Architecting the Intraday Trading Agent: A State Machine Approach

Introduction: The FSM as the Central Nervous System of the Trading Agent

For an automated intraday trading agent, the Finite State Machine (FSM) is not merely a convenient design pattern; it is the fundamental architectural choice for ensuring deterministic, resilient, and auditable behavior in the chaotic, event-driven environment of modern financial markets.[1] The core challenge in automated trading lies in managing a confluence of asynchronous data streams—real-time market data, order execution confirmations, news feeds, and internal signals—while reacting to unpredictable events and enforcing strict risk and operational protocols.[3] Naive implementations that rely on simple, linear logic or a monolithic block of conditional statements are inherently brittle; they lack the structure to handle the combinatorial complexity of market states and system events, making them prone to catastrophic failure.[5]

An FSM provides an elegant and robust solution to this problem. As a formal model of computation, an FSM makes the agent's behavior explicitly dependent on two factors: the nature of an incoming event and the system's current context, or its state.[2] This enforces a disciplined, structured approach to logic, which is paramount when financial capital is at risk. It transforms a potentially unmanageable web of

if-then-else statements into a clear, verifiable map of states and transitions, where the system can only be in one well-defined state at any given time.[6] This inherent structure simplifies complexity, enhances testability, and provides a clear audit trail of the agent's decision-making process.

This report will architect the "brain" of a sophisticated intraday trading agent as a multi-layered, hierarchical state machine (HSM) operating within a broader event-driven architecture (EDA). We will provide a definitive blueprint covering the

agent's primary operational states, the specific market and system events that drive its lifecycle, the nested FSMs required to manage lower-level tasks like order execution, and a practical guide to implementation frameworks in Python. The resulting architecture is designed not only to execute a trading strategy but to do so with the discipline, resilience, and risk-awareness required for a production environment.

I. The Foundational State Model: A High-Level Agent Lifecycle

The foundation of the agent's logic is a set of high-level states that define its operational status throughout the entire trading day. This model extends beyond a simple trading loop to encompass a complete lifecycle, from pre-market initialization to post-trade analysis and shutdown. Each state encapsulates a distinct set of responsibilities and permissible actions, forming the macro-structure of the agent's behavior.

A. State: INITIALIZING

The INITIALIZING state is the agent's entry point upon system startup. In this state, the agent is inert and has no active connections to external systems. Its primary function is to prepare for operation by loading all necessary configurations and establishing foundational services.

- **Responsibilities:** The core responsibilities in this state include parsing configuration files that contain critical parameters such as API keys for brokers and data vendors, strategy-specific parameters (e.g., lookback periods, risk/reward ratios), and global risk limits (e.g., maximum drawdown, maximum position size).[9] The agent will also initialize its logging system to ensure all subsequent actions are recorded for debugging and auditing. Concurrently, it begins the process of establishing network connections to data providers and brokerage execution venues. Finally, it instantiates all necessary sub-components, such as the signal generation module, the risk management engine, and the order management system.[9]
- **Entry/Exit Actions:** Upon entering this state, the agent immediately begins the

connection and loading sequence. A successful initialization, confirmed by healthy connections and valid configurations, triggers an event that transitions the agent to the AWAITING_MARKET_OPEN state. Any critical failure during this process—such as an inability to connect to the broker or a missing configuration file—triggers a transition to the SYSTEM_FAILURE state to prevent the agent from proceeding in a compromised condition.

B. State: AWAITING_MARKET_OPEN

This state is active during pre-market hours, after successful initialization. The agent is now fully connected and receiving data but is programmatically prohibited from submitting trading orders. Its purpose is to achieve a state of full readiness for the instant the market opens.

- **Responsibilities:** In this state, the agent actively monitors pre-market data feeds for the instruments on its watchlist.[9] This allows it to perform preliminary calculations, such as warming up technical indicators (e.g., moving averages) that require a history of data points. This "warm-up" prevents the agent from making decisions based on incomplete data at the market open. The agent also performs continuous health checks on its connections and system resources. This state is critical for mitigating overnight risk while ensuring the agent does not miss opportunities in the high-volume period immediately following the market open.[10]
- **Entry/Exit Actions:** On entry, the agent subscribes to the necessary real-time data streams for its watchlist. It remains in this state until it receives a MARKET_OPEN_EVENT, which triggers the transition to MONITORING_WATCHLIST.

C. State: MONITORING_WATCHLIST

This is the primary operational state during active trading hours when the agent holds no open positions. It is a "search" or "scan" mode, where the agent is actively analyzing market data to identify a valid trading opportunity based on its programmed strategy.

- **Responsibilities:** The agent's core function here is to consume high-frequency, real-time market data (e.g., tick or bar data) and feed it into its signal generation

logic.[12] This logic could range from simple technical indicator crossovers (e.g., 50-day moving average crossing the 200-day) to complex predictions from a deep reinforcement learning (DRL) model.[5] The agent continuously evaluates the output of this signal generator against its entry criteria.

- **Entry/Exit Actions:** The agent enters this state from AWAITING_MARKET_OPEN or after successfully closing a position from EXITING_POSITION. It will remain in this state, analyzing the market, until one of two primary events occurs. The detection of a valid trade signal via an ENTRY_SIGNAL_TRIGGERED event will transition the agent to ENTERING_POSITION. Alternatively, as the end of the trading day approaches, an END_OF_DAY_WARNING event will transition the agent to LIQUIDATING_POSITIONS to begin its shutdown routine.

D. State: ENTERING_POSITION

This is a transient state where the agent has identified a trading opportunity and is actively attempting to execute an order to open a new position. The focus shifts from analysis to execution management.

- **Responsibilities:** Upon receiving the entry signal, the first action is to perform a series of pre-trade risk checks.[14] These checks, handled by the Order Management System (OMS), are non-negotiable and verify that the prospective trade complies with all risk parameters: sufficient buying power, adherence to position size limits, and no violation of intraday loss limits. If the risk checks pass, the agent constructs the order, specifying the instrument, quantity, order type (e.g., limit, market), and price. This order is then submitted to the broker via the execution API.[9] This state's logic interacts intimately with a nested Order Lifecycle FSM (detailed in Section IV) to track the order's status precisely.
- **Entry/Exit Actions:** Entry is triggered by the ENTRY_SIGNAL_TRIGGERED event. A FILL_CONFIRMATION event from the broker, indicating the order has been successfully executed, transitions the agent to MANAGING_POSITION. If the broker rejects the order (REJECTION_EVENT) or if the order is canceled before being filled (CANCELLATION_EVENT), the agent transitions back to MONITORING_WATCHLIST to look for a new opportunity.

E. State: MANAGING_POSITION

Once an entry order is filled, the agent transitions to the MANAGING_POSITION state. This is arguably the most complex state, where the agent holds an open position and is actively managing its risk and potential profit.

- **Responsibilities:** The agent's primary duties in this state are to continuously monitor the real-time Profit and Loss (P&L) of the open position. It must vigilantly track the market price relative to its pre-defined stop-loss and take-profit levels.[9] More advanced strategies may involve dynamic adjustments to these levels, such as implementing a trailing stop that moves the stop-loss price up as the trade becomes more profitable. The agent also continues to evaluate its strategy logic for any signals that might warrant an early exit. Due to its complexity, this state is best modeled as a Hierarchical State Machine (HSM), a concept explored in detail in Section III.
- **Entry/Exit Actions:** Entry is triggered by a fill confirmation. The agent will exit this state and transition to EXITING_POSITION upon the occurrence of several possible events: STOP_LOSS_HIT, PROFIT_TARGET_HIT, an explicit EXIT_SIGNAL_TRIGGERED from the strategy logic, or a MANUAL_EXIT_COMMAND from a human operator.

F. State: EXITING_POSITION

This is another transient state, analogous to ENTERING_POSITION, where the agent is actively working to close its open position.

- **Responsibilities:** The agent's sole responsibility here is to construct and submit the closing order (e.g., a sell order to close a long position). The logic for constructing this order might differ based on the event that triggered the exit; for instance, a stop-loss hit might trigger an aggressive market order to ensure immediate execution, while a take-profit hit might use a limit order.
- **Entry/Exit Actions:** Upon entry, the closing order is submitted to the broker. A FILL_CONFIRMATION for this closing trade signifies that the agent is now "flat" (holds no position), triggering a transition back to the MONITORING_WATCHLIST state to search for the next trade.

G. State: LIQUIDATING_POSITIONS (End-of-Day Routine)

This is a critical, non-negotiable risk management state. It is entered a configurable amount of time before the market close to ensure the agent adheres to its intraday mandate by holding no positions overnight.[10]

- **Responsibilities:** The first action in this state is to cancel any and all open orders that have not been filled. If the agent still holds an open position, it must immediately transition to the EXITING_POSITION state to flatten it. The exit logic in this context is typically more aggressive, favoring market orders to guarantee execution before the close, even at the cost of some slippage. The agent is prohibited from initiating any new positions while in this state.
- **Entry/Exit Actions:** This state is triggered by the END_OF_DAY_WARNING_EVENT. Once the agent confirms that it is flat and has no working orders, it transitions to AWAITING_MARKET_CLOSE.

H. State: AWAITING_MARKET_CLOSE

In this state, the agent is confirmed to be flat and is simply waiting for the official end of the trading session.

- **Responsibilities:** The agent may continue to stream market data for post-trade analysis or to record closing prices, but all trading capabilities are disabled. It is effectively in a read-only mode.
- **Entry/Exit Actions:** The MARKET_CLOSE_EVENT triggers the transition to POST_TRADE_PROCESSING.

I. State: POST_TRADE_PROCESSING

After the market has closed, the agent enters this state to perform its end-of-day bookkeeping and reporting duties. It is fully disconnected from live market activity.

- **Responsibilities:** The agent's tasks include reconciling its internal record of trades with the broker's official statements, calculating final daily P&L figures, generating performance metrics and reports (e.g., Sharpe ratio, max drawdown) [3],

and persisting its final state and all relevant data to a database or disk for long-term storage and analysis.[16] This ensures a clean shutdown and prepares the agent for the next trading session.
- **Entry/Exit Actions:** Once all post-trade tasks are complete, the agent transitions to the final TERMINATED state.

J. State: SYSTEM_FAILURE / PAUSED

This is a critical "safe mode" state entered upon unrecoverable errors or by manual command. It serves as a circuit breaker to prevent further losses or erratic behavior in the event of a system malfunction.

- **Responsibilities:** Upon entering this state, the agent's first priority is to mitigate risk. It should immediately attempt to cancel all open orders and, if possible, alert a human operator through mechanisms like email or SMS notifications.[16] All trading logic is halted. A
  PAUSED sub-state might allow for a manual resume, while a full SYSTEM_FAILURE (e.g., due to loss of broker connectivity) is a terminal state that requires a full system restart and manual intervention.[1]
- **Entry/Exit Actions:** This state is a "sink," meaning there are no automated transitions out of it. It is entered from any other state upon a critical failure event and can only be exited through manual intervention.

A well-designed state model does more than just control execution flow; it serves as a direct implementation of the system's risk management policy. Many systems treat risk management as an external module that the trading logic must consult.[1] A more robust architectural approach embeds these policies directly into the state machine's structure. For example, the very existence of the

LIQUIDATING_POSITIONS state and the hard-coded transition to it near the market close is the embodiment of the "no overnight positions" rule.[10] The transition from

ENTERING_POSITION to MANAGING_POSITION is gated by pre-trade risk checks; the transition is impossible if the checks fail, thus enforcing risk limits at the most fundamental level. The SYSTEM_FAILURE state acts as the ultimate circuit breaker. This design philosophy elevates the FSM from a mere logic controller to an active and intrinsic risk management framework, providing a much stronger guarantee of safety

and compliance than a loosely coupled, advisory risk module.

The following table formalizes the high-level states of the agent, providing a clear specification for development.

Table 1: Agent State Definitions

State Name	Description	Primary Responsibilities	Key Entry Actions	Key Exit Actions
INITIALIZING	Agent startup and setup. Inert, no connections.	Load configs, API keys, risk limits. Initialize logging and sub-components.	Begin connection process to broker/data feeds.	Transition to AWAITING_MARKET_OPEN on success; SYSTEM_FAILURE on error.
AWAITING_MARKET_OPEN	Pre-market readiness. Connected but no trading.	Monitor pre-market data, warm up indicators, perform health checks.	Subscribe to real-time data for watchlist instruments.	Transition to MONITORING_WATCHLIST on MARKET_OPEN_EVENT.
MONITORING_WATCHLIST	Primary search state during market hours. No open positions.	Consume real-time data, run signal generation logic, evaluate entry conditions.	Start market data analysis loop.	Transition to ENTERING_POSITION on ENTRY_SIGNAL_TRIGGERED; LIQUIDATING_POSITIONS on END_OF_DAY_WARNING.
ENTERING_POSITION	Transient state to open a new position.	Perform pre-trade risk checks, construct order, submit order to broker.	Submit entry order to the execution venue.	Transition to MANAGING_POSITION on FILL_CONFIRMATION; MONITORING_WATCHLIST on REJECTION or CANCELLATION.

MANAGING_POSITION	Actively managing an open position.	Monitor P&L, track stop-loss/take-profit, evaluate exit conditions.	Record position details (entry price, size).	Transition to EXITING_POSITION on STOP_LOSS_HIT, PROFIT_TARGET_HIT, or other exit signals.
EXITING_POSITION	Transient state to close an existing position.	Construct closing order, submit order to broker.	Submit closing order to the execution venue.	Transition to MONITORING_WATCHLIST on FILL_CONFIRMATION.
LIQUIDATING_POSITIONS	End-of-day risk management.	Cancel all open orders, flatten any existing positions aggressively.	Disable new position entries.	Transition to AWAITING_MARKET_CLOSE once flat.
AWAITING_MARKET_CLOSE	Post-liquidation, waiting for market close.	Monitor data for reporting, no trading activity.	Confirm agent is flat with no working orders.	Transition to POST_TRADE_PROCESSING on MARKET_CLOSE_EVENT.
POST_TRADE_PROCESSING	End-of-day bookkeeping and reporting.	Reconcile trades, calculate P&L, generate reports, persist state.	Disconnect from live feeds.	Transition to TERMINATED upon completion.
SYSTEM_FAILURE / PAUSED	Safe mode triggered by critical error or manual command.	Halt all trading, attempt to cancel open orders, send alerts.	Log the critical error condition.	Requires manual intervention to reset/exit.

II. Orchestrating the Flow: State Transition Events and Triggers

The states define *what* the agent can be doing, but the events define *why* and *when* it changes its behavior. These events are the nervous impulses of the system, triggering transitions between states. A comprehensive taxonomy of these events is essential for building a robust agent that can react appropriately to the full spectrum of market and system conditions. These events can be categorized by their source: time, market data, internal position management, and system-level commands.

A. Time-Based Events (The Pacemaker)

These events are driven by the clock and provide the fundamental rhythm for the trading day. They are predictable and essential for adhering to the market's schedule.

- **MARKET_OPEN_EVENT**: This event, typically triggered at 9:30 AM ET for US equities, signals the official start of the main trading session. It is the trigger that moves the agent from a passive, preparatory state to an active, analytical one. It causes the transition from AWAITING_MARKET_OPEN to MONITORING_WATCHLIST.[15]
- **END_OF_DAY_WARNING_EVENT**: This is a configurable, time-based trigger that fires at a pre-set time before the market close (e.g., 3:45 PM ET for a 4:00 PM ET close). Its purpose is to initiate the mandatory end-of-day liquidation process to eliminate overnight risk. It triggers the transition from MONITORING_WATCHLIST or MANAGING_POSITION to LIQUIDATING_POSITIONS.[15]
- **MARKET_CLOSE_EVENT**: This event marks the official end of the trading session. It triggers the transition from AWAITING_MARKET_CLOSE to POST_TRADE_PROCESSING, moving the agent from its read-only observation mode to its offline bookkeeping duties.
- **HEARTBEAT_EVENT**: This is a periodic, high-frequency internal event (e.g., every 1 second or even more frequently). It doesn't typically cause a major state transition but is used to trigger actions *within* a state. For example, in the MANAGING_POSITION state, the HEARTBEAT_EVENT would trigger the recalculation of the position's real-time P&L and check if any trailing stop conditions have been met.

B. Market Data & Signal Events (The Senses)

These events are generated by the agent's analysis of incoming market data. They represent the agent's "perception" of the market and are the primary drivers of trading decisions.

- **ENTRY_SIGNAL_TRIGGERED**: This is the core event that initiates a trade. It is generated by the strategy logic when its conditions for entry are met. This could be a simple technical pattern like a moving average crossover [5], a breakout of a price channel [17], or a complex prediction from a machine learning model.[10] Crucially, this event must carry a data payload specifying the details of the trade, such as
  {symbol, direction (long/short), confidence_score}. It triggers the transition from MONITORING_WATCHLIST to ENTERING_POSITION.[18]
- **EXIT_SIGNAL_TRIGGERED**: This event is generated by the strategy logic when it determines that an existing position should be closed for reasons other than hitting a fixed stop-loss or profit target. For example, a momentum strategy might generate an exit signal if its momentum indicators weaken. This event triggers the transition from MANAGING_POSITION to EXITING_POSITION.
- **MARKET_REGIME_SHIFT_EVENT**: More advanced agents can employ models to classify the overall market's character. A Hidden Markov Model (HMM), for instance, can identify transitions between "trending," "ranging," or "volatile" regimes.[19] A clustering algorithm can detect distinct market states based on microstructure features.[20] An event signaling such a shift might not cause a full state transition but could trigger an internal action, such as widening stop-loss levels in response to increased volatility.
- **SIGNIFICANT_PRICE_VOLUME_EVENT**: This event is triggered when a price or volume movement in an instrument exceeds a predefined statistical threshold. It can indicate a significant market event, such as a news release or the start of a new trend, and can be used as a conditional input for other transitions.[21]

C. Position & Order Management Events (Feedback Loop)

These events originate from the agent's interaction with the broker and its management of its own portfolio. They provide the critical feedback loop that confirms the outcome of trading actions.

- **STOP_LOSS_HIT**: This event is generated internally when the agent observes that

the market price of an asset it holds has crossed its predefined stop-loss level. This is a primary risk management trigger, causing an immediate transition from MANAGING_POSITION to EXITING_POSITION.[9]

- **PROFIT_TARGET_HIT**: This event is generated internally when the market price reaches the agent's predefined take-profit level. It also triggers the MANAGING_POSITION to EXITING_POSITION transition.[9]
- **ORDER_FILL_CONFIRMATION**: This is an external event received from the broker's API, confirming that a submitted order has been executed. It can be a partial or full fill. This event is critical, as it provides the ground truth about the agent's position. Its payload must include {symbol, quantity_filled, execution_price, order_id}. A fill on an entry order triggers the transition from ENTERING_POSITION to MANAGING_POSITION. A fill on a closing order triggers the transition from EXITING_POSITION back to MONITORING_WATCHLIST.[14]
- **ORDER_REJECTED_EVENT**: An event from the broker indicating that an order submission was rejected (e.g., due to insufficient funds, invalid symbol, or a risk limit breach at the broker level). This immediately transitions the agent from ENTERING_POSITION back to MONITORING_WATCHLIST and should trigger an alert.
- **ORDER_CANCELED_EVENT**: An event from the broker confirming that a previous request to cancel a working order was successful. This is essential for ensuring the agent's internal state accurately reflects the status of its orders on the exchange.

D. System & Manual Events (Health and Control)

These events relate to the operational health of the agent itself and allow for human oversight and intervention.

- **CONNECTIVITY_LOST_EVENT**: This event is triggered when the agent's connection to a critical service, like the brokerage API or the primary market data feed, is lost. This is a critical failure that should trigger a transition from any active state to SYSTEM_FAILURE to prevent the agent from operating "blind".[4]
- **RISK_LIMIT_BREACH_EVENT**: This event is generated by an internal risk management component when a global risk limit is violated, such as the maximum daily drawdown or maximum gross exposure.[1] Depending on severity, it could trigger a transition to LIQUIDATING_POSITIONS to de-risk, or to PAUSED to halt new trading.

- **MANUAL_OVERRIDE_EVENT**: This event originates from a human operator via a control dashboard. It allows for commands like "PAUSE TRADING," "RESUME TRADING," or the emergency "FLATTEN ALL POSITIONS," providing an essential layer of human control over the automated system.

Defining these events with specific names and data payloads establishes a formal, versioned API between the FSM and the other components of the trading system. This architectural choice is more profound than it appears. By creating this "Event-as-API" contract, the system can be built in a highly modular, decoupled fashion, often using an event bus like Kafka or RabbitMQ.[23] For example, a separate microservice dedicated to natural language processing of news feeds can publish a

NEWS_SENTIMENT_CHANGED event to the bus. The trading agent FSM, as a subscriber, can consume this event and decide if it warrants a state change, all without the news service needing any knowledge of the trading agent's internal logic.[25] This decoupling allows for independent development, testing, and upgrading of system components. The ML model generating

ENTRY_SIGNAL_TRIGGERED events can be swapped out for a new version, and as long as it adheres to the event contract, the core FSM agent requires zero modification. This dramatically enhances system evolvability and maintainability, which are critical attributes for any complex, long-lived trading platform.

The following State Transition Matrix provides a complete and unambiguous map of the agent's logical flow, specifying exactly which events can cause a transition between any two states.

Table 2: State Transition Matrix

From State / To State	INITIALIZING	AWAITING_MARKET_OPEN	MONITORING_WATCHLIST	ENTERING_POSITION	MANAGING_POSITION	EXITING_POSITION	LIQUIDATING_POSITIONS	AWAITING_MARKET_CLOSE	POST_TRADE_PROCESSING	TERMINATED	SYSTEM_FAILURE
INITIALIZING		INIT_SUCCESS									INIT_FAILURE

AWAITING_MARKET_OPEN			MARKET_OPEN_EVENT								CONNECTIVITY_LOST
MONITORING_WATCHLIST				ENTRY_SIGNAL_TRIGGERED			END_OF_DAY_WARNING_EVENT				CONNECTIVITY_LOST
ENTERING_POSITION			ORDER_REJECTED, ORDER_CANCELED		ORDER_FILL_CONFIRMATION						CONNECTIVITY_LOST
MANAGING_POSITION						STOP_LOSS_HIT, PROFIT_TARGET_HIT, EXIT_SIGNAL_TRIGGERED,	END_OF_DAY_WARNING_EVENT				CONNECTIVITY_LOST

						MANUAL_EXIT					
EXITING_POSITION			ORDER_FILL_CONFIRMATION								CONNECTIVITY_LOST
LIQUIDATING_POSITIONS						POSITION_HELD		ALL_POSITIONS_FLAT			CONNECTIVITY_LOST
AWAITING_MARKET_CLOSE									MARKET_CLOSE_EVENT		CONNECTIVITY_LOST
POST_TRADE_PROCESSING										PROCESSING_COMPLETE	PROCESSING_ERROR
TERMINATED											

SYSTEM_FAILURE											

Note: The table shows primary transitions. Manual overrides and global risk limit breaches can trigger transitions to PAUSED or LIQUIDATING_POSITIONS from most active states.

III. Architectural Deep Dive: Hierarchical State Machines for Advanced Agents

As the complexity of a trading agent's logic grows, a simple "flat" FSM with a single layer of states can become unwieldy and difficult to maintain. This is a well-known problem in FSM design referred to as "state explosion," where the number of states grows combinatorially with the number of conditions the system must track.[26] To manage this complexity effectively, we must evolve our design from a flat FSM to a Hierarchical State Machine (HSM).

A. The Limits of Flat State Machines

Consider the MANAGING_POSITION state. In a real-world strategy, the way a position is managed is not static. The agent might initially have a wide, fixed stop-loss. After the price moves favorably, it might switch to a dynamic trailing stop based on the Average True Range (ATR). If the strategy involves scaling out, it might enter a mode where it sells portions of the position at different targets.

In a flat FSM, each of these management styles would require a distinct top-level state: MANAGING_POSITION_INITIAL, MANAGING_POSITION_TRAILING, MANAGING_POSITION_SCALING_OUT, and so on. This approach has severe drawbacks. First, it clutters the main state diagram with highly specific, low-level details. Second, it leads to massive code duplication. For example, the logic to handle

an END_OF_DAY_WARNING event or a MANUAL_EXIT_COMMAND would need to be replicated in every one of these MANAGING_POSITION_* states. This violates the "Don't Repeat Yourself" (DRY) principle and makes the system brittle and hard to modify.

B. Introduction to Hierarchical State Machines (HSMs)

HSMs, also known as statecharts, solve this problem by introducing the concept of state nesting.[26] In an HSM, a state can itself contain another, complete state machine. This creates a parent-child relationship between states, with several powerful architectural benefits [26]:

1. **Complexity Management:** HSMs allow for the decomposition of a complex state into a set of simpler, more manageable substates. The internal complexity of managing a position is hidden, or encapsulated, within a single MANAGING_POSITION superstate.
2. **Behavioral Inheritance:** Substates automatically inherit the transitions and actions of their superstates.[26] If a transition for an event like MANUAL_EXIT_COMMAND is defined on the MANAGING_POSITION superstate, it is automatically available to all of its substates (INITIAL_RISK, TRAILING_STOP, etc.). There is no need to redefine this transition for each substate, which dramatically reduces code duplication and improves maintainability.
3. **Encapsulation and Modularity:** The logic for a specific, complex behavior is entirely contained within its superstate. This promotes a clean, modular design where different parts of the system's logic can be developed and tested in isolation.

C. Case Study: Decomposing the MANAGING_POSITION State

To illustrate the power of this approach, the MANAGING_POSITION state is redefined not as a single state, but as a **superstate** that contains its own nested FSM for trade management.

- **Superstate:** MANAGING_POSITION
    - **Inherited Transitions:** This superstate defines transitions that are common to

all forms of position management. For example, it will always transition to EXITING_POSITION upon receiving a RISK_LIMIT_BREACH_EVENT or an END_OF_DAY_WARNING_EVENT.

- **Substates within MANAGING_POSITION:**
  - INITIAL_RISK (Initial Substate): This is the entry point for the nested FSM, entered immediately after a position is opened. In this state, the agent enforces a fixed, initial stop-loss and take-profit target based on the entry signal's parameters. It is a relatively static state, waiting for a significant price move or an exit signal.
  - AWAITING_EXIT_SIGNAL: A more general substate where the initial risk parameters are set, and the sub-machine is simply waiting for one of the primary exit events (STOP_LOSS_HIT, PROFIT_TARGET_HIT, etc.) to occur.
  - TRAILING_STOP: This substate is entered after the price has moved a significant, predefined amount in the trade's favor. The logic within this state is active and dynamic; on every HEARTBEAT_EVENT, it recalculates the stop-loss level based on a trailing mechanism, such as the low of the last N bars, a multiple of the ATR, or a moving average.
  - SCALING_OUT: If the trading strategy dictates taking profits at multiple levels, this substate is entered after the first partial profit target is hit. This substate's logic is responsible for submitting orders to sell portions of the position and tracking the remaining size.
- **Transitions within the MANAGING_POSITION Sub-machine:**
  - PRICE_BREAKEVEN_PLUS_X_HIT: An internal event triggered when the price moves far enough for the initial stop to be moved to breakeven or better. This would trigger a transition from INITIAL_RISK to TRAILING_STOP.
  - PARTIAL_TARGET_HIT: An internal event triggered when the first profit target is reached. This could cause a transition from AWAITING_EXIT_SIGNAL to SCALING_OUT.
  - STOP_LOSS_HIT / FINAL_TARGET_HIT: These events, when processed by the sub-machine, trigger an important hierarchical action. The sub-machine's action would be to initiate the exit, which in turn generates an event that causes the *superstate* (MANAGING_POSITION) to transition to the top-level EXITING_POSITION state. This demonstrates how events can be handled at different levels of the hierarchy to orchestrate complex behavior.[26]

This hierarchical structure provides a powerful mechanism for creating modular, reusable strategy components. Different trading strategies often have unique trade management rules. A classic trend-following strategy might employ a sophisticated trailing stop logic to let profits run, while a mean-reversion strategy might use fixed,

tight profit targets and never trail the stop.[17]

With an HSM, the MANAGING_POSITION superstate acts as a plug-and-play container. One can define different "trade management modules"—each a self-contained FSM—for different strategies. When the agent's signal generator produces an ENTRY_SIGNAL_TRIGGERED event, the payload can include the name of the required management module. The agent then loads the corresponding sub-machine into the MANAGING_POSITION superstate for the duration of that trade. This architecture allows a single agent to trade a diverse portfolio of strategies, each with its own bespoke risk and trade management logic, without altering the agent's core high-level state machine. This approach is directly inspired by concepts from hierarchical reinforcement learning, where a high-level agent selects from a pool of specialized, low-level agents to perform specific tasks.[28]

IV. The Execution Layer: Modeling the Order Lifecycle FSM

At the lowest level of the agent's operational hierarchy lies a component that is absolutely critical for robust, production-grade trading: a dedicated FSM for managing the lifecycle of a single order. A common but dangerous oversimplification is to treat order submission as an atomic "fire-and-forget" operation. In reality, once an order is sent to a broker, it enters a complex lifecycle with multiple possible states and outcomes.[14] The agent must track this lifecycle with precision to maintain an accurate view of its position and to handle errors gracefully. Failure to do so can lead to severe operational risks, such as submitting duplicate orders, failing to act on a fill, or believing an order is working when it has actually been rejected.

A. The Necessity of an Order FSM

The interaction with a brokerage execution API is asynchronous. When the agent sends an order, it receives an initial acknowledgment, but the final outcome—fill, partial fill, cancellation, rejection—arrives later as a separate series of events. The agent must be ableto correlate these subsequent events with the original order and update its internal state accordingly. An Order Lifecycle FSM provides a formal,

reliable mechanism for managing this process for every single order the agent generates.

B. States of the Order FSM

Each instance of an order placed by the agent will have its own instance of this FSM. The states represent the order's status from the perspective of the agent's interaction with the broker.

- PENDING_SUBMIT (Initial State): The order object has been created locally within the agent's memory but has not yet been transmitted over the network to the broker.
- SUBMITTED: The order has been successfully sent to the broker's API endpoint. The agent is now awaiting a response from the broker's system.
- ACKNOWLEDGED: The broker has sent a confirmation that the order has been received and is now "working" in the market. It is now live on the exchange's order book.
- PARTIALLY_FILLED: The broker has sent a notification that a portion of the order has been executed. The remaining portion of the order is still active and working in the market. The FSM remains in this state as subsequent partial fills arrive.
- FILLED: The broker has confirmed that the order has been fully executed. This is a **terminal state** for the FSM, signifying the end of a successful lifecycle.
- PENDING_CANCEL: The agent has sent a request to the broker to cancel a working order (one that is in the ACKNOWLEDGED or PARTIALLY_FILLED state). It is now awaiting confirmation of the cancellation.
- CANCELED: The broker has confirmed that the working order has been successfully canceled. This is another **terminal state**.
- REJECTED: The broker has rejected the order submission. This can happen for various reasons, such as invalid parameters, insufficient margin, or a breach of pre-trade risk controls at the broker level. This is also a **terminal state**.

C. Events and Transitions for the Order FSM

The events that drive the Order FSM are almost exclusively messages received from

the broker's execution API, which are then translated into the FSM's event vocabulary.

- API_SEND_SUCCESS: An internal event confirming the order was sent without network errors. Triggers PENDING_SUBMIT -> SUBMITTED.
- BROKER_ACK: A message from the broker acknowledging receipt. Triggers SUBMITTED -> ACKNOWLEDGED.
- BROKER_PARTIAL_FILL: A fill report from the broker for a quantity less than the remaining order size. Triggers ACKNOWLEDGED -> PARTIALLY_FILLED or PARTIALLY_FILLED -> PARTIALLY_FILLED.
- BROKER_FULL_FILL: A fill report from the broker for the full remaining quantity. Triggers ACKNOWLEDGED or PARTIALLY_FILLED -> FILLED.
- BROKER_REJECT: A rejection message from the broker. Triggers SUBMITTED -> REJECTED.
- BROKER_CANCEL_CONFIRM: A confirmation that a cancellation request was successful. Triggers PENDING_CANCEL -> CANCELED.

D. Interaction with the Parent Agent FSM

This nested Order FSM does not operate in a vacuum. It is a sub-component managed by the higher-level agent FSM, specifically within the ENTERING_POSITION and EXITING_POSITION states. The interaction is bidirectional:

1. **Creation:** When the agent enters the ENTERING_POSITION state, it creates an instance of the Order FSM for the entry order it is about to place.
2. **Notification:** The terminal states of the Order FSM (FILLED, CANCELED, REJECTED) are critical because they generate events that are consumed by the parent agent FSM. For example, when an Order FSM instance reaches the FILLED state, its exit action is to publish the ORDER_FILL_CONFIRMATION event to the main event bus. The parent agent FSM consumes this event, which in turn triggers its own state transition from ENTERING_POSITION to MANAGING_POSITION.

This hierarchical interaction is the key to robust execution. The main agent delegates the messy, low-level details of order tracking to a specialized sub-machine, and only reacts to the final, unambiguous outcomes.

The following table provides a clear specification for this critical sub-component.

Table 3: Order Lifecycle FSM States and Transitions

State Name	Description	Triggering Event (from Broker API)	Action / Notification to Parent Agent
PENDING_SUBMIT	Order created locally, not yet sent.	SUBMIT_COMMAND (internal)	None
SUBMITTED	Order sent to broker, awaiting acknowledgement.	API_SEND_SUCCESS (internal)	None
ACKNOWLEDGED	Broker confirms order is live/working.	BROKER_ACK	Publish ORDER_IS_WORKING event.
PARTIALLY_FILLED	A portion of the order has been executed.	BROKER_PARTIAL_FIL L	Publish ORDER_PARTIAL_FILL event with fill details.
FILLED	The order is fully executed. (Terminal State)	BROKER_FULL_FILL	Publish ORDER_FILL_CONFIR MATION event with full details.
PENDING_CANCEL	Cancellation request sent for a working order.	CANCEL_COMMAND (internal)	None
CANCELED	Broker confirms order was successfully canceled. (Terminal State)	BROKER_CANCEL_C ONFIRM	Publish ORDER_CANCELED_E VENT.
REJECTED	Broker rejected the order submission. (Terminal State)	BROKER_REJECT	Publish ORDER_REJECTED_E VENT with reason.

V. Implementation in Python: Frameworks and Architectural Patterns

With the conceptual architecture of the hierarchical state machine defined, the focus shifts to practical implementation. Python, with its rich ecosystem of libraries and strong support for data science, is a dominant language in quantitative finance.[30] This section analyzes suitable Python libraries and design patterns for building the FSM-based trading agent.

A. Library-Based Implementations: A Comparative Analysis

For a system of this complexity, leveraging a dedicated FSM library is strongly recommended over building the entire mechanism from scratch. These libraries handle the boilerplate logic of state storage, transition validation, and callback dispatching, allowing developers to focus on the application's core business logic.

1. pytransitions

- **Overview:** pytransitions is a lightweight, object-oriented, and highly flexible library that is popular for its ease of use and powerful features.[31] Its core design philosophy is to "decorate" an existing model object (e.g., an Agent class) with stateful behavior, adding a state attribute and methods that trigger transitions.[31]
- **Strengths:** Its key strengths lie in its intuitive API, excellent support for hierarchical state machines (via pytransitions.extensions.HierarchicalMachine), and conditional transitions.[31] A particularly valuable feature for development and debugging is the transitions-gui extension, which can visualize the state machine in a web browser in real-time, showing the current state and allowing transitions to be triggered by clicking on the graph.[33]
- **Code Example (Conceptual):**

```python
from transitions.extensions import HierarchicalMachine

class TradingAgent:
```

```
    # Agent properties and methods would be defined here
    pass

agent = TradingAgent()

states =},
    'LIQUIDATING_POSITIONS', 'TERMINATED'
]

transitions =

machine = HierarchicalMachine(model=agent, states=states,
transitions=transitions, initial='INITIALIZING')

# Trigger a transition
agent.init_success()
print(agent.state) # Output: AWAITING_MARKET_OPEN
```

2. python-statemachine

- **Overview:** python-statemachine is a more formal and structured library that emphasizes correctness and validation.[34] In this paradigm, the state machine is typically defined as its own class that inherits from StateMachine, rather than being attached to an external model.
- **Strengths:** Its most significant advantage is its rigorous set of validations at class definition time. It ensures that there is exactly one initial state, that no transitions emanate from final states, and that all states are reachable, preventing common logical errors before the code is even run.[34] It also boasts first-class, native support for asyncio, making it exceptionally well-suited for building modern, asynchronous trading systems. Its dependency injection mechanism for callbacks is also a powerful feature for writing clean, testable action handlers.[34]
- **Code Example (Conceptual):**
  Python
  ```
  from statemachine import StateMachine, State
  ```

```
class TradingAgentFSM(StateMachine):
    # Define states
    initializing = State(initial=True)
    awaiting_market_open = State()
    monitoring_watchlist = State()
    managing_position = State() # This could be further refined with HSM features
    terminated = State(final=True)

    # Define transitions
    init_success = initializing.to(awaiting_market_open)
    market_open = awaiting_market_open.to(monitoring_watchlist)
    #... other transitions

    # Define actions (callbacks)
    def on_enter_awaiting_market_open(self):
        print("Agent is now awaiting market open...")

agent_fsm = TradingAgentFSM()
agent_fsm.init_success()
print(agent_fsm.current_state.id) # Output: awaiting_market_open
```

B. The State Design Pattern: A First-Principles Approach

- **Overview:** Before dedicated libraries were common, FSMs were implemented using the classic State design pattern.[6] In this object-oriented pattern, each state is implemented as a separate class that conforms to a common State interface. The main Context class (our TradingAgent) holds a reference to the current concrete state object and delegates all state-dependent behavior to it. State transitions are achieved by having the Context or the state objects themselves change the reference to a new state object.
- **Strengths:** This approach offers maximum control and has no external dependencies. It can be a good educational tool for understanding the mechanics of state machines.
- **Weaknesses:** The State pattern is extremely verbose. It requires a significant amount of boilerplate code to manage the states, transitions, and callbacks, which the libraries handle automatically. For a complex HSM, maintaining this

structure manually is error-prone and inefficient.[6]

- **Code Example (Conceptual):**

```python
from abc import ABC, abstractmethod

class AgentState(ABC):
    @abstractmethod
    def on_market_open(self, agent):
        pass
    #... other event handlers

class AwaitingMarketOpenState(AgentState):
    def on_market_open(self, agent):
        print("Market is open. Transitioning to monitoring.")
        agent.state = MonitoringWatchlistState()

class MonitoringWatchlistState(AgentState):
    def on_market_open(self, agent):
        print("Already monitoring; no transition.")
    #...
```

C. The Broader Context: Event-Driven Architecture (EDA)

The FSM, regardless of its implementation, is not a standalone application. It is the "brain" that must be connected to the "senses" and "muscles" of the trading system. The most robust and scalable way to achieve this is through an Event-Driven Architecture (EDA).[23]

- **Components of the EDA:**
  1. **Event Producers:** These are independent services or modules that generate the events defined in Section II. Examples include a MarketDataHandler that consumes a raw feed and publishes PRICE_UPDATE events, a SignalGenerator that publishes ENTRY_SIGNAL_TRIGGERED events, and a BrokerConnector that translates API messages into ORDER_FILL_CONFIRMATION events.[24]
  2. **Event Bus/Broker:** This is the central nervous system of the architecture. It is a messaging queue (e.g., Apache Kafka, RabbitMQ) that decouples the producers from the consumers. Producers publish events to specific "topics"

on the bus without knowing who, if anyone, is listening.[24]

3. **Event Consumers:** The FSM-based trading agent is the primary consumer. It subscribes to the topics it cares about (e.g., market-data.SPY, signals.mean-reversion, orders.fills).

- **Code Example (Conceptual Main Loop):**

```python
from kafka import KafkaConsumer
import json

# Assume 'agent_fsm' is an initialized state machine instance
consumer = KafkaConsumer('trading_events', bootstrap_servers='localhost:9092')

for message in consumer:
    event = json.loads(message.value)
    event_type = event.get('type')
    event_payload = event.get('payload')

    if hasattr(agent_fsm, event_type):
        # The FSM library provides a trigger method for the event
        trigger_method = getattr(agent_fsm, event_type)
        trigger_method(**event_payload) # e.g., agent_fsm.entry_signal_triggered(symbol='SPY',...)
```

This EDA pattern provides immense flexibility. The FSM agent becomes a pure-logic component that simply reacts to a stream of events, making it highly testable and modular.

D. Architectural Recommendations

1. **Library Choice:** For a production-grade intraday trading agent, using a dedicated library is essential.
   - **python-statemachine** is the recommended choice for new, complex, and mission-critical projects. Its emphasis on correctness, compile-time validation, and first-class async support aligns perfectly with the requirements of financial systems where errors can be extremely costly.[34]
   - **pytransitions** is an excellent alternative, especially if the goal is to add

stateful logic to an existing object model or if rapid prototyping and visualization are top priorities. Its flexibility and powerful GUI are significant assets during the development phase.[31]

2. **Persistence:** The agent's state is volatile and resides in memory. To ensure resilience against crashes or restarts, the state must be persisted. On every state transition, the agent's current state (and the state of any open positions and orders) should be saved to a durable store like a database (e.g., Redis, PostgreSQL) or a file. On startup, the agent should first check this store to recover its last known state before initializing.[16]
3. **Architecture:** The recommended overall architecture is the FSM implemented with a robust library, operating as a consumer within a broader Event-Driven Architecture. This combination provides the best balance of logical rigor, scalability, and modularity.

Table 4: Python FSM Library Comparison

Feature	pytransitions	python-statemachine
API Style	Decorates an existing model object. Transitions are methods on the model.	Declarative. The FSM is its own class inheriting from StateMachine.
HSM Support	Yes, via HierarchicalMachine extension. Well-documented.	Supported, though less explicitly featured than in pytransitions.
Async Support	Yes, via an AsyncMachine extension.	Native, first-class support. Core design is async-aware.
Validation	Runtime validation (e.g., throws error on invalid transition).	Definition-time validation (e.g., checks for reachability, final states).
Key Strengths	Flexibility, ease of integration, excellent visualization GUI.	Strictness, correctness guarantees, native async, dependency injection.
Best Suited For	Rapid prototyping, adding state to existing models, projects benefiting from visualization.	New, mission-critical applications requiring high correctness and robust async behavior.

Conclusion and Strategic Recommendations

The architecture detailed in this report presents a robust and resilient blueprint for the "brain" of an intraday trading agent, centered on a Hierarchical State Machine (HSM) within an Event-Driven Architecture (EDA). This design moves beyond simple, brittle scripts to a professional-grade system capable of managing the immense complexity of live financial markets. By formalizing the agent's logic into a precise set of states, events, and transitions, the FSM provides determinism and auditability. By leveraging hierarchy, it manages complexity and enables strategy modularity. By nesting a dedicated FSM for the order lifecycle, it ensures reliable execution management.

The success of such a system, however, depends not only on the elegance of the architecture but also on the discipline of its implementation and operation. The full lifecycle of strategy development—from formulation and backtesting to paper trading and finally, cautious live deployment—is paramount.[9] The proposed event-driven architecture directly supports this lifecycle, allowing for the same FSM logic to be driven by historical data during backtesting, simulated data during paper trading, and live broker feeds in production, simply by swapping the event producers.

Based on this comprehensive analysis, the following strategic recommendations are provided for the development team:

1. **Design with Hierarchy from Day One:** Do not begin with a flat FSM with the intention of adding hierarchy later. The complexity of trade and risk management demands a hierarchical approach from the outset. Design the MANAGING_POSITION and ORDER_LIFECYCLE FSMs as integral, nested components of the initial architecture.
2. **Adopt Event-First Thinking:** Before writing implementation code, finalize the definition of the events—their names, triggers, and data payloads. These events form the contract that decouples the system's components. A well-defined event schema will guide the development of the FSM, signal generators, and data handlers, ensuring modularity and testability.
3. **Implement Comprehensive, Structured Logging:** Logging is not an afterthought; it is a critical feature. Every state transition, every action executed, and every event received must be logged in a structured format (e.g., JSON). This data is non-negotiable for post-mortem debugging, performance analysis, and

regulatory auditing.

4. **Embrace Visualization During Development:** For any FSM library chosen, make aggressive use of visualization tools like transitions-gui or the diagramming capabilities of python-statemachine.[33] Visually stepping through the state machine's logic during development is an invaluable technique for catching logical errors and understanding the complex interactions within the HSM.

By adhering to this architectural blueprint and these strategic principles, a development team can construct an intraday trading agent that is not only effective in its strategy execution but also robust, resilient, and manageable—the hallmarks of a professional automated trading system.

Ouvrages cités

1. Algorithmic Trading Briefing Note - Federal Reserve Bank of New York, dernier accès : août 12, 2025, https://www.newyorkfed.org/medialibrary/media/newsevents/news/banking/2015/SSG-algorithmic-trading-2015.pdf
2. Finite-state machine - Wikipedia, dernier accès : août 12, 2025, https://en.wikipedia.org/wiki/Finite-state_machine
3. Automate your trading: Algorithmic strategies explained | Trading knowledge | OANDA | US, dernier accès : août 12, 2025, https://www.oanda.com/us-en/trade-tap-blog/trading-knowledge/automate-your-trading-an-inside-look-at-algorithmic-strategies/
4. All You Need to Know about Automated Trading Systems - Platinum Trading Solutions, dernier accès : août 12, 2025, https://www.platinumtradingsolutions.com/automated-trading-systems/
5. Basics of Algorithmic Trading: Concepts and Examples - Investopedia, dernier accès : août 12, 2025, https://www.investopedia.com/articles/active-trading/101014/basics-algorithmic-trading-concepts-and-examples.asp
6. State Pattern in Python - Auth0, dernier accès : août 12, 2025, https://auth0.com/blog/state-pattern-in-python/
7. Key concept: Finite State Machine (FSM) - Quantum Leaps, dernier accès : août 12, 2025, https://www.state-machine.com/fsm
8. Use State Machines! - Richard Clayton - Silvrback, dernier accès : août 12, 2025, https://rclayton.silvrback.com/use-state-machines
9. Algo Trading Stages Explained Simply - QuantInsti Blog, dernier accès : août 12, 2025, https://blog.quantinsti.com/algo-trading-stages-explained-simply/
10. Deep Reinforcement Learning with Positional Context for Intraday Trading - arXiv, dernier accès : août 12, 2025, https://arxiv.org/html/2406.08013v1
11. Intraday trading patterns in an intelligent autonomous agent-based stock market, dernier accès : août 12, 2025, https://ideas.repec.org/a/eee/jeborg/v79y2011i3p226-245.html

12. Automated Trading Systems: Architecture, Protocols, Types of Latency - QuantInsti Blog, dernier accès : août 12, 2025, https://blog.quantinsti.com/automated-trading-system/
13. High-frequency trading using Azure Stream Analytics - Microsoft Learn, dernier accès : août 12, 2025, https://learn.microsoft.com/en-us/azure/stream-analytics/stream-analytics-high-frequency-trading
14. The Intricacies of Electronic Trading: Understanding the Order Life ..., dernier accès : août 12, 2025, https://sterlingtradingtech.com/news-insights/the-intricacies-of-electronic-trading-understanding-the-order-life-cycle
15. Three Automated Stock-Trading Agents: A Comparative Study - UCLA Computer Science Department, dernier accès : août 12, 2025, https://web.cs.ucla.edu/~sherstov/pdf/amec04-plat.pdf
16. The Lifecycle of an Algorithmic Trading Bot: From Optimization to ..., dernier accès : août 12, 2025, https://medium.com/ai-simplified-in-plain-english/the-lifecycle-of-an-algorithmic-trading-bot-from-optimization-to-autonomous-operation-3f9d5ceba12e
17. 20 Automated Trading Strategies 2025 - QuantifiedStrategies.com, dernier accès : août 12, 2025, https://www.quantifiedstrategies.com/automated-trading-systems/
18. Multi-Period Reversal Point Detection and Automated Trading Strategy | by FMZQuant | Jun, 2025 | Medium, dernier accès : août 12, 2025, https://medium.com/@FMZQuant/multi-period-reversal-point-detection-and-automated-trading-strategy-213815675d26
19. Intraday Application of Hidden Markov Models - QuantConnect.com, dernier accès : août 12, 2025, https://www.quantconnect.com/research/17900/intraday-application-of-hidden-markov-models/
20. Detecting intraday financial market states using temporal clustering - Oxford Man Institute of Quantitative Finance, dernier accès : août 12, 2025, https://www.oxford-man.ox.ac.uk/wp-content/uploads/2020/04/Detecting-Intraday-Financial-Market-States-Using-Temporal-Clustering.pdf
21. Algorithmic trading - Wikipedia, dernier accès : août 12, 2025, https://en.wikipedia.org/wiki/Algorithmic_trading
22. Trade Lifecycle – How Market Trades Actually Happen? - B2CORE™, dernier accès : août 12, 2025, https://b2core.com/news/trade-lifecycle-what-happens-after-you-place-a-market-order/
23. Event-Driven Architecture - AWS, dernier accès : août 12, 2025, https://aws.amazon.com/event-driven-architecture/
24. Event-Driven Architecture in Python for Trading - PyQuant News, dernier accès : août 12, 2025, https://www.pyquantnews.com/free-python-resources/event-driven-architecture-in-python-for-trading

25. The Complete Guide to Event-Driven Architecture - Solace, dernier accès : août 12, 2025, https://solace.com/what-is-event-driven-architecture/
26. Introduction to Hierarchical State Machines (HSMs) - Barr Group, dernier accès : août 12, 2025, https://barrgroup.com/blog/introduction-hierarchical-state-machines
27. (PDF) Hierarchical State Machines - ResearchGate, dernier accès : août 12, 2025, https://www.researchgate.net/publication/226020512_Hierarchical_State_Machines
28. Hierarchical Model-Based Deep Reinforcement Learning for Single-Asset Trading - MDPI, dernier accès : août 12, 2025, https://www.mdpi.com/2813-2203/2/3/31
29. State representation for the high-level agent. | Download Scientific Diagram - ResearchGate, dernier accès : août 12, 2025, https://www.researchgate.net/figure/State-representation-for-the-high-level-agent_fig1_370955433
30. PyFi - Learn Python for Finance, dernier accès : août 12, 2025, https://pyfi.com/
31. pytransitions/transitions: A lightweight, object-oriented finite state machine implementation in Python with many extensions - GitHub, dernier accès : août 12, 2025, https://github.com/pytransitions/transitions
32. transitions · PyPI, dernier accès : août 12, 2025, https://pypi.org/project/transitions/0.6.4/
33. pytransitions/transitions-gui: A frontend for transitions state machines - GitHub, dernier accès : août 12, 2025, https://github.com/pytransitions/transitions-gui
34. python-statemachine · PyPI, dernier accès : août 12, 2025, https://pypi.org/project/python-statemachine/
35. python-statemachine 2.5.0 - Read the Docs, dernier accès : août 12, 2025, https://python-statemachine.readthedocs.io/en/latest/readme.html
36. Event-Driven Architecture (EDA): A Complete Introduction - Confluent, dernier accès : août 12, 2025, https://www.confluent.io/learn/event-driven-architecture/
37. Event-Driven Architecture - System Design - GeeksforGeeks, dernier accès : août 12, 2025, https://www.geeksforgeeks.org/system-design/event-driven-architecture-system-design/
38. python-statemachine 2.5.0, dernier accès : août 12, 2025, https://python-statemachine.readthedocs.io/