mean (e.g., VWAP, SMA). Go long the underperformer and short the outperformer in pairs. *Risks:* trends that persist (anti-momentum), and regime shifts. *FINBOT:* Many micro mean-reversion strategies (like legging in-out of VWAP) can fit a mid-frequency horizon.

- **Order-Flow Trading**: Use real-time LOB and trade flow to predict short moves. For instance, if "iceberg" or hidden orders appear, one tries to infer institutional flows. Edges: microstructural predictions of price ticks. *Risks:* noisy signal, adversarial order detection. *FINBOT:* Continually monitor LOB imbalances (as in strategy compendium) to detect big entrants.

- **Statistical/Quant Models**: Machine learning or regression models predicting short-term returns or regime states. Edges: discovering subtle patterns, combining many factors. *Risks:* overfitting, model decay, look-ahead bias. *FINBOT:* ML components used for volatility/regime forecasting rather than pure direction prediction.

For each strategy, FINBOT will define precise entry/exit rules and risk filters. For example, a momentum breakout strategy might *enter long* when price closes above the rolling maximum and volume surges, with a stop at a volatility-adjusted ATR distance. Conversely, a mean-reversion VWAP strategy might *enter short* when price rises a threshold above VWAP, expecting a reversion to VWAP. Each strategy must include checks: e.g., avoid opening new trades near market close, or skip if overall volatility is low (noisy). The complete strategy library (see Part 5) will detail these in quantifiable terms.

Part 2 — FINBOT Concept & Vision

FINBOT is envisioned as a **mid-frequency algorithmic trading platform** integrating multiple quantitative strategies and machine learning insights. It will not compete in the lowest-latency HFT tier (no co-location or microwave link), but will draw inspiration from HFT microstructure strategies, applying them on slightly slower timescales (seconds). The goal is to accumulate many *micro-edges* — tiny profit opportunities (fractions of a tick or basis points) — across strategies and instruments [24]. Over time, these micro-edges compound into significant returns, provided risk is well-managed.

Core Principles: - **Micro-Edge Accumulation**: Like HFT firms, FINBOT seeks a lot of small trades rather than a few large ones. Each trade's edge may be minute, but the system aims for high hit rates and tight controls to net positive. - **Robust Data & Tech Stack**: High-quality data feeds (tick-level) and fast processing pipelines ensure signals are timely. We use **Python-based technologies** (as per user preference) – e.g., *asyncio* or *multiprocessing* for I/O, Pandas/NumPy/DuckDB for data handling, and Cython or Numba where critical for speed. - **Hybrid Strategies**: No one strategy always wins. By combining **multiple strategies** (market-making, trend-following, arbitrage, anomaly detection), FINBOT benefits from diversification. Different strategies excel in different regimes – e.g. trend followers in trending markets, mean-reverters in range markets. Research suggests that strategy portfolios have lower volatility and higher risk-adjusted returns than any single strategy [25]. We'll dynamically weight or switch strategies as conditions change. - **Performance Goals**: FINBOT will set specific KPIs: target Sharpe > X (e.g. 2.0), max drawdown < Y%, hit-rate (win-rate) ~Z%, etc. It will be capital-efficient with tightly monitored leverage. Targets will be realistic for mid-frequency strategies (far lower frequency of trades than HFT, but still aiming for multiple trades per day per strategy). Profit goals might be, e.g., 20-30% annual return with controlled risk, though this is aspirational. - **Risk and Resilience**: Protecting capital is paramount. FINBOT embraces strict stop-loss rules (hard stops, daily loss limits) and kill-switches (global shutdown if markets behave erratically). All strategies must have clearly defined risk parameters (see Part 7). Drawdowns are expected, but mitigated by not over-leveraging any one signal and by hedging when possible.

Why Hybrid Outperforms Single-Strategy: Empirically, no single alpha lasts forever. Markets adapt. If FINBOT ran just one strategy (e.g. only mean-reversion), it would bloom in certain conditions then blow up in others (trending breakouts). A combination approach smooths the equity curve: when mean-reverters struggle in a fast trend, a momentum strategy can pick up some slack. Backtests and real-world practice show that multi-strategy portfolios enjoy lower volatility (due to low correlation between strategies) and steadier returns [25]. Thus, FINBOT's philosophy is *"don't put all eggs in one algorithmic basket."* We will maintain a mix: market-neutral (arbitrage, LOB microstructure), directional (momentum, breakout), statistical (pairs), and volatility-based (ATR-breakout). This balanced approach aims to make FINBOT robust across changing markets.

In summary, FINBOT is envisioned as a cloud-based (or VPS) quantitative trading system that leverages Python and open data technologies, applies a slew of complementary strategies, and emphasizes data-driven performance with professional risk controls. Its mission is to turn the trader into a *"machine that sees the market microstructure clearly"*, accumulating pips and micro-dollars one trade at a time, while carefully managing the inevitable risks.

Part 3 — FINBOT System Architecture (Detailed)

Below is an overview of the FINBOT architecture, broken into functional modules. A high-level schematic (see embedded illustration) shows data flowing from market feeds through ingestion, strategy logic, and execution, with supporting systems for risk, ML, and monitoring.

【96†】 *Figure: An illustrative high-level architecture of FINBOT, showing data flow between Market Data, Strategy Engine, Execution Engine, Risk/State, and supporting systems (data store, ML module, monitoring).*

Market Data Engine

Purpose: Ingest live market data (ticks, order book updates, 1s/5s candles) from chosen exchanges or market data vendors. Provide a normalized, unified data feed to downstream components.

Inputs: Exchange feeds (e.g. NSE/BSE data via FAST/FIX feeds, or WebSocket price feeds), including: - **Tick data:** every trade (price, volume, time). - **Book updates:** best bid/ask and, if possible, multiple LOB levels. - **Bar data:** aggregated 1-minute and 5-minute OHLC (could be generated in-engine).

Outputs: Stream of normalized tick and bar data. A consolidated order book snapshot updated per tick.

Internal Logic: 1. **Connection Handling:** Maintain live connections to exchange or broker APIs. Use non-blocking I/O (asyncio or multithreading). Implement reconnect logic on dropouts. 2. **Parsing & Normalization:** Convert vendor-specific fields to a common schema. E.g., standardize time (UTC), symbols, and fields (bid1, ask1, bidSize1, askSize1, etc.). 3. **Aggregation:** For systems that need 1s/5s bars, aggregate ticks (or use exchange-provided candles). Store these in-memory or DB. 4. **Book Management:** Apply incremental updates to internal LOB state. Keep best bid/ask and a few depth levels for quick reference by strategies. 5. **Distribution:** Publish data to in-memory queues or via pub/sub (e.g. ZeroMQ, Redis streams) so that Strategy Engine can consume with minimal latency.

Data Structures: Efficient ring buffers or dequeues for recent ticks. A tree or sorted list for LOB levels. Using native arrays or memoryviews for speed. Possibly store a small time-series window per symbol (e.g. last N ticks) in a circular buffer for fast access.

Latency Considerations: Minimize processing per tick. Use compiled JSON parsing (or binary protocols) if possible. Batch outputs if needed. Critical path: feed handler to strategy input should be microseconds to low milliseconds. Avoid Python-GIL delays by isolating intensive parsing in separate processes or using libraries like *ciso8601* for fast timestamp parsing.

Security/Scalability: Encrypt feed credentials, use authenticated APIs. For scalability, run multiple feed handlers (sharded by symbols) and use multicore. Implement a backup feed or replay from cache to guard against downtime.

Data Normalization

Purpose: Ensure all historical and live data is clean, consistent, and adjusted for corporate actions.

Inputs: Raw data from Market Data Engine; historical datasets (parquet/csv).

Outputs: - Price series normalized to account for splits/dividends. - Continuous data streams for indicators. - Data flagged for anomalies.

Internal Logic: - **Corporate Actions Adjustment:** Apply historical adjustments so that prices are on a continuous scale. For equities, adjust pre-split prices down by split factor and adjust volumes up [26] [27]. For example, after a 2-for-1 split, historical prices double and volumes halve to maintain consistency. Record split and dividend events to apply on-the-fly to candles or indicator calculations.
- **Rolling Windows:** Maintain sliding windows (arrays/deques) of recent data points (e.g. last 100 ticks, last 50 bars). Useful for fast indicator computation.
- **Resampling:** From tick or trade data, compute time-bars (1s, 5s, 1m) on the fly using the formula in [87]: VWAP, ATR, etc.
- **Cleaning Rules:** Filter out outlier ticks (e.g. test or zero-priority trades). If trade price deviates improbably from book, check for exchange glitches. Ensure monotonic timestamps; drop or interpolate missing data. Handle non-trading periods (illiquid symbols).
- **Live vs Historical Feeds:** Implement two interfaces: *LiveDataFeed* (streaming ticks/candles to strategies in real-time) and *HistoricalDataFeed* (serving batched data for backtests). They share normalization logic but differ in time control (real-time vs simulation time).

Data Structures: For historical storage, Parquet files partitioned by date/symbol and queried via DuckDB or Pandas [28] [29]. In-memory, use NumPy arrays or Polars DataFrames for columnar operations. Store tick LOB snapshots as compressed formats (parquet, or specialized like ProtocolBuffers for speed) for possible replay.

Latency: Historical loading can be slower (batch mode), but live normalization must keep up with incoming ticks. Use compiled C extensions or optimized libraries (e.g. Polars) for on-the-fly aggregation. Avoid Python loops on ticks whenever possible.

Strategy Engine

Purpose: Host the trading strategies logic. Ingest normalized data feeds and compute signals (buy/sell triggers) based on quantitative rules.

Inputs: Streams from Data Normalization (ticks/bars per symbol). Model outputs from ML Engine (e.g. volatility regime). Current positions/PnL from Position State Machine.

Outputs: Order intents: "Go long X shares at price P, or short Y shares…".

Internal Logic: - **Strategy Scheduler:** Manage multiple strategies in parallel. For each strategy, run its logic on incoming data ticks or bars. Could be synchronous or event-driven (on new tick or at bar close). - **Signal Generation:** Each strategy implements mathematical rules. For example: *Order Book Imbalance Strategy* computes imbalance I=(Σ bids-Σ asks)/(Σ all) [20] ; if I exceeds a threshold, place orders expecting immediate price shift. *VWAP Microtrend Strategy* checks price vs intraday VWAP [30] , etc. Parameters (window lengths, thresholds) are configurable. Indicators (e.g. ATR) computed in normalization are consumed here. - **Filters:** Before signaling trade entry, check filters: is volatility above minimum? Is market direction confirmed by auxiliary indicator? Are positions already at limit? For example, skip momentum entry if overall market (e.g. NIFTY index) is flat [31] . - **Position Sizing:** Determine quantity to trade. Could be fixed, or based on risk (e.g. 1% volatility risk via ATR stops [32]). Consult Risk Engine for max exposure. - **Order Generation:** Create an order object with type (market/limit/IOC), price (if limit), quantity, time-in-force. The **Execution Engine** will handle actual submission.

Data Structures: Each strategy maintains its own state (e.g. last action time, indicator values). A common messaging bus or task queue can pipeline signals to execution. Use vectorized math or compiled code for heavy computations. Complex strategies (e.g. machine-learning based) may keep feature buffers.

Latency: Strategy logic should be lean. Ideally, few milliseconds from data arrival to signal. In Python, use Cython/Numba or C++ modules for hot paths, or run strategies asynchronously. Prioritize simpler strategies (limit orders) for speed; complex ML in lower-frequency lanes.

Execution Engine

Purpose: Manage the sending of orders to broker/exchange, handle confirmations, cancellations, and maintain internal order state.

Inputs: Order commands from Strategy Engine. Market tick data (for price reference). Current account positions.

Outputs: API calls to broker/exchange (order submissions). Updates on order status (ACKs, fills).

Internal Logic: - **Order Routing:** Given an order (e.g. Buy 100 shares), decide the venue (e.g. direct to exchange via broker API). For Indian markets, likely FIX connection to a broker or direct API (if exchange permits DMA). Implement **smart routing** if necessary (choose NSE vs BSE based on liquidity). - **Order Types:** Support Limit, Market, Stop-Loss, Stop-Limit (SL-M), Immediate-or-Cancel (IOC), Fill-Or-Kill (FOK)

[33] [34]. For example, to enter a breakout, one might place a market order; for a cautious entry, a limit just inside bid/ask. - **State Machine:** Track each order through states: *Pending* (sent), *Partially Filled*, *Filled*, *Cancelled*, *Rejected*. Update the Position State Machine on fills. - **Retry & Failures:** If a limit order isn't filled in desired timeframe, decide whether to cancel or repost at updated price. On rejection (e.g. order size too large, price outside limit), log and handle (maybe split the order or reduce size). - **Safety Checks:** Before sending, re-validate (e.g., do not send orders outside trading hours, or with quantity exceeding balance). Enforce **risk rules** from Risk Engine (e.g. global daily loss limit). - **Slippage Measurement:** For market orders, measure fill price vs expected mid to log slippage. Use this data offline to calibrate execution assumptions. - **Rate Limits:** Some brokers impose order rate limits. Throttle order submissions accordingly (e.g. no more than X messages per second). Queue orders if necessary.

Data Structures: Order book for our own orders (ID → details). Use asynchronous callbacks or polling on execution API (e.g. REST or FIX).

Latency: Aim to minimize round-trip. Using a fast broker API (HTTP/WebSocket or FIX). If available, route orders via co-lo colocated brokers.

Position State Machine

Purpose: Track current holdings and P&L. Provide unified interface to order engines and risk checks.

Inputs: - Fills from Execution Engine (price, quantity, symbol). - Order cancellations. - External positions (if any manual trades).

Outputs: - Current position (quantity, average cost) per instrument. - Realized/unrealized P&L. - Exposure stats.

Internal Logic: On each fill, update position and cash. Calculate mark-to-market PnL using last prices. Maintain *open orders list* so you know intended positions vs filled.

Data is used by strategies (to avoid conflicting trades) and Risk Engine (to enforce exposure limits).

Data Structures: A dictionary keyed by symbol: `{quantity, avg_cost, pnl, ...}`. Also track short vs long separately.

Risk Engine (Hard + Soft)

Purpose: Enforce risk limits in real-time and during strategy generation.

Inputs: Real-time PnL, positions, market data (for volatility), time of day.

Outputs: Signals or actions: e.g. "halt trading", "do not open new trades", "apply stop-loss", or direct order cancels.

Internal Logic:

- **Hard Stops:** Absolute cut-offs (e.g. a 10% daily drawdown triggers full system halt). Monitored continuously; triggers a *kill-switch*. Example pseudocode:

```
if (dailyLoss >= hardLimit) or (totalDrawdown >= maxDrawdown):
    activateKillSwitch()
```

- **Soft Stops:** Warnings or strategy throttles (e.g. reduce position sizes if losses > 5%). Might stop new entries but allow exiting existing positions.
- **Max Daily Loss/Runup:** If cumulative losses exceed threshold, pause trading until reset. Likewise, if extremely profitable (large runup), consider booking profits.
- **Exposure Limits:** Cap position size per symbol/sector and overall leverage. E.g. no more than 5% of equity in one stock, or 10:1 net leverage.
- **Time-Based Rules:** E.g. no new trades 5 minutes before market close; all positions must be flat by a set time (if mandated). Also, enforce start/stop trading hours.
- **Slippage & Volatility Filters:** If market volatility (e.g. VIX or ATR) exceeds a threshold, either widen stop-losses or halt scalping strategies. If price moves too quickly, cancel pending orders to avoid chasing a gap.
- **Circuit Breakers:** If any instrument hits circuit breakers (exchange-imposed price bands) or if index declines > X%, pause trading.
- **Hedging:** If holding large directional risk (e.g. portfolio net long), optionally hedge using futures or indices to reduce market exposure.

Data Structures: Risk config table (limits per strategy, per symbol). Counters and timers.

Config Example: (YAML format)

```yaml
risk:
  hard_stop:
    daily_loss: 50000  # INR
    drawdown_pct: 10%
  soft_stop:
    daily_loss: 20000
  exposure:
    max_position_percent: 2%  # of capital
    max_leverage: 5
  circuit_breaker:
    index_drop_pct: 10%
    volatility_atr: 5  # ATR threshold
```

Enforcement Pseudocode: (called before sending orders)

```python
def check_risk_before_order(symbol, qty, price):
    if positions[symbol] + qty > exposure_limit[symbol]:
        return False
    if current_pnl < -hard_stop_daily:
        return False  # kill switch active
    if time_close_to_end or circuit_breaker_triggered:
        return False
    return True
```

All such rules run in the background. The **kill-switch** is the ultimate safety: a manual or automatic trigger that immediately cancels all orders and shuts down new signals.

Machine Learning Engine

Purpose: Provide advanced predictive analytics (without predicting price direction directly) to inform strategies. ML tasks include regime detection, volatility forecasting, anomaly detection, and feature extraction.

Inputs: Historical and real-time features (see Part 6). Data from Data Engine, plus strategy performance logs.

Outputs: - Regime or volatility labels/predictions to Strategy Engine. - Feature transformation pipelines.

Internal Logic: - **Offline Training Pipeline:** Backtester uses historical data to engineer features (technical indicators, order flow stats, cross-asset signals) and label regimes (e.g., "high volatility" vs "low", "bullish vs bearish micro-regime"). Models (XGBoost, LSTM, etc.) are trained and validated using walk-forward (rolling) methodology. No peeking into future allowed [27]. Hyperparameters tuned on separate in-sample data.
- **Feature Engineering:** Derived features include normalized order book imbalance [20], volume delta, rolling ATR, VWAP deviation, momentum scores, time-of-day indicators, etc. Use time-series rolling windows (Pandas window, NumPy arrays) to compute on the fly.
- **Model Types:** Use **tree-based models (e.g. XGBoost)** for tabular regime classification; **LSTM/ Transformers** for sequence patterns if needed (with caution about overfitting). For example, an XGBoost could classify "expected volatility increase next 5min" based on current features [35]. Unsupervised (like K-means) can cluster market states to label regimes [36].
- **Online Inference:** Trained models are deployed to predict in real-time (or near-real-time). The ML Engine subscribes to live feature feed, outputs predictions (e.g. a "volatility score" or "regime class") on a schedule (e.g. every minute). These labels modulate strategy parameters (e.g. widen stops in volatile regime).

Latency: Inference should be fast (<< 1s). Models can run on GPU or CPU; streaming frameworks (TensorFlow Serving, ONNX Runtime) could be used. Ensure no lookahead: use only past data for predictions.

Monitoring/Drift: Continuously check model performance. If a model's accuracy degrades (e.g. sudden distribution shift), retrain or disable it. Keep a labeled evaluation set for sanity checks.

Backtester (Order-Level Simulation)

Purpose: Simulate execution of strategies over historical microstructure to validate performance before live deployment.

Components: - **Data Replay:** Feed historical tick/order-book data into the system at accelerated pace.
- **Latency Simulation:** Introduce artificial delays to mimic real-world order latency (network + processing). For instance, if live latency ~100ms, then a strategy signal generated at time *t* might only actually send an order at *t+100ms*.
- **Slippage & Partial Fills:** Model fills realistically. If strategy places a limit order, simulate whether it would have been filled based on the historical LOB. For market orders, simulate eating through book levels. E.g. if buying 1000 shares at market at time *t*, it might fill 800 at price1 and 200 at price2.

Incorporate exchange rules (ICEberg orders, minimum sizes).
- **Multiple Strategies Interaction:** Allow multiple strategy threads to run on the same historical data, updating the virtual portfolio. This captures portfolio-level PnL and margin interactions.
- **Metrics & Portfolio Analytics:** Compute daily PnL, drawdowns, Sharpe/Sortino ratios, win-rate, average profit/loss, etc. Perform Monte Carlo by bootstrapping trade sequences to estimate statistical confidence. Use walk-forward optimization: retrain any parameters on rolling windows and test out-of-sample.

Architecture: Often the backtester mirrors the live system modules: the same Strategy Engine code consumes a *HistoricalDataFeed* and interacts with a simulated Execution Engine that writes to a virtual Order Book simulator.

Example Pseudocode:

```
for timestamp, tick in historical_ticks:
    data_engine.process(tick)
    strategy_engine.run_strategies(timestamp)
    for order in strategy_engine.new_orders:
        exec_simulator.submit(order, timestamp)
    exec_simulator.process_book(tick)  # match vs historical LOB
    position_state.update(exec_simulator.fills)
    risk_engine.check(position_state, timestamp)
```

Monitoring Layer

Purpose: Real-time observability of system health, strategy performance, and market conditions.

Components: - **Metrics Exporter:** Instrument the code to expose metrics (latency timings, PnL, position sizes, number of open orders, fills per minute, errors) via a time-series system (Prometheus).
- **Dashboards (e.g. Grafana):** Visualize key stats: portfolio equity curve, per-strategy PnL, exposure heatmap, latency histograms, and real-time PnL heat. Provide alerting (e.g. email/SMS) on critical events (kill switch triggered, server down). Example: use a Prometheus/Grafana stack as in [80] to collect and display metrics from FINBOT's processes [37].
- **Logging & Telemetry:** All events (orders, fills, errors) are logged with timestamps. Use structured logging (JSON) so logs can be parsed (e.g. ELK stack). Telemetry includes profiling (what part of code takes time). Logs are time-synced for post-mortem analysis.

Logging + Telemetry

Purpose: Record everything for debugging, compliance, and analysis.

Key Logs: - **Trade Logs:** Every order and fill (time, symbol, side, qty, price). The OMS log style.
- **Signal Logs:** Strategy events (e.g. "Strategy X: enter long 100 at 1000.5"), along with rationale.
- **Error/Exception Logs:** Any failed API call, disconnect, or logic error. These trigger alerts.
- **System Logs:** Process start/stop, config changes.

Telemetric Traces: Optionally use a tracing system (OpenTelemetry) to trace execution paths and latency (e.g. measure time from strategy signal to execution publish).

Logs must be time-stamped (UTC) and ideally replicated off the machine (e.g. remote syslog or cloud logging) for durability.

Deployment Layer (Cloud + VPS)

Purpose: Infrastructure for running FINBOT reliably.

- **Servers/VPS:** FINBOT can run on cloud VMs (AWS, GCP) or a dedicated colocation if needed. Each module (Data Engine, Strategy Engine, etc.) can run as separate processes/microservices (Docker containers) for isolation and scalability. Use Linux (Ubuntu).
- **Orchestration:** A container system (Docker Compose or Kubernetes) can manage service start-up order and restarts.
- **Secrets Management:** Store API keys, database passwords in a secure vault (AWS Secrets Manager or HashiCorp Vault), not in code.
- **CI/CD:** Use GitHub Actions or similar to test code (unit tests, linting) and deploy to servers. Automated tests before release (including a small backtest).
- **Deployment:** Container images built with version tags. Rolling update of strategies by loading new Docker image.
- **Redundancy:** Run critical services (like data ingestion) in at least 2 instances (possibly in different AZs) for failover. If one feed handler dies, backup continues.
- **Monitoring:** Use cloud watch or Prometheus node exporters for system health (CPU, RAM, disk, network). Keep automated alerts for high CPU (could indicate runaway process) or disk space low (for logs).
- **Disaster Recovery:** Regular backups of historical data and code repository. Plan to redeploy on new VM within minutes. All persistent state (positions, config) replicated across machines or cloud storage.

Fail-Safes and Watchdogs

Purpose: Automatically catch and handle failures.

- **Process Watchdogs:** Supervisord or systemd to auto-restart any crashed module (e.g. if Strategy Engine crashes, restart it).
- **Health Checks:** Periodic self-check: e.g. a ping thread ensures every subsystem reports alive (heartbeat). If not, alert.
- **Kill Switch:** A hardware or manual software “panic button” that instantly halts execution across system (can be a simple setting in the Risk Engine to reject all orders).
- **Timeouts:** If any external API (market data or execution) hangs, drop and reconnect after timeout.
- **Redundancy:** Duplicate critical lines (two data feeds, two brokers) to avoid single-point-of-failure.

In summary, each module of FINBOT is carefully designed with clear input/output interfaces, robust logic, and performance considerations. The architecture is layered: data feeds → normalization → strategies → execution, with shared support from risk, ML, and monitoring. This separation of concerns aids both development and troubleshooting.

Part 4 — Data Framework

Tick Data vs Candle Data

- **Tick (Trade) Data:** The finest granularity – every transaction or quote update. Includes price, volume, timestamp (often to microseconds). Best for microstructure strategies and precise backtesting, but large in size.
- **Candle (OHLC) Data:** Aggregated time bars (e.g. 1s, 1m). Shows open, high, low, close, and volume for interval. Less granular, but easier for broad patterns.

FINBOT uses both: strategies like order-flow/imbalance rely on ticks or top-of-book updates; others (momentum on intraday trend) may use 1m bars. The Data Engine converts tick streams into fast-updating bars.

Rolling Windows

Many algorithms need features computed over sliding windows. Implement **deque** structures or circular buffers of fixed length (e.g. last 100 ticks or last 60 bars). As new data comes in, old data is popped. For example:

```
from collections import deque
ticks_window = deque(maxlen=1000)
on_new_tick(tick):
    ticks_window.append(tick)
    # compute features on ticks_window[-n:]
```

For time bars, similarly maintain last N bars to compute moving averages or Bollinger Bands.

LiveDataFeed vs HistoricalDataFeed

- **LiveDataFeed**: Interfaces with Market Data Engine. Provides synchronous or callback API for strategies. Abstracts away historical indexing (no random access). Methods like `get_latest_price(symbol)` or subscription callbacks.
- **HistoricalDataFeed**: Reads from stored data (Parquet, CSV) at simulation clock. Offers random access or replay mode. Used in backtests. It should honor timestamps exactly for chronological accuracy.

Both implement a common normalization pipeline, so strategies code can use similar interfaces (IReplace "live" with "history" in tests).

Storage: Parquet, DuckDB, PostgreSQL

- **Parquet Files:** Ideal for columnar storage of historical data (tick or bar), compressed, fast reading of specific columns or date ranges. Partition by symbol and date for quick reads.
- **DuckDB:** An in-process SQL OLAP engine (SQLite-like but with complex queries). Excellent for analytics on Parquet without importing. E.g., run a query to get VWAP over a date range [38] .
- **PostgreSQL:** Possibly used for position logs or small reference data (e.g. instrument list, corporate action calendars). But tick data in Postgres is impractical. More common to use lightweight timeseries DB like InfluxDB for metrics rather than market data.

In practice, FINBOT will archive raw data as Parquets on cloud storage (S3) and use DuckDB or Polars for research queries [39]. The live system may store a rolling cache of recent tick data in memory or in a Redis (for cross-language access).

Order Book Snapshots

For detailed microstructure, FINBOT can periodically snapshot the full LOB (top N levels) for each symbol. Snapshot interval might be every few seconds, or on significant events. Stored as binary (e.g. FlatBuffers or compressed JSON) for replay. Useful for training ML or debugging.

Tick-to-Candle Aggregation

Tick data is aggregated into time bars. Pseudocode:

```
current_bar = init_bar()
for tick in ticks:
    if tick.time < current_bar.end_time:
        current_bar.update(tick.price, tick.volume)
    else:
        emit_bar(current_bar)
        current_bar = new_bar(start=tick.time)
        current_bar.update(tick.price, tick.volume)
```

End-of-bar logic can also compute VWAP (sum(price*vol)/sum(vol)) and close = last tick price.

Data Cleaning Rules

- **Remove Duplicates:** Sometimes feeds resend old ticks. Filter by (time, price, volume) uniqueness.
- **Outlier Removal:** If price jumps without trades (fat-finger errors), discard or flag. For example, discard a tick if it deviates >10σ from recent mean.
- **Nulls/Gaps:** If feed disconnects, mark missing period and not interpolate for live strategies. For backtest, can impute by carrying last price or skipping period.

Corporate Actions (Splits, Dividends, Symbols)

- **Splits/Bonus:** Adjust past prices and volumes by split factor. Keep a table of split events (date, ratio). Upon loading historical data, apply backward adjustment. Real-time feeds post-split come as raw; use adjusted series for indicators and PnL. For example, after a 2:1 split, historical $200 becomes $100, volume doubles [26] [27].
- **Dividends:** Optionally adjust price series (total return style) or simply ignore for short-term strategies. If payouts are significant (like 10%), avoid trading that day or account for gap.
- **Symbol Changes:** If a company changes ticker, merge histories or map old symbol to new. A lookup table can handle renames.

Anomaly Detection

Implement rules to detect bad data: - **Sudden Depth Loss:** If LOB depth drops to zero on one side, pause trading until book refills. - **Volume Spikes:** Unusual surge (10× average volume) may indicate news; use this as an input to MFT strategies. - **Crossed Book:** If best bid > best ask (shouldn't happen), pause consumption.

Data Layer Pseudocode

```
class DataFeed:
    def __init__(self):
        self.ticks_window = deque(maxlen=10000)
        self.bars_1s = deque()
        self.bars_5s = deque()
        self.last_bar_1s = None

    def on_tick(self, tick):
        # Normalize time, symbol, price/volume format
        tick = normalize_tick(tick)
        self.ticks_window.append(tick)
        self._update_bars(tick)

    def _update_bars(self, tick):
        # 1-second bar
        ts = tick.timestamp // 1000 * 1000  # floor to second
        if not self.last_bar_1s or ts > self.last_bar_1s.start:
            if self.last_bar_1s:
                self.bars_1s.append(self.last_bar_1s)
            self.last_bar_1s = Bar(start=ts, open=tick.price,
high=tick.price,
                                    low=tick.price, close=tick.price,
volume=tick.volume)
        else:
            b = self.last_bar_1s
            b.high = max(b.high, tick.price)
            b.low = min(b.low, tick.price)
            b.close = tick.price
            b.volume += tick.volume
        # Similarly for 5s bars...
```

This pseudocode shows a basic streaming tick-to-bar aggregation and rolling window storage.

Part 5 — Strategy Compendium (10–20 Strategies)

Below are detailed specifications for a suite of strategies. Each is described quantitatively, with conditions and rationale.

1. **Order Book Imbalance Scalping**:
2. *Intuition:* If buyers greatly outnumber sellers in LOB, price is likely to tick up. Vice versa if sellers dominate [20].
3. *Logic:* Compute imbalance I = (ΣBidVol – ΣAskVol) / (ΣBidVol + ΣAskVol) over top N levels (e.g. 5 levels).

4. *Entry:* If I > +Δ or I < –Δ (e.g. Δ=0.3) and no current position in symbol, place a small market order in the direction of imbalance. For I > +0.3: buy market order; for I < –0.3: sell short.
5. *Exit:* Take profit after +1 tick of adverse move or fixed targets, or if imbalance reverses sign. Use trailing stop of 1-2 ticks.
6. *Filters:* Only when spread < threshold and volume > threshold (avoid illiquid). Skip if volatility is extremely high (to avoid noisy imbalances).
7. *Risks:* Rapid book changes can lead to false triggers; strategy may suffer when front-run by faster traders.

8. *Works when:* Slight one-sided book imbalances push price. Fails when sharp news causes book to sweep.

9. **Cumulative Volume Delta (CVD) Reversion**:

10. *Intuition:* Track net aggressive buys vs sells; extreme values often mean exhaustion.
11. *Logic:* Maintain CVD = Σ(buyVol – sellVol) over rolling window (e.g. last 1000 ticks). Normalize by volume.
12. *Entry:* If price has moved one direction, but CVD is extreme opposite (e.g. price down while CVD is strongly positive), bet on reversion: enter against price.
13. *Exit:* Exit when mean CVD is neutralized or after fixed profit (e.g. +0.05% gain). Stop-loss at e.g. 0.1% adverse move.
14. *Filters:* Only trade when divergence is statistically significant (outside 2σ) and not near day edges.
15. *Fails when:* Persistent buying/selling (trend stronger than imprint).

16. *Example:* Stock XYZ down 0.5% intraday, but CVD indicates net buying pressure – possibly a short-squeeze or algorithmic support; enter long.

17. **Liquidity Vacuum Detection**:

18. *Intuition:* When LOB at best levels is very thin (small sizes), the price may "vacuum" to next available liquidity, causing jumps.
19. *Logic:* If both best bid and ask sizes fall below a threshold (e.g. 100 shares) and spread is normal, an incoming large order could sweep.
20. *Entry:* If a significant marketable order is detected (e.g. a big trade occurs), follow its direction. If no trade, preemptively do nothing.
21. *Alternate:* Place limit orders just beyond current spread, to catch runaway moves.
22. *Filters:* Only consider very liquid instruments where vacuum is meaningful.

23. *Risk:* False breakouts – quoted liquidity may refill quickly.

24. **Stop-Hunt Reversal**:

25. *Intuition:* In illiquid markets, price often aggressively moves to trigger stops (e.g., below round levels), then reverses as it was purely liquidity seeking.
26. *Logic:* Identify cluster of stop orders (e.g. just below moving support). If price quickly breaks support by X% (say 0.5%) and then stalls, suspect stop-hunt.
27. *Entry:* Enter long after the spike (e.g. if price falls 0.5% quickly then recovers above mid of that range). Stop 0.3% below recent low.
28. *Exit:* Profit target at original support (reversion).
29. *Filters:* Only when volume spike accompanies move (shows big activity).

30. *Fails:* If genuine breakouts.

31. **VWAP Microtrend**:

32. *Intuition:* VWAP is the average price weighted by volume. Price trending above intraday VWAP suggests strength.
33. *Logic:* Calculate rolling intraday VWAP every minute [30] .
34. *Entry:* If price crosses above VWAP and average price (e.g. previous 5-min MA) is trending up, go long. Conversely, short when price crosses below and trend down.
35. *Exit:* Exit when price re-crosses VWAP or at small profit. Use stop = multiple of intraday ATR below entry.
36. *Filters:* Only between 9:45am-3:00pm (avoid open/close volatilities). Ensure enough time remaining (e.g. ignore last 10 minutes).
37. *Risk:* Range-bound markets cause whipsaws.

38. *Example:* NIFTY50 crosses above its VWAP at 11:15am and is above its 15-min SMA – buy a scaled order.

39. **ATR Breakout Continuation**:

40. *Intuition:* Large price moves often continue. Use Average True Range to quantify breakouts.
41. *Logic:* Compute ATR(14). If intraday price move exceeds k×ATR (e.g. k=2) within a short window, momentum likely.
42. *Entry:* If |close-now – close-5min| > 2×ATR and volume > norm, enter in direction of move. E.g. if price jumps >2×ATR up, buy.
43. *Exit:* Trail at 1×ATR below peak (or reverse signal).
44. *Filters:* Avoid if already in extreme overbought (e.g. RSI>90).

45. *Fails:* Reversals from exhausted moves.

46. **Mean Reversion after Extension**:

47. *Intuition:* After a strong swing (e.g. 3×ATR from a moving average), price often reverts partway.
48. *Logic:* Maintain an exponential moving average (EMA) of short period (e.g. 20-min EMA). If price diverges >2×ATR above/below this EMA, look to trade back toward it.
49. *Entry:* Price 2×ATR above EMA → sell short expecting reversion. Vice versa buy if 2×ATR below EMA.
50. *Exit:* Target EMA level; stop at further 1×ATR against.
51. *Filters:* Not if trending strongly (checked via ADX or higher ATR).

52. *Example:* Stock ABC has been rising; it is now 2 ATR above its 20min EMA – enter a short; cover when price hits EMA.

53. **Breakout Scalps**:

54. *Intuition:* Many patterns (range breakouts) give quick scalps.
55. *Logic:* Identify recent consolidation (e.g. tight 5-min range). If price breaks above high or below low by a threshold, scalp in that direction.
56. *Entry:* Use market or aggressive limit order when price moves 0.1% beyond the consolidation.
57. *Exit:* Small profit (0.1%-0.2%) or upon close back into range. Tight stop if reversal.
58. *Filters:* Ensure breakout follows volume surge (confirmation).

59. *Risk:* False breakouts (throwbacks). Always use stops (possibly guaranteed).

60. **Spread Widening Arbitrage**:

61. *Intuition:* In normal markets, bid-ask spreads are stable. If one side's spread suddenly widens (due to one large hidden order leaving book), profit from crossing it.
62. *Logic:* Monitor inside spread. If spread jumps (e.g., best ask moves away), place a limit order at old best ask price and opposite side. Essentially buying cheap or selling expensive.
63. *Exit:* When book rebalances (profitable).
64. *Filters:* High liquidity symbols only.

65. *Fail:* If spread stays wide or moves further.

66. **Micro Pullback Continuation**:

    - *Intuition:* In a strong trend, small pullbacks (against trend) often quickly resume.
    - *Logic:* Determine trend via short MA (e.g. 10-bar). If price briefly retraces (e.g. hits 5-bar MA) and volume drops, then resumes, take trend side.
    - *Entry:* Place limit order at retest of trend line support.
    - *Exit:* When counter-trend signal appears or fixed TP.
    - *Example:* Uptrend: price dips 0.1% then rises – buy at bottom of dip.

Each strategy above is codified with clear rules. In practice, FINBOT will implement them as discrete algorithms (ideally in interchangeable modules). *Risk controls* for each include per-trade stop-loss (often ATR-based) and overall strategy position limits. Through backtesting and live observation, parameters (e.g. thresholds Δ, ATR multiples) are tuned. Performance is continuously evaluated: if a strategy underperforms, its signals can be disabled or adjusted.

(For brevity, only 10 strategies are detailed. Additional ones – e.g. pure market-making (posting at inside), volatility arbitrage, seasonal pairs – could be added similarly.)

Part 6 — Machine Learning Layer

Rather than trying to *predict price direction* directly (a notoriously hard problem due to market efficiency), FINBOT uses ML to support trading in other ways. **ML Roles in FINBOT:**
- **Regime Detection:** Classify market state (e.g. low vs high volatility, trending vs mean-reverting, bull vs bear micro-regime). E.g., K-means clustering on features to label regimes [36]. Strategies can switch or reweight based on regime (e.g. disable mean-reversion in trending regime).
- **Volatility Prediction:** Use features (e.g. recent order flow, economic calendar info) with models (XGBoost) to predict short-term realized volatility. If predicted volatility is high, the Risk Engine may widen stops or reduce position sizes.
- **Anomaly Detection:** Unsupervised models (e.g. autoencoders) monitor live data to flag unusual LOB patterns (e.g. possible quote stuffing).
- **Feature Extraction:** Deep networks (CNN/Transformer) might extract latent patterns in the order book depth over the last N updates, to feed into other rules. For instance, a CNN on the LOB can be a strong feature generator for pattern recognition [40] [35].

Feature Engineering: Carefully craft inputs from raw data. Examples:
- *Price-based features:* Recent returns (1s, 1m), moving averages, RSI, VWAP deviation [41].
- *Volume features:* Order flow imbalance, on-balance volume, recent trade size distribution [42].
- *Order book features:* Depth imbalance at multiple levels, spread changes, number of orders.
- *Macro/Cross-assets:* Index moves, FX or commodity prices impacting correlated stocks, sentiment scores (news classification).

All features are computed without future leakage – only data up to current tick.

Labeling Methodology: If doing supervised tasks, labels might be: future realized volatility, or regime classes (encoded as integers). Data is split chronologically: train on past, validate on rolling forward windows. Avoid lookahead by only using information available up to each point.

Model Choices: For tabular features, **XGBoost** and **LightGBM** often excel and train fast. For sequence patterns, **LSTM** or **Temporal CNNs** can be used, but risk overfit (use sparingly). **Transformers** could encode LOB snapshots or trade sequences. In practice, start simple: e.g., Random Forest on volatility targets, then experiment with deep nets if justified.

Offline Training Pipeline: Use a separate compute environment for model building. Data scientists extract features from historical datasets (DuckDB/Polars for heavy SQL), train models, backtest their contribution (does regime filter improve strategy Sharpe?). They perform **walk-forward validation**: repeatedly train on X weeks, test on next Y. This respects time ordering to avoid lookahead.

Online Inference System: The deployed ML models run in production: a service (could be a Python microservice) subscribes to live features and outputs predictions (e.g., a "volatility up" boolean) at regular intervals (say every minute). This output is consumed by strategies (e.g. skip signal if volatility is flagged high) or risk systems.

Avoiding Lookahead Bias: All features use only data up to the current tick. Indicators that rely on future data (e.g. forward returns) are strictly offline-only for training labels, never fed into live models. For walk-forward, use non-overlapping windows or expanding windows properly.

Walk-Forward Validation: Continuously test models forward in time: after each new month of data, retrain using all data up to now, then test on subsequent period before retraining again. This simulates real re-calibration and avoids selecting models that only happened to fit one timeframe.

Deployment & Monitoring: Models in production are versioned. Their performance is monitored (e.g. how often their predictions were correct, or how often strategies proceeded under each predicted regime). If model performance degrades or market changes drastically (e.g. new market microstructure rules), the team retrains or re-validates. There should be a fallback to disable ML components if they cause issues.

Part 7 — Risk Management (Institutional-Grade)

FINBOT's risk framework operates at multiple levels:

- **Hard Stops:** Non-negotiable shutdowns. E.g. *Max Daily Loss*: if the portfolio loses >X (INR/USD) in a day, trigger full stop-out of trades. *Max Drawdown*: if peak-to-valley equity drops >Y%, halt all strategies. These must be enforced in code (Risk Engine).
- **Soft Stops:** Gradual controls. For example, if down 2% in a day, reduce position sizing by half; if 5%, stop new entries (only exits). These are warning phases before hard stop.
- **Exposure Limits:** Predefined maximum notional exposure per instrument or sector. For instance, no more than 5% of NAV in any one stock [43] . Also overall leverage cap. Enforced by refusing signals that would exceed limits.
- **Time-Based Rules:** Do not hold positions overnight (if intraday-only strategy). Liquidate or set guaranteed stops near close. No new trades within 5 min of close to avoid the final volatility burst.

- **Slippage & Liquidity Filters:** Monitor realized slippage; if it spikes (e.g. average slippage > 1× tick), raise risk flag. Check market volume; if volume falls below threshold, narrow order sizes.
- **Volatility Filters:** Calculate ATR or rolling volatility. If ATR > cutoff (market too wild), pause scalping strategies. Conversely, if volatility < very low, skip momentum (flat markets).
- **Circuit Breakers:** If broader market tumbles (e.g. index down 10%) or single stock hits exchange limit-up/down, wind down positions to avoid gap risk.
- **Hedging Framework:** Optionally use index futures or ETFs to hedge systemic exposure. Example: If net portfolio is +10% long beta to NIFTY, partially short NIFTY futures to neutralize market risk (with a correlation model).
- **Kill-Switch:** A master emergency stop. Can be triggered manually (GUI button) or automatically by risk rules. It should instantly cancel all open orders and disable new order sending. Often implemented by having a flag in the Execution Engine (`if kill_switch: refuse all orders`).

Example Soft-Stop Pseudocode:

```
# Called after each fill or periodically
if daily_loss > hard_limit:
    kill_switch = True
elif daily_loss > soft_limit:
    stop_new_entries()
```

Configuration Example (YAML snippet):

```yaml
risk_controls:
  hard_stop:
    max_daily_loss: 100000  # in account currency
    max_equity_drawdown: 15%  # since inception or since rollover
  soft_stop:
    start_daily_loss: 50000
  exposure_limits:
    per_symbol_pct: 3%  # of equity
    total_leverage: 3  # e.g. notional/equity
  time_rules:
    no_trade_after: "15:25"
    no_trade_before: "09:20"
  volatility:
    atr_multiplier_threshold: 3.0
  circuit_breaker:
    index_drop_pct: 10%
```

In practice, these rules ensure that even if an algorithm goes awry, FINBOT stops losing money fast. The goal is to **fail-safe**: never wipe out capital on a single strategy glitch.

Part 8 — Backtesting Framework

A robust backtester is essential before deploying any strategy. Key elements:

- **Microstructure Simulation:** Rather than just candle charts, the backtester must simulate the Limit Order Book and order interactions. If full historical LOB data is available, replay it tick-by-tick. Otherwise, use L1 data and assume a certain order book shape beyond the top. For aggressive orders (market or IOC), simulate walking through multiple levels. For limit orders, assume either full fill when price reaches it (conservative) or probabilistic partial fills (based on tick volume matching).

- **Latency Simulation:** Incorporate delays. For example, if a strategy logic fires at timestamp T (upon tick arrival), add N milliseconds before the order is inserted into the book. The strategy may miss some price moves. This can be randomized or fixed. E.g.:

  ```
  send_time = signal_time + latency_ms
  ```

  Then find the fill that matches send_time.

- **Slippage and Partial Fills:** Model realistic fills. If a limit order is partially filled (because price moved favorably but not enough volume), the remainder can stay pending or cancel at expiration. For market orders, split across LOB levels if needed. Use **limit order book snapshots** or volume-weighted methods to simulate.

- **Portfolio and Multi-Strategy:** The backtest should handle multiple strategies trading the same portfolio. The position and PnL are computed at portfolio level. Weight allocations can be simulated by scaling each strategy's bets so total leverage limit is respected.

- **Performance Metrics:** Compute standard stats:

- **Sharpe Ratio** = (mean return - risk-free) / std deviation.
- **Sortino Ratio** = (mean return - rf) / downside deviation. (Investopedia)
- **Max Drawdown (MDD)** = largest peak-to-trough decline.
- **Average Return per Trade**, **Win/Loss Rate**, **Average Profit vs. Loss**.
- **Trade Utilization:** number of trades per day, per strategy.

- **Execution Metrics:** fill rates, average slippage, turnover.

- **Monte Carlo Simulation:** To assess robustness, randomly perturb key assumptions (e.g. shuffle order of trades, vary slippage, or random start date) and observe distribution of outcomes. This tests sensitivity to luck and sequence of returns.

- **Walk-Forward Optimization:** The idea is to avoid overfitting. Split data into sequential training/test sets. E.g. train parameters on Jan-Jun, test on Jul-Sep; then slide window: train Feb-Jul, test Aug-Oct, etc. Select strategy parameters that perform well across all windows, not just one.

- **Diagram:** A typical backtesting diagram would show: Historical market data → Data normalization (exchanges splits) → Simulated market and order execution logic → Strategy logic triggers → Portfolio aggregator → Performance analytics. (In text, since we can't embed, describe each step).

- **Pseudocode Outline:**

```
for bar in historical_data:
    data_engine.feed(bar)
    strategy_engine.evaluate(bar.timestamp)
    for order in strategy_engine.orders:
        execution_engine.execute(order, bar)
    risk_engine.update(position_state)
compute_statistics()
```

By faithfully modeling how orders would have filled (including delays, slippage, and partial fills), FINBOT's backtester provides realistic simulated performance. It flags false expectations if, for example, a strategy only worked because we assumed instantaneous fills. Results guide adjustments before going live.

Part 9 — Execution Engine (Low Latency)

FINBOT's execution engine acts as the final gatekeeper, turning signals into market orders.

- **Order Types:**
- **Market Order:** Execute immediately at best available price [33]. Use when speed > price precision.
- **Limit Order:** Execute at specified price or better [44]. Ensures price control but may not fill.
- **Stop-Loss (Stop Market):** Becomes market order when a trigger price is hit [45]. Protects against large losses.
- **Stop-Limit (SL-M):** At trigger, becomes limit order at a preset price [46]. Avoids extreme fills but may not trigger.
- **IOC (Immediate-Or-Cancel):** Fill any possible, cancel rest [34]. Good for sniping liquidity.

- **FOK (Fill-Or-Kill):** Must fill entire quantity immediately or cancel [34]. Rarely used, but ensures all-or-nothing.

- **Order Routing Logic:**
  FINBOT will connect to a broker's API (likely via FIX or WebSocket). The system chooses route (e.g. NSE vs BSE) based on best bid-ask available and fees. A **smart router** could split orders or send to multiple venues for best execution. For example, if NSE offers a better ask price than BSE, route buy orders there first.

- **Fill Confirmation State Machine:**
  Each order sent is tracked. States: `NEW -> PENDING -> (PARTIAL -> PARTIAL ...) -> FILLED` or `CANCELLED/REJECTED`. Use callbacks or polling to update. The Position State Machine listens for "FILLED" events to adjust positions.

- **Retry Logic:**
  If an order is rejected (e.g. bad price or quantity), log the reason. For temporary issues (e.g.

exchange busy), could retry once after delay. If a limit order expires (time in force), the strategy may reprice and resend.

- **Rejections:**
  If brokerage rejects (e.g. due to limit exceed, risk check failure), surface that to Risk Engine and halt further orders on that path.

- **Safety Checks:**
  Before sending, confirm order is valid: price within daily high-low bands, quantity <= allowance, not outside trading hours. This prevents obvious errors (like fat-fingers).

- **Smart Order Logic:**
  For large orders, FINBOT may slice into smaller parts (TWAP/VWAP execution algos). E.g., for a 10,000 share target, send 1000 share limit orders at the best bid repeatedly. Measure average fill price vs benchmark (e.g. VWAP).

- **Slippage Measurement:**
  After execution, compare the fill price to mid-market price at order time. Log slippage (in price and INR). Use this to dynamically adjust future order aggressiveness (e.g. if slippage was high, consider limit orders or smaller size).

- **Code-Like Structure Example:**

```python
class ExecutionEngine:
    def send_order(self, order):
        # e.g. via FIX client
        fix_client.send(order.to_fix())
        order.status = 'PENDING'

    def on_fill(self, fill_event):
        order = self.orders[fill_event.order_id]
        order.update(fill_event.qty, fill_event.price)
        if order.remaining == 0:
            order.status = 'FILLED'
        else:
            order.status = 'PARTIAL'
        positions.update(order)  # notify position manager
```

Overall, the Execution Engine aims for reliability and minimum latency. Using asynchronous I/O (e.g. C library or PyC) helps avoid Python GIL delays. Critical decisions (which orders to send and when to resend) are made off the main thread if needed to not block data flow.

Part 10 — Cloud & DevOps

Building FINBOT requires solid infrastructure and DevOps practices:

- **VPS/Servers Setup:** Deploy FINBOT on reliable VPS (e.g. AWS EC2, Google Compute). Use managed databases and services where possible. Each module can run in Docker containers for easy management. For example, one container for data ingestion, one for strategies, etc.
- **Monitoring (Grafana/Prometheus):** As discussed, instrument all processes. Run a Prometheus server (could be on a separate VM) that scrapes metrics endpoints from FINBOT components. Grafana dashboards visualize CPU, memory, tick latency, PnL, fill rates. Set up alerts (e.g. Slack/email) on anomalies (e.g. strategy margin beyond threshold, node down).
- **Logging System:** Logs are centralized. Use JSON logs sent to an aggregator (e.g. ELK or cloud logging like AWS CloudWatch Logs). This allows full-text search of past events. Save logs for compliance (audit of trades). Rotating file logs with Kafka/Logstash forwarder is one approach.
- **Disaster Recovery (DR):** Plan how to recover from failures. Maintain offsite backups of code and data (daily). In AWS, multi-AZ deployments ensure even an AZ outage doesn't kill the system. Test DR procedures regularly (e.g. redeploy on a fresh machine from backup).
- **CI/CD:** Use Git repository with branching. On merges to main, trigger CI pipeline: run unit tests for each module, static analysis (flake8), and perhaps a short integration backtest. On success, auto-deploy updated code to a staging environment. After manual checks, promote to production. Use tools like GitHub Actions or Jenkins. Containerize with version tags for reproducibility.
- **Configuration Management:** Store config (broker endpoints, risk limits, API keys reference) separately from code (e.g. as environment variables or protected config files). Use versioned config.
- **Secrets Management:** Do NOT hard-code API keys. Use a secret manager or encrypted files. For example, AWS Secrets Manager or a Hashicorp Vault.
- **Hot Reload of Strategies:** Support reloading strategy code or parameters without full restart. E.g., design Strategy Engine to watch a directory for updated Python scripts or parameter files and reload classes. Or use an RPC interface to update rules.
- **Remote Debugging:** Run with the capability to attach a debugger or print logs at different verbosity. Possibly use Jupyter notebooks connected to the system for live data inspection. However, production should minimize interactive access to avoid introducing latency; do debugging in staging.
- **Scalability:** FINBOT's architecture should scale horizontally. If more symbols are added, simply start more instances of Market Data and Strategy modules. Use a message broker (e.g. RabbitMQ) for decoupling if needed.
- **CI/Testing:** Automated unit tests for each strategy (feed in synthetic ticks and check expected signals), and integration tests (full loop using mocked exchange). Also have regression tests on historical data to ensure code changes haven't broken past performance.

- **Documentation:** Maintain technical docs (like this blueprint!), and runbooks for operations (how to restart, how to handle common errors, checklists for launch).

Part 11 — Project Roadmap (Phases + Milestones)

Phase 0 → 1 (Prototype to MVP): (0-3 months)
1. **Design & Setup:** Finalize architecture, set up dev environment, select exchanges (NSE, BSE).
2. **Market Data Integration:** Connect to live data feeds. Validate tick and candle ingestion.
3. **Basic Backtester:** Implement HistoricalDataFeed and Execution Simulator. Backtest one strategy (e.g. VWAP microtrend) end-to-end.
4. **Execution Connectivity:** Integrate with a broker API (demo/test) to send and cancel orders.
5. **Risk Framework:** Code core hard stops and logging.
6. **Monitoring:** Deploy Prometheus/Grafana, log pipelines.
7. **Initial Live Testing:** In parallel, run strategies in paper-trade mode (no real orders) to compare with backtest results. Identify any data or logic mismatches.

Week-by-Week Plan (sample):
- *Week 1:* Set up code repo, Docker environment, connect sample data feed.
- *Week 2:* Develop DataFeed classes, test normalization.
- *Week 3:* Build simple Strategy Engine with 1 strategy, test with historical CSV data.
- *Week 4:* Create Execution Engine stub, simulate fills. Build Position tracker.
- *Week 5:* Integrate Risk Engine; run backtest including risk checks.
- *Week 6:* Improve data (splits handling), add logging, metrics.
- *Week 7:* Test on cloud VPS, set up CI pipeline.
- *Week 8:* Add additional strategies, conduct comparative backtests.
- *Week 9:* Refine ML pipeline, start one ML model (regime detection).
- *Week 10:* Start live paper trading on one exchange, monitor via dashboards.
- *Week 11:* Collect paper-trade data, adjust any mismatches.
- *Week 12:* Code freeze for first release, documentation draft.

Testing Plan: Extensive unit tests for data processing and strategy logic. Backtest known edge cases (market open, missing ticks). Simulated “market close” to ensure positions flatten. Load-test data feed at high tick rates.

Paper Trading Plan: Run the full system in *dry-run* mode with live data. Every real signal logs what order *would* be sent. Compare simulated PnL vs theoretical. Validate order execution logic by comparing with small real trades or test orders (if allowed).

Live Rollout Plan: After successful paper run, switch to real small-capital deployment. Start with a single strategy or low volume to ensure correct behavior. Gradually scale up capital and strategy count as confidence grows. Maintain killswitch ready at all times.

Scaling Plan: As FINBOT matures, add more instruments (MCX commodities, cross-listings). Possibly expand to international markets (subject to regulation). Enable multiple server instances for throughput.

Part 12 — Cost Estimates

1. Development Cost:
- *Personnel:* A team of 2-3 quant developers (Python) and 1 systems engineer. In India market, assume

₹25 lakh (~$32k) per developer/year (mid-level), ₹30 lakh for lead. For a 6-month build: ~₹40-60 lakh in salaries. If offshore contractors, similar budgets.
- *Total dev+QA:* ~₹60-70 lakh (~$80k).

2. Cloud/Infrastructure Cost:
- *Servers:* A dev cluster on AWS/GCP: e.g., 3 m5.large (2 CPUs, 8GB) for data/strategy, plus db server. Assume $0.1/hr per instance => $72/mo each, ~$300/mo total => ₹2-3 lakh/year.
- *Data Feeds:* NSE co-location is expensive (tens of lakhs) – but using public or aggregated feeds is cheaper. Indian exchanges offer data packages; assume ₹5-10 lakh/year for live tick data. Historical vendor data (e.g. TickData) could be ₹2-3 lakh.
- *APIs/Broker:* Some brokers charge connectivity fees (one-time ~₹50k) and per-order fees. Budget ₹1 lakh/yr.
- *Software:* Python and OSS mainly. If using premium tools (MATLAB, etc.), cost extra.
- *Misc (AWS RDS, S3 storage):* small for text logs and Parquet. Add ₹1-2 lakh/year.

3. Maintenance Cost:
- Ongoing dev (bug fixes, improvements): ~₹10-15 lakh/yr (1 dev).
- Data and licensing renewals: ₹5 lakh/yr.
- Cloud ops (monitoring, backups): ₹2 lakh/yr.

4. Optimization/Integration:
- If needing to build custom C++ components (for very low latency), that's additional ~₹10-20 lakh.
- Integrating new exchanges or asset classes: developer hours as needed.

5. Total First-Year Estimate: Approximately ₹1.5-2 crore (including salaries and all overheads) for a minimal institutional-grade build. Recurring costs (2nd year) drop to ₹0.8-1 crore (maintenance, modest upgrades, data).

Part 13 — Glossary

(A selection of key terms for quant trading, HFT, microstructure, ML, etc. Citations in definitions where applicable.)

- **Algorithmic Trading:** Automated trading using pre-programmed strategies. See automated trading systems [47] .
- **Alpha:** The excess return of a strategy relative to a benchmark. (A pure driver of profit after adjusting for risk.)
- **Arbitrage:** Simultaneous trades to profit from price discrepancies (e.g. futures vs spot, cross-listed stocks).
- **ATR (Average True Range):** A measure of volatility, computed as the average of true ranges over a period [48] .
- **Backtesting:** Simulating a strategy on historical data to assess performance.
- **Bid/Ask Spread:** Difference between lowest sell (ask) and highest buy (bid) price [33] .
- **Book (Order Book):** See *Limit Order Book*.
- **Bookmap/Level II:** Depth-of-book data beyond best bid/ask.
- **Candlestick Chart:** Price chart using open-high-low-close bars for intervals (e.g. 1min).
- **Candle Data:** Aggregated OHLC data per time interval.
- **Cumulative Volume Delta (CVD):** Sum of (buy volume – sell volume), used to gauge buying pressure.
- **Dark Pool:** Private trading venue where orders are not displayed publicly.

- **Depth:** Aggregate quantity available at each price level in the LOB.
- **DMA (Direct Market Access):** Direct electronic access to exchange order books, bypassing human brokers.
- **EWMA (Exponentially Weighted Moving Average):** A moving average giving more weight to recent data.
- **Fill:** Execution of an order (in part or full).
- **FOK (Fill-or-Kill):** An order that must fully fill immediately or be cancelled [34].
- **Flash Crash:** A very rapid and deep market drop and recovery (e.g., May 6, 2010).
- **Gradient Boosting (XGBoost):** A machine learning method using ensemble of decision trees.
- **High-Frequency Trading (HFT):** Millisecond-scale trading capturing tiny edges [22].
- **ICE (Intercontinental Exchange):** One example of an electronic exchange operator.
- **IOC (Immediate-or-Cancel):** Order executed immediately, any unfilled part is cancelled [49].
- **Institutional Trader:** Large entity trading on behalf of funds (hedge, mutual) as opposed to retail.
- **Latency:** Delay in systems, e.g. time from market event to strategy reaction or order submission.
- **Limit Order:** Order to buy/sell at a specified price or better [44].
- **Limit Order Book (LOB):** The list of outstanding limit buy/sell orders at each price [2].
- **Liquidity:** How easily an asset can be bought/sold; high liquidity means large depth and tight spreads.
- **Liquidity Provider:** Entity that posts significant orders (quotes) to provide market depth (often market makers).
- **Machine Learning (ML):** Algorithms that learn patterns from data. In FINBOT, used for regime detection, etc.
- **Mean Reversion:** Strategy betting that price will revert to a mean (e.g. moving average).
- **Mid-Price:** (Bid+Ask)/2; a synthetic "fair price".
- **Market Maker:** A firm providing bid & ask quotes continuously to earn spread.
- **Market Order:** Order to execute immediately at current best prices [33].
- **Matching Engine:** The software at an exchange that matches buy and sell orders.
- **Momentum:** Price continuation strategy (buy when price is trending up).
- **Order Flow:** Sequence of trades and quotes coming into the market.
- **Order Flow Imbalance:** Difference between buy-initiated and sell-initiated volumes in LOB [20].
- **Order Management System (OMS):** Manages the lifecycle of orders and allocations [5].
- **Portfolio:** Collection of trading positions/strategies managed together.
- **Position State Machine:** Tracks current holdings and PnL.
- **Prometheus/Grafana:** Tools for metrics collection and dashboarding (monitoring).
- **Quantitative Trading (Quant):** Trading using mathematical models and statistics.
- **Regime Detection:** Using ML or statistics to identify market states (volatile vs calm, etc.).
- **Risk Management:** Processes and rules to control financial losses.
- **SAS (System Audit and Surveillance):** Indian regulatory requirement for algo trading logs.
- **SDK (Software Dev Kit):** Tools for interacting with an API.
- **Sharpe Ratio:** (Mean return – risk-free)/std dev of returns. Measures risk-adjusted performance.
- **Slippage:** Difference between expected and actual execution prices.
- **Smart Order Router (SOR):** Routes orders across multiple venues for best execution.
- **Spread:** See *Bid/Ask Spread*.
- **Stop-Loss Order:** Triggers to exit position at market price when a threshold is hit [45].
- **Stop-Limit Order:** On trigger, becomes a limit order [46].
- **Strategies:** Predefined trading algorithms (e.g. momentum, scalping, arbitrage).
- **Swing Trading:** Holding positions from days to weeks, slower than MFT/HFT.
- **Tick:** Minimum price increment or a data update (trade tick).
- **Time-in-Force:** Order expiry instructions (e.g. Day, GTC, IOC, FOK) [50].

- **Transparency:** Exchange requirement to show best prices to participants. (MiFID II enforces higher transparency).
- **Trade (Fill):** Completed part of an order.
- **Transaction Cost:** Commissions, fees, slippage from trading.
- **Trading Bot:** Automated system executing trades (like FINBOT).
- **VWAP (Volume-Weighted Average Price):** Intraday volume-weighted average price [30].
- **Walk-Forward Validation:** Backtesting procedure of rolling in-sample/out-of-sample testing to prevent overfitting.
- **Wash Trade:** Illicit trade with no economic substance (for testing).
- **Window (Rolling):** A fixed-length set of recent data (ticks, minutes) used for indicators.

Part 14 — Final Book-Like Document

FINBOT's documentation above is crafted as an in-depth operational manual and research compendium. It reads like a technical textbook for the new quant hire, covering exchange mechanics, strategy design, system architecture, data engineering, ML integration, risk protocols, and operational deployment. Each section contains detailed explanations, formulas, pseudocode, and industry best practices, with authoritative citations where applicable. The goal is that a reader—from trading desk to developer—gains full expertise in both the financial concepts and the system implementation needed to **design, build, and run** FINBOT end-to-end, with nothing left ambiguous.

---

[1] An introduction to matching engines: A guide by Databento | by Databento | Medium
https://medium.databento.com/an-introduction-to-matching-engines-a-guide-by-databento-d055a125a6f6?gi=f60ee1c7da97

[2] What Is a Limit Order Book? Definition and Data
https://www.investopedia.com/terms/l/limitorderbook.asp

[3] Indian Stock Market: Exchanges and Indexes
https://www.investopedia.com/articles/stocks/09/indian-stock-market.asp

[4] Multi Commodity Exchange - Wikipedia
https://en.wikipedia.org/wiki/Multi_Commodity_Exchange

[5] Order Management Systems (OMS) and their purpose
https://unitedfintech.com/order-management-systems/

[6] Execution Management System vs. Order Management System | SS&C Eze
https://www.ezesoft.com/insights/blog/execution-management-system-vs-order-management-system-selecting-system-your-firms-needs

[7] Understanding Central Counterparty Clearing Houses (CCPs) in Trading
https://www.investopedia.com/terms/c/ccph.asp

[8] [9] Market Making vs. Liquidity Provision in Crypto Explained
https://www.dwf-labs.com/news/market-making-vs-liquidity-provisioning-in-crypto-what-is-the-difference

[10] [11] Institutional vs. Retail Traders: Key Differences Explained
https://www.investopedia.com/articles/active-trading/030515/what-difference-between-institutional-traders-and-retail-traders.asp

[12] [13] [14] Trade Execution Latency: How to Measure and Reduce It
https://finchtrade.com/blog/trade-execution-latency-how-to-measure-and-reduce-it

[15] [16] [17] [18] NSE - National Stock Exchange of India Ltd.
https://www.nseindia.com/static/invest/first-time-investor-sebi-turnover-fees-stt-other-levies

[19] SEC.gov | Section 31 Transaction Fee Rate Advisory for Fiscal Year 2025
https://www.sec.gov/rules-regulations/fee-rate-advisories/2025-2

[20] [21] Market Making with Alpha - Order Book Imbalance — hftbacktest
https://hftbacktest.readthedocs.io/en/latest/tutorials/Market%20Making%20with%20Alpha%20-%20Order%20Book%20Imbalance.html

[22] [24] What are HFTs and How Do They Make Money? - In The Money - Trading Q&A by Zerodha - All your queries on trading and markets answered
https://tradingqna.com/t/what-are-hfts-and-how-do-they-make-money/189210

[23] [31] What is Mid Frequency Trading (MFT)? A Practical Guide
https://bigbrainmoney.com/what-is-mid-frequency-trading/

[25] Multi-Strategy Portfolios: Combining Quantitative Strategies
https://blog.quantinsti.com/multi-strategy-portfolios-combining-quantitative-strategies-effectively/

[26] Corporate Actions - QuantConnect.com
https://www.quantconnect.com/docs/v2/writing-algorithms/securities/asset-classes/us-equity/corporate-actions

[27] The Good Backtest Practices: Adjusted vs. Unadjusted Price Data | by Palmarium AI | Medium
https://medium.com/@contact_9367/the-good-backtest-practices-adjusted-vs-unadjusted-price-data-35e15172b509

[28] [29] [38] [39] Beyond Pandas: Build a Quant-Level Research Database with Polars and DuckDB | by Nayab Bhutta | InsiderFinance Wire
https://wire.insiderfinance.io/beyond-pandas-build-a-quant-level-research-database-with-polars-and-duckdb-ca5b93930cd2?gi=c1188e9e0e75

[30] Volume-Weighted Average Price (VWAP): Definition and Calculation
https://www.investopedia.com/terms/v/vwap.asp

[32] [43] Risk Management Strategies for Algo Trading
https://www.luxalgo.com/blog/risk-management-strategies-for-algo-trading/

[33] [34] [44] [45] [46] [49] [50] Order (exchange) - Wikipedia
https://en.wikipedia.org/wiki/Order_(exchange)

[35] [36] [40] [41] [42] Machine Learning Trading Systems Guide - TradersPost Blog
https://blog.traderspost.io/article/machine-learning-trading-systems

[37] GitHub - thraizz/freqtrade-dashboard: A grafana+prometheus+freqtrade solution to monitor your trading algorithms.
https://github.com/thraizz/freqtrade-dashboard

[47] Automated Trading Systems: Design, Architecture & Low Latency
https://www.quantinsti.com/articles/automated-trading-system/

[48] Average True Range (ATR) Formula, What It Means, and How to Use It
https://www.investopedia.com/terms/a/atr.asp