ALGORITHMIC TRADING · SYSTEMS ARCHITECTURE

Finite-State Machines for Quantitative Trading Execution in Python

From Knight Capital to LangGraph

A technical guide to the pattern that separates the bots that survive from the ones that don't: explicit state, verifiable transitions and kill switches by construction — with runnable implementations (manual, transitions and python-statemachine), the FIX order lifecycle, property-based testing and the conceptual bridge to LangGraph.

PUBLISHED AT
pythonparatrading.com

SERIES
Architecture Fundamentals — Part 1

DATE
July 2026

Contents

1. The day 45 minutes cost 460 million

1.1 The Knight Capital disaster as a state failure

1.1.1 Official SEC timeline: 212 parent orders → >4M fills in 154 stocks, >397M shares, +$3.5bn/−$3.15bn positions, loss >$460M in 45 minutes

On August 1, 2012, between 9:30 and 10:15 a.m. ET, Knight Capital's SMARS order router processed 212 retail parent orders and, from them, produced more than 4 million fills in 154 stocks totaling more than 397 million shares. When the flow was stopped, it had accumulated a net long position of roughly $3.5 billion in 80 stocks and a net short position of roughly $3.15 billion in 74; the final loss exceeded $460 million according to the SEC order[1] (Knight announced ~$440 million pre-tax; both figures are correct in their context). In 75 stocks, its fills exceeded 20% of total market volume[2]: one firm's internal failure became a risk to third parties.

Date / time (ET)	Event
Jul 27–31, 2012	Staged deployment of the RLP code; a technician fails to copy the code to one of the eight SMARS servers, with no review by a second technician[1]
Aug 1, 8:01	An internal system starts generating 97 automated "BNET rejects" emails mentioning SMARS and the error "Power Peg disabled"; no one acts[3]
Aug 1, 9:30	Market open. The eighth server interprets a reused flag as Power Peg activation and enters a child-order sending loop[1]
9:30–10:15	212 parent orders → >4M fills, 154 stocks, >397M shares[1]
10:15	Knight stops the sending; unwanted positions of +$3.5bn / −$3.15bn[1]
Aug 5–6	~$400M in emergency financing to avoid collapse; in December Knight agrees to its sale to Getco[4]
Oct 16, 2013	$12M fine: first SEC enforcement action under Rule 15c3-5[3]

The timeline shows two clocks worth keeping separate. The incident clock —45 minutes, about $10 million per minute— is how long a machine with no stop state takes to destroy a firm. The signal clock —running since 8:01, an hour and a half before the open— is how long the system spent warning without a single transition connecting that signal to an action. At that loss rate, detecting, diagnosing, and deciding by committee costs more than most firms' capital: containment has to be structural, not deliberative.

1.1.2 The root cause was not a logic bug but a state bug: "dead" code reactivated, 97 ignored alert emails, literal SEC phrase: "halt SMARS's operations in response to its own aberrant activity"

The causal chain, according to the SEC, is a lesson in state theory disguised as an operational incident. Knight stopped using the Power Peg functionality in 2003 but never removed it from the code; in 2005 it moved the cumulative share-counting function to another point in the sequence and never tested it again[1]. In state-machine terms: a state believed to have been removed from the graph was still reachable, and the new code reused the flag that activated it. The eighth server fell into that state and entered a loop with no exit condition, because another part

of the system —which did know the parent orders were completed— did not communicate that to SMARS[1]: two components with divergent views of the same state.

The 97 emails were events emitted without a handler. And the SEC order says it without metaphors: Knight "did not have procedures in place to halt SMARS's operations in response to its own aberrant activity"[1]. There was no lack of stop code in the abstract; what was missing was stop as a state of the system.

1.2 The market is already a hierarchy of state machines

1.2.1 Regulatory circuit breakers as literal FSMs: MWCB 7%/13%/20% on the S&P 500 and LULD (Limit State 15s → 5 min halt); CME's Stop Logic stopped the 2010 Flash Crash with a 5-second pause

Regulators learned the lesson before many developers did. Market-Wide Circuit Breakers (MWCB), in their current form since April 8, 2013, are a literal finite-state machine (FSM): S&P 500 declines of 7% (Level 1) or 13% (Level 2) before 3:25 p.m. halt the entire market for 15 minutes; 20% (Level 3) closes it for the day, and each level can only trigger once per session[5]. Level 1 has been triggered four times: March 9, 12, 16, and 18, 2020[6]. At the single-security level, Limit Up-Limit Down (LULD) defines another automaton: if the price touches the band and does not return inside it within 15 seconds ("Limit State"), trading is paused for 5 minutes[7].

The strongest empirical precedent is the Flash Crash of May 6, 2010: a fundamental seller dumped 75,000 E-mini S&P 500 contracts (~$4.1 billion) using an algorithm with no price or time limits. At 2:45:28 p.m., CME's Stop Logic Functionality paused the E-mini for five seconds; the joint SEC-CFTC report documents that this minimal pause broke the selling cascade[8]. A five-second state transition did what no human could in twenty minutes.

1.2.2 SEC Rule 15c3-5 (Market Access): pre-trade controls and kill switch as a regulatory obligation; MiFID II RTS 6 Art. 12

Rule 15c3-5, adopted in 2010 and mandatory since July 2011, requires automated, pre-trade risk controls —credit and capital thresholds, rejection of erroneous or duplicate orders— under the broker-dealer's "direct and exclusive control"[9]; the order against Knight was its first application. In Europe, MiFID II's RTS 6 (Article 12) requires algorithmic trading firms to have "kill functionality", and ESMA spells it out: "a single decision of the investment firm should be able to result in an immediate withdrawal of all orders or any subset of them"[10]. The kill switch is not a best practice: it is a legal obligation shaped like an absorbing state.

1.3 Article thesis

1.3.1 Your bot already has a state machine: explicit (verifiable) or implicit (flags and scattered if/else); preview of the thread: IDLE → SCANNING → PENDING_ENTRY → IN_POSITION_LONG → COOLDOWN → KILL_SWITCH

The thesis —our interpretation built from the facts cited— is that every trading bot already has a state machine. The only real choice is whether it is explicit —a graph of states and transitions you can draw, test, and audit— or implicit —a constellation of boolean flags and scattered if/else whose reachable states nobody has enumerated. Knight had the second kind: Power Peg was a ghost state, the 97 emails were events with no transition, and stopping was a human procedure outside the machine.

Throughout this guide we will build the explicit version in Python: an IDLE → SCANNING → PENDING_ENTRY → IN_POSITION_LONG → COOLDOWN cycle, plus a KILL_SWITCH reachable from any state and escapable only through human reset. The first step is to stop speaking in metaphors: chapter 2 formalizes exactly what a finite-state machine is, and that formalization is what allows you to prove —not hope— that your bot cannot end up where SMARS did.

2. What an FSM is: the mathematical model, no sugarcoating

The previous chapter showed the cost of operating without explicit state. This one formalizes the alternative, anchored to the definition that guides the entire article: **a Finite State Machine (FSM) is not a framework; it is a mathematical pattern: Current State + Event → Transition Function → New State**.

2.1 Formal definition

2.1.1 The quintuple (Q, Σ, δ, q0, F); central formula: State + Event → δ → New State

A finite automaton is defined as a 5-tuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where Q is a finite set of states, Σ a finite input alphabet, $\delta : Q \times \Sigma \to Q$ the transition function, $q_0 \in Q$ the initial state, and $F \subseteq Q$ the accepting states.[11] The article's central formula is the evaluation of δ at each step:

$$q' = \delta(q, e)$$

The translation to trading is direct: Q is your strategy's states (FLAT, IN_POSITION_LONG, COOLDOWN…), Σ the events (setup_validado, fill, stop_tocado), q_0 the startup state, and δ the transition table that you write. It is the same structure with which RFC 9293 specifies TCP: 11 states, with transitions triggered by user calls, incoming segments, and timeouts.[12]

2.1.2 DFA vs NFA: determinism required in execution; completeness: partial δ = a bot that "hangs" on an unconsidered event

The only difference between a Deterministic Finite Automaton (DFA) and a nondeterministic one (NFA) is the nature of δ: in the NFA it returns a *set* of possible successors ($\delta : Q \times \Sigma \to 2^Q$); in the DFA, exactly one, so that given a state and an input sequence there is a single reached state.[13] In live execution, nondeterminism is not an option: if on (FLAT, fill) your bot could go "to LONG or to SHORT depending on the case", no risk auditor could reason about it.

The flip side is **completeness**: an automaton is deterministic if for every (p, σ) there is *at most* one successor, and complete if there is *at least* one;[14] if δ is partial, "when no transition is defined, the automaton halts".[15] A (state, event) pair with no defined transition is not an edge case that "will never happen": it is undefined behavior in production. If your bot receives a `fill` while FLAT and the table has no answer, the decision will be made by Python's exception stack. The engineering version of completeness is an exhaustive table, even if many cells resolve to "ignore and log".

2.2 Mealy vs Moore: where orders live

2.2.1 Moore (outputs on state) vs Mealy (outputs on transition); position = Moore, buy/sell order = Mealy (emitted while crossing the edge)

To act, the FSM needs outputs, and there are two classic models — George Mealy (Bell Labs, 1955) and Edward Moore (Princeton, 1956)[16] — differing only in the output function λ: in Moore, $\lambda : Q \to O$ — the output depends only on the current state; it is stable but reacts one cycle late; in Mealy, $\lambda : Q \times \Sigma \to O$ — it lives on the transition and reacts in the same cycle.[17]

The division of labor in a trading engine is asymmetric. The **net position** is a Moore output: a function exclusively of state (IN_POSITION_LONG $\Rightarrow$ +1 unit), stable and auditable — the thing you query for risk and reporting. The **buy or sell order** is a Mealy output: it is emitted *while crossing the edge* — FLAT + setup_validado → LONG *with the action* `send_order(BUY)` — because sending it one cycle later introduces exactly the latency Moore pays for its stability. Harel labeled the arrows of his statecharts with "Mealy-like outputs" (*"Mealy-like outputs, or actions"*),[18] and modern event sourcing states it cleanly: the transition returns the new state plus a list of actions as data that the environment executes afterward.[19] That separation —a pure machine that decides, effects that act— is what makes the strategy testable without a broker.

2.3 Properties you get for free

2.3.1 Determinism → replay (backtest==live); reachability (BFS); absorbing states → KILL_SWITCH with no outgoing edges except human reset

Making δ explicit buys you three properties that nested-`if` code does not have. First, **replay**: if the strategy is a pure function $\delta(\text{state}, \text{event})$ and events are journaled, the backtest is literally a replay of live;[20] when they diverge, the culprit is a specific impurity (a hidden clock, a side-effect outside δ). Second, **reachability**: the table is a graph, and a breadth-first traversal (BFS) from q_0 mechanically detects unreachable states — dead code or a forgotten edge.[21] Third, **absorbing states**: a trap state is one the automaton "can never escape",[22] and matching engines use them deliberately: an order's terminal states are absorbing by construction, "with no arrows leaving them".[23] KILL_SWITCH is exactly that: a state with no outgoing edges except a human `reset_manual`. Safety stops being a convention and becomes a theorem about the graph.

2.3.2 Minimal Mermaid diagram FLAT/LONG/SHORT + transition table as executable specification

We close with the minimal machine that will govern the examples (the six-state version arrives in chapter 3):

```
stateDiagram-v2
    [*] --> FLAT
    FLAT --> LONG : setup_validado_long / send BUY
    FLAT --> SHORT : setup_validado_short / send SELL
    LONG --> FLAT : fill_salida / send SELL
    SHORT --> FLAT : fill_salida / send BUY
```

The diagram is readable, but the executable specification is the state × event table — the same format with which the TCP RFC specifies what to do in each state for each segment:[12]

State \ Event	setup_validado_long	setup_validado_short	fill_salida	stop_tocado
FLAT	→ LONG / `send BUY`	→ SHORT / `send SELL`	— (what if it arrives?)	— (what if it arrives?)

State \ Event	setup_validado_long	setup_validado_short	fill_salida	stop_tocado
LONG	— (double entry?)	— (direct reversal?)	→ FLAT / `send SELL`	→ FLAT / `send SELL`
SHORT	— (direct reversal?)	— (double entry?)	→ FLAT / `send BUY`	→ FLAT / `send BUY`

Read it in two passes. As a designer: every cell with an arrow is a transition with its Mealy action, and the symmetric pattern of exits (closing with the operation opposite to opening) is visible at a glance. As an auditor: the cells with "—" are the pairs where δ is partial, and each one is a concrete risk question — a `fill_salida` arriving at FLAT is usually a duplicate or late fill from the broker; a validated setup while LONG raises whether you allow pyramiding or reversing without passing through FLAT. Completing the table means making those decisions in design, not in production.

3. Anatomy of a real strategy: the six states

Chapter 2 defined the machine as a quintuple and the transition table as its operational representation. Here we instantiate that formalism with a concrete, documented strategy: a 20-day Donchian channel breakout in the style of the Turtle system (System 1), whose mechanical rules —entry on a new 20-day high, re-entry filter, exit on the opposite 10-day channel— are published.[24] Being a fully mechanical system, every rule maps unambiguously to a state, an event, or a guard.

3.1 The system's state map

3.1.1 Catalog of the 6 states with their exact responsibility

Each state has a single responsibility and a closed set of events it consumes; an event that arrives at a state that does not handle it is ignored or is a bug —that is the property that makes the machine testable.

State	Exact responsibility	Events it consumes	Exit transitions
IDLE	No position and no active watch; waits for the next bar	`bar_close`, `kill`	→ SCANNING; → KILL_SWITCH
SCANNING	Evaluate whether price breaks the 20-day Donchian channel	`setup_detectado`, `bar_close`, `kill`	→ PENDING_ENTRY; → IDLE; → KILL_SWITCH
PENDING_ENTRY	Setup validated: compute sizing, set the stop, and send the order	`setup_validado`, `fill_entrada`, `bar_close`, `kill`	→ IN_POSITION_LONG; → SCANNING (rejection); → KILL_SWITCH
IN_POSITION_LONG	Manage the position: trailing stop, invalidation, time-stop	`bar_close` (internal), `stop_tocado`, `kill`	→ COOLDOWN; → KILL_SWITCH
COOLDOWN	Block re-entry for 1 bar (or N seconds) after the close	`cooldown_expirado`, `kill`	→ SCANNING; → KILL_SWITCH
KILL_SWITCH	Circuit breaker: do not trade under any market signal	`reset_manual`	→ IDLE

The table encodes three non-obvious design decisions. First: SCANNING and PENDING_ENTRY are distinct states even though both are "flat"; separating them forces the size to be computed cold, before entry, and not with the position live and PnL fluctuating. Second: IN_POSITION_LONG consumes `bar_close` as an *internal* event —it recalculates the trailing stop without changing state— while `stop_tocado` is an exit transition; section 3.3 develops this distinction, the subtlest one in the design. Third: KILL_SWITCH is the only state reachable from every other state and the only one whose exit requires a human event (`reset_manual`), never a market signal: a circuit breaker "completely disables the algorithm's ability to open new positions" until a manual audit.[25]

3.1.2 Full Mermaid diagram with transitions labeled by event and guard

The diagram is the system's contract: every edge carries its event and, in brackets, the guard. The code in chapter 4 will implement exactly these transitions, no more and no less.

```
stateDiagram-v2
    [*] --> IDLE
    IDLE --> SCANNING : bar_close
    SCANNING --> IDLE : bar_close [sin_setup]
    SCANNING --> PENDING_ENTRY : setup_detectado [ruptura_20d y precio_sobre_sma200 y
sesion_ok]
    PENDING_ENTRY --> PENDING_ENTRY : setup_validado [size > 0] / enviar_orden
    PENDING_ENTRY --> SCANNING : bar_close [size == 0 o setup_expirado]
    PENDING_ENTRY --> IN_POSITION_LONG : fill_entrada
    IN_POSITION_LONG --> IN_POSITION_LONG : bar_close [interno → recalcular_trailing]
    IN_POSITION_LONG --> COOLDOWN : stop_tocado
    IN_POSITION_LONG --> COOLDOWN : bar_close [cierre_bajo_donchian_10]
    COOLDOWN --> SCANNING : cooldown_expirado [barras_desde_salida >= 1]
    IDLE --> KILL_SWITCH : kill [drawdown_supera_umbral]
    SCANNING --> KILL_SWITCH : kill [drawdown_supera_umbral]
    PENDING_ENTRY --> KILL_SWITCH : kill [drawdown_supera_umbral]
    IN_POSITION_LONG --> KILL_SWITCH : kill [drawdown_supera_umbral]
    COOLDOWN --> KILL_SWITCH : kill [drawdown_supera_umbral]
    KILL_SWITCH --> IDLE : reset_manual [revision_humana]
```

`kill` appears five times: in a flat FSM, a global event requires one edge per source state (libraries with hierarchy let you factor it as a "from any state" transition).[26] And notice what is *not* there: there is no IDLE → IN_POSITION_LONG edge. Nobody enters a position without going through SCANNING and PENDING_ENTRY; the topology structurally forbids the shortcut.

3.2 Guards: the line between elegance and spaghetti

3.2.1 Guards as regime filters (long only if price > SMA200), session, and volatility

A guard is a boolean predicate over the context that is evaluated when the event arrives: if it fails, the transition does not happen.[27] On the SCANNING → PENDING_ENTRY edge, three canonical guards coexist: regime (long only if price is above the SMA200, a filter that does not create alpha but removes bad trades), session (do not trade outside the liquid session), and volatility (do not open if the ATR exceeds a multiple of its average). They are legitimate; the

danger lies in their abuse. Miro Samek warns that "abuse of extended state variables and guards is the primary mechanism of architectural decay": guards end up becoming the very if/else chains the machine was meant to eliminate.[28]

The design heuristic we propose (our interpretation, not Samek's): **every boolean flag a guard consults is a hidden state; two boolean flags are four hidden states**. If your transition requires `in_regime and not recently_stopped`, you are maintaining $2^2 = 4$ implicit configurations with no name and no topology. One of them —bullish regime with an active cooldown?— already has a name in your domain: promote it to an explicit state.

3.3 Internal events vs exit transitions

3.3.1 Chandelier trailing stop as an internal event vs stop, invalidation, and time-stop as exits

Inside IN_POSITION_LONG, two kinds of occurrences coexist that should not be confused. The **internal event** updates the context without changing state: this is the case of the Chandelier trailing stop, $\text{stop} = \text{HH}(22) - 3 \times \text{ATR}$, where $\text{HH}(22)$ is the highest high of the last 22 bars. Its key property is the monotonic ratchet —the stop only rises, never falls—; without it, it degenerates into a noisy band that kicks you out on every pullback.[29] Every `bar_close` recalculates the level through an internal transition that does not run the state's entry/exit actions.[30] **Exit transitions**, by contrast, destroy the position: `stop_tocado` (price reaches the level), thesis invalidation (close below the 10-day Donchian channel: the idea stops being valid even if the stop was never touched), and time-stop (the expected move does not arrive within its window). All three lead to COOLDOWN, but it is worth logging the reason: they are different diagnostics about the health of the strategy.

3.4 Sizing in PENDING_ENTRY

3.4.1 Fixed fractional (0.5–2% of equity) and Turtle N-units, with a reproducible numerical example

PENDING_ENTRY answers a single question: how much do I buy? The standard scheme is *fixed fractional*: risk a fixed fraction of equity per trade, typically between 0.5% and 2%, so that the size is derived from the distance to the stop and not the other way around.[31]

$$\text{size} = \left\lfloor \frac{ff \times \text{Equity}}{\text{trade\_risk}} \right\rfloor$$

with rounding down to the nearest whole share.[32] The Turtle variant (N-units) normalizes by volatility: a "unit" is the quantity such that a 1-ATR move equals 1% of equity, $\text{unit} = (0.01 \times \text{Equity})/\text{ATR}$.[33]

Self-contained example. Equity = 100,000 €, $ff = 1\%$, ATR(20) = 2.50 €, breakout entry at 48.00 € and initial stop at $2 \times \text{ATR}$, that is at $48.00 - 5.00 = 43.00$ €. Risk per share: 5.00 €.

- Fixed fractional: $(0.01 \times 100{,}000)/5.00 = 1{,}000/5.00 = 200$ shares. Capital tied up: $200 \times 48.00 = 9{,}600$ €; loss if the stop is hit: $200 \times 5.00 = 1{,}000$ €, exactly 1%.
- Turtle N-units: $(0.01 \times 100{,}000)/2.50 = 400$ shares. An adverse 1-ATR move costs $400 \times 2.50 = 1{,}000$ € (1%), but with the stop at 2 ATR the potential loss is 2%: it is worth checking which version of the formula is being used, as some sources include the $2 \times \text{ATR}$ in the denominator. And if the calculation came out to 0 shares, the `size > 0` guard fails and the machine returns to SCANNING: setup rejected with no order sent.

Two clarifications about the COOLDOWN that closes the cycle. The *time-based* cooldown —blocking re-entry into an asset for N candles or minutes after closing— is documented in Freqtrade as the `CooldownPeriod` protection.[34]

The original Turtle rule, by contrast, was *conditional*, not time-based: if the last 20-day breakout was a winner (whether traded or not), the next signal was skipped.[35] They are different mechanisms —one waits for a time, the other consults history— and in our FSM the second is a guard on `setup_detectado`, not a state.

4. Python implementation: from 50 lines to a professional library

Chapter 3 set the contract: six states (IDLE, SCANNING, PENDING_ENTRY, IN_POSITION_LONG, COOLDOWN, KILL_SWITCH) and an event alphabet (setup_detectado, setup_validado, fill_entrada, stop_tocado, bar_close, cooldown_expirado, kill, reset_manual). Here we implement that contract three times —by hand, with `transitions`, and with `python-statemachine`—, with a single simplification relative to the diagram: the `setup_validado` self-transition on PENDING_ENTRY is folded into `setup_detectado` to keep the examples minimal. This lets you compare what each library buys you and what price it charges. All three code listings are runnable and produce exactly the same state trajectory given the same event sequence.

4.1 Version zero: Enum + transition table

4.1.1 Complete manual implementation (~50 lines): StrEnum of states, dict (state, event)→state, InvalidTransition exception, append-only history; maximum determinism, zero dependencies

The minimal serious implementation fits on one screen: a `StrEnum` for the states, a dictionary that materializes the transition function $\delta(\text{state}, \text{event}) \to \text{state}$, an exception for the undefined pairs, and an append-only history[36]:

```
from enum import StrEnum

class Estado(StrEnum):
    IDLE = "IDLE"
    SCANNING = "SCANNING"
    PENDING_ENTRY = "PENDING_ENTRY"
    IN_POSITION_LONG = "IN_POSITION_LONG"
    COOLDOWN = "COOLDOWN"
    KILL_SWITCH = "KILL_SWITCH"

class InvalidTransition(Exception):
    """Raised when an event is not valid from the current state."""

# delta: (state, event) -> destination state
TRANSICIONES = {
    (Estado.IDLE, "bar_close"): Estado.SCANNING,
    (Estado.SCANNING, "setup_detectado"): Estado.PENDING_ENTRY,
    (Estado.PENDING_ENTRY, "fill_entrada"): Estado.IN_POSITION_LONG,
    (Estado.PENDING_ENTRY, "bar_close"): Estado.SCANNING,  # setup expires
    (Estado.IN_POSITION_LONG, "stop_tocado"): Estado.COOLDOWN,
    (Estado.COOLDOWN, "cooldown_expirado"): Estado.SCANNING,
    (Estado.KILL_SWITCH, "reset_manual"): Estado.IDLE,
```

```
}
# kill is reachable from any non-absorbing state
for _e in Estado:
    if _e is not Estado.KILL_SWITCH:
        TRANSICIONES[(_e, "kill")] = Estado.KILL_SWITCH

class EstrategiaFSM:
    def __init__(self):
        self.estado = Estado.IDLE
        self.historial = [(Estado.IDLE, None)]  # append-only: (state, event)

    def enviar(self, evento):
        clave = (self.estado, evento)
        if clave not in TRANSICIONES:
            raise InvalidTransition(f"{evento} is not valid from {self.estado}")
        self.estado = TRANSICIONES[clave]
        self.historial.append((self.estado, evento))
        return self.estado

if __name__ == "__main__":
    fsm = EstrategiaFSM()
    for evento in ["bar_close", "setup_detectado", "fill_entrada",
                   "stop_tocado", "cooldown_expirado"]:
        fsm.enviar(evento)
        print(f"evento={evento:<18} -> estado={fsm.estado}")
    try:
        fsm.enviar("fill_entrada")  # no pending order
    except InvalidTransition as exc:
        print(f"InvalidTransition: {exc}")
    fsm.enviar("kill")
    fsm.enviar("reset_manual")
    print("historial:", fsm.historial)
```

Three design decisions matter more than the line count. First: the dictionary *is* the table from chapter 3, so any unlisted (state, event) pair is illegal by construction — KILL_SWITCH only accepts `reset_manual` because the table says so, not because of a comment. Second: the error policy is *fail-fast* (`InvalidTransition`), the right choice for an execution engine where ignoring an unexpected event hides bugs. Third: the append-only history makes the strategy replayable: save the events and you reconstruct any later state, which is exactly the property that makes backtest and live run the same code. What this version does not have is guards (conditions over data, such as "price > SMA200") or entry/exit callbacks: the moment you add them as `if` statements inside `enviar`, you will be reimplementing a library, badly. That is the honest limit of the manual approach[36].

4.2 The same strategy with `transitions`

4.2.1 Runnable code with Machine: prepare/before/after callbacks, conditions as guards, may_trigger() for introspection, GraphMachine to export the diagram

`transitions` (0.9.3) separates **model** (your object, which holds the state and the data) from **machine** (the `Machine` class, which injects the triggers as methods of the model)[37]. Transitions are declared as data —lists of dicts— which fits well with generating the graph from configuration:

```
from transitions import Machine

ESTADOS = ["IDLE", "SCANNING", "PENDING_ENTRY",
           "IN_POSITION_LONG", "COOLDOWN", "KILL_SWITCH"]

TRANSICIONES = [
    {"trigger": "bar_close", "source": "IDLE", "dest": "SCANNING"},
    {"trigger": "setup_detectado", "source": "SCANNING",
     "dest": "PENDING_ENTRY", "conditions": "regimen_ok"},
    {"trigger": "fill_entrada", "source": "PENDING_ENTRY",
     "dest": "IN_POSITION_LONG", "after": "registrar_entrada"},
    {"trigger": "actualizar_trailing", "source": "IN_POSITION_LONG",
     "dest": None, "after": "recalc_stop"},          # internal transition
    {"trigger": "stop_tocado", "source": "IN_POSITION_LONG",
     "dest": "COOLDOWN", "after": "registrar_salida"},
    {"trigger": "bar_close", "source": "PENDING_ENTRY",
     "dest": "SCANNING"},                            # setup expires
    {"trigger": "cooldown_expirado", "source": "COOLDOWN", "dest": "SCANNING"},
    {"trigger": "kill", "source": "*", "dest": "KILL_SWITCH"},
    {"trigger": "reset_manual", "source": "KILL_SWITCH", "dest": "IDLE"},
]

class Estrategia:
    def __init__(self):
        self.precio_sobre_sma200 = True   # regime filter (injected data)
        self.stop = 0.0
        self.log = []

    # ---- guards: return bool, no side effects ----
    def regimen_ok(self):
        return self.precio_sobre_sma200

    # ---- action callbacks ----
    def registrar_entrada(self):
        self.stop = 95.0
        self.log.append("entry: initial stop 95.0")

    def recalc_stop(self, maximo_barra):
```

```
        self.stop = max(self.stop, maximo_barra - 3.0)  # monotonic ratchet
        self.log.append(f"trailing stop -> {self.stop}")

    def registrar_salida(self):
        self.log.append(f"stop exit at {self.stop}")

if __name__ == "__main__":
    e = Estrategia()
    Machine(e, states=ESTADOS, transitions=TRANSICIONES, initial="IDLE")

    e.bar_close()
    print("estado:", e.state)                       # SCANNING

    e.precio_sobre_sma200 = False
    print("can enter?", e.may_trigger("setup_detectado"))  # False (guard)
    e.setup_detectado()                             # fails silently
    print("state after failed guard:", e.state)     # still SCANNING

    e.precio_sobre_sma200 = True
    e.setup_detectado()
    e.fill_entrada()
    e.actualizar_trailing(maximo_barra=101.0)       # internal: no re-entry
    print("estado:", e.state, "| stop:", e.stop)
    e.actualizar_trailing(maximo_barra=99.0)        # the ratchet does not go down
    print("stop after pullback:", e.stop)
    e.stop_tocado()
    print("estado:", e.state)
    e.kill()
    print("estado:", e.state)
    e.reset_manual()
    print("estado:", e.state)
    print("log:", e.log)
```

Four things the code demonstrates. Callbacks follow a fixed order (`prepare` → `conditions` → `before` → state change → `after`), so a transition is a deterministic sequence of phases[37]. The trailing stop is modeled as an internal transition (`dest=None`): it updates data without exiting or re-entering the state, which is the exact semantics of a stop with a monotonic ratchet (it only rises, never falls)[38]. The kill switch is declared with `source="*"`, reachable from any state. And `may_trigger()` gives introspection without mutating: a risk check can ask "can I enter now?" by evaluating the guards without transitioning[39]. Operational warning: when a `condition` fails, `transitions` **fails silently** (it returns `False`, it does not raise); your tests must assert the resulting state, not just call the trigger[37].

Ecosystem bonus: with `GraphMachine` and `graph_engine="mermaid"` you export the diagram directly to the format we used in chapter 3, without installing Graphviz:

```
from transitions.extensions import GraphMachine
import io

ESTADOS = ["IDLE", "SCANNING", "PENDING_ENTRY",
           "IN_POSITION_LONG", "COOLDOWN", "KILL_SWITCH"]
TRANSICIONES = [
    {"trigger": "bar_close", "source": "IDLE", "dest": "SCANNING"},
    {"trigger": "setup_detectado", "source": "SCANNING",
     "dest": "PENDING_ENTRY", "conditions": "regimen_ok"},
    {"trigger": "fill_entrada", "source": "PENDING_ENTRY",
     "dest": "IN_POSITION_LONG"},
    {"trigger": "actualizar_trailing", "source": "IN_POSITION_LONG",
     "dest": None},
    {"trigger": "stop_tocado", "source": "IN_POSITION_LONG", "dest": "COOLDOWN"},
    {"trigger": "bar_close", "source": "PENDING_ENTRY", "dest": "SCANNING"},
    {"trigger": "cooldown_expirado", "source": "COOLDOWN", "dest": "SCANNING"},
    {"trigger": "kill", "source": "*", "dest": "KILL_SWITCH"},
    {"trigger": "reset_manual", "source": "KILL_SWITCH", "dest": "IDLE"},
]

class Estrategia:
    def regimen_ok(self):     # regime guard (always True in the demo)
        return True

if __name__ == "__main__":
    e = Estrategia()
    GraphMachine(e, states=ESTADOS, transitions=TRANSICIONES,
                 initial="IDLE", graph_engine="mermaid", show_conditions=True)
    buf = io.BytesIO()
    e.get_graph().draw(buf)   # writes the diagram in mermaid format
    print(buf.getvalue().decode())
```

The output is a `stateDiagram-v2` block that includes the guards as labels (`setup_detectado [regimen_ok]`) and marks `actualizar_trailing` as `[internal]` — living documentation generated from the code itself[37].

4.3 The same strategy with `python-statemachine`

4.3.1 Declarative runnable code: cond/unless guards with priority by declaration order, internal transition (internal=True) for the trailing stop, KILL_SWITCH accessible from any state (from_.any() or equivalent pattern), enabled_events() for introspection; mentions StateChart v3 (compound states) as a preview

`python-statemachine` (3.2.0, requires Python ≥3.10) adopts the opposite style: the machine *is* a class and the transitions are declarative, self-documented attributes[40]:

```
from statemachine import StateMachine, State

class EstrategiaSM(StateMachine):
    IDLE = State(initial=True)
    SCANNING = State()
    PENDING_ENTRY = State()
    IN_POSITION_LONG = State()
    COOLDOWN = State()
    KILL_SWITCH = State()

    bar_close = IDLE.to(SCANNING) | PENDING_ENTRY.to(SCANNING)
    setup_detectado = SCANNING.to(PENDING_ENTRY, cond="regimen_ok")
    fill_entrada = PENDING_ENTRY.to(IN_POSITION_LONG)
    actualizar_trailing = IN_POSITION_LONG.to.itself(
        internal=True, on="recalc_stop")          # internal transition
    stop_tocado = IN_POSITION_LONG.to(COOLDOWN)
    cooldown_expirado = COOLDOWN.to(SCANNING)
    kill = KILL_SWITCH.from_.any()                # from any state
    reset_manual = KILL_SWITCH.to(IDLE)

    def __init__(self):
        self.precio_sobre_sma200 = True
        self.stop = 0.0
        self.log = []
        super().__init__()

    # guard: no side effects; if it fails, the transition does not run
    def regimen_ok(self):
        return self.precio_sobre_sma200

    def on_fill_entrada(self):
        self.stop = 95.0
        self.log.append("entry: initial stop 95.0")

    def recalc_stop(self, maximo_barra):
        self.stop = max(self.stop, maximo_barra - 3.0)  # monotonic ratchet
        self.log.append(f"trailing stop -> {self.stop}")

    def on_stop_tocado(self):
        self.log.append(f"stop exit at {self.stop}")

def estado(sm):
    return next(iter(sm.configuration_values))
```

```
if __name__ == "__main__":
    sm = EstrategiaSM()
    sm.send("bar_close")
    print("estado:", estado(sm))                          # SCANNING

    sm.precio_sobre_sma200 = False
    print("enabled without regime:",
          [e.id for e in sm.enabled_events()])            # only kill
    sm.precio_sobre_sma200 = True
    print("enabled with regime:",
          [e.id for e in sm.enabled_events()])            # setup_detectado, kill

    sm.send("setup_detectado")
    sm.send("fill_entrada")
    sm.send("actualizar_trailing", maximo_barra=101.0)  # internal: no exit/enter
    print("estado:", estado(sm), "| stop:", sm.stop)
    sm.send("stop_tocado")
    print("estado:", estado(sm))
    sm.send("kill")                                       # valid from COOLDOWN
    print("estado:", estado(sm),
          "| KILL_SWITCH active:", sm.KILL_SWITCH.is_active)
    sm.send("reset_manual")
    print("estado:", estado(sm))
    print("log:", sm.log)
```

Three semantic differences from `transitions` that change how you test. First: the default policy is strict — an event with no enabled transition (or whose guard fails) raises `TransitionNotAllowed`, the fail-fast behavior we want in execution[41]. Second: when several transitions share an event, the engine evaluates them **in declaration order and the first one whose guard passes wins**, a deterministic semantics ideal for encoding risk priorities (the kill switch first)[41]. Third: introspection is `enabled_events()`, which returns the events with satisfied guards *right now* — note in the demo how `setup_detectado` disappears when the regime filter is turned off while `kill` always stays enabled[41]. The `KILL_SWITCH.from_.any()` pattern creates the global transition from every non-final state in a single line[42]. Finally, a preview: since v3.0 the library includes `StateChart`, with compound, parallel, and history states under SCXML semantics (a W3C standard); if your FSM grows toward substates (e.g. IN_POSITION_LONG with pyramiding phases), that is the native path[43].

4.4 Which one to choose

4.4.1 Comparison table: transitions 0.9.3 (6.6k★) vs python-statemachine 3.2.0 (1.3k★, StateCharts SCXML, Py≥3.10) vs manual; criteria: determinism, testability, state serialization, async, introspection; warning about transitions' AsyncMachine (callbacks with asyncio.gather, issue #717) for live engines

Criterion	Manual (Enum + dict)	transitions 0.9.3	python-statemachine 3.2.0
Style	Data table + custom loop	Data-driven: dicts + model	Declarative: class + `a.to(b)`
Invalid event	`InvalidTransition` (your call)	Fails silently by default	`TransitionNotAllowed` (strict)
Guards	Manual	`conditions`/`unless`	`cond`/`unless`, priority by declaration order
Internal transition	Separate logic	`dest=None`	`to.itself(internal=True)`
Introspection	Direct `TRANSICIONES[estado]`	`may_trigger()` / `may_<trigger>()`[39]	`enabled_events()`[41]
State serialization	Trivial (the state is a string)	`model.state`; pickleable machine[37]	`configuration_values`; definition exportable to YAML/SCXML/JSON[43]
Async	By hand	`AsyncMachine` (separate class)	Autodetected async engine, sequential callbacks
Hierarchy	No	`HierarchicalMachine` (extension)	`StateChart`: compound, parallel, history (SCXML)[43]
Project health	No dependency	6.6k★, release 2025-07, slow cadence	1.3k★, v3.2.0 (2026-06), very active, Py ≥3.10[40]

Our recommendation by use case. For deterministic backtesting and for learning, the manual version or `transitions`: the transition table is inspectable at a glance, the state serializes as a string, and `may_trigger()` makes it easy to write tests that mutate nothing. For a live engine with asyncio, hierarchy, and auditability, `python-statemachine` 3.2: its async engine runs callbacks sequentially inside the run-to-completion model, whereas the `AsyncMachine` of `transitions` launches the callbacks of a given phase with `asyncio.gather` — concurrent execution that breaks ordering assumptions when managing resources, an open problem in its issue #717 that forces you to subclass if two callbacks must run in order (record fill → update PnL)[44]. On top of that, the SCXML semantics with W3C conformance tests and the YAML-loadable definition make the graph auditable as data, not just as code[43]. The manual option remains the gold standard for verifying by replay that both libraries implement the same contract: same events, same history.

With the strategy FSM implemented and verified, the system's other state machine is still pending: the one for each order's lifecycle, which is where real execution meets the broker. That is the subject of chapter 5.

5. The second machine: the order lifecycle

The strategy FSM from chapter 4 decides *when* to act. But calling `buy()` brings a second machine into play, independent of the first: every live order has its own state, events, and transitions. You don't have to invent this machine: the FIX protocol (Financial Information eXchange) standardized it in 1992, and every modern API — Binance, Alpaca, Interactive Brokers (IBKR)— is a dialect of the same graph. Your job is not to design the alphabet, but to make it explicit in your Order Management System (OMS) instead of suffering it implicitly.

5.1 The state alphabet of an order

5.1.1 FIX OrdStatus (tag 39)

FIX defines the state of an order as a single character in tag 39, `OrdStatus`: `0` = New, `1` = PartiallyFilled, `2` = Filled, `4` = Canceled, `8` = Rejected, `A` = PendingNew, `6` = PendingCancel, `E` = PendingReplace, among others.[45] The specification includes an official transition matrix (Appendix D): rows = current state, columns = next reported state, and each state carries a precedence value that decides what to report when the order is in several states at once (partially filled *and* cancel-pending).[46] Two details are design lessons. First: the `Canceled` and `Rejected` rows are empty —absorbing states. Second: `Replaced` existed as a state (`5`) in FIX 4.2, but it was removed in FIX 4.4 and today appears as "No longer used":[47] the standard corrected its own FSM, demoting "replaced" from a state to an *event* (`ExecType=5`); after a replace, the order returns to `New` or `PartiallyFilled`.

```
stateDiagram-v2
    [*] --> PendingNew : NewOrderSingle
    PendingNew --> New : ACK venue (ExecType=0)
    PendingNew --> Rejected : rejection (ExecType=8)
    New --> PartiallyFilled : Trade, CumQty < OrderQty
    New --> Filled : Trade, CumQty = OrderQty
    PartiallyFilled --> PartiallyFilled : partial Trade
    PartiallyFilled --> Filled : Trade, CumQty = OrderQty
    New --> PendingCancel : CancelRequest
    PartiallyFilled --> PendingCancel : CancelRequest
    PendingCancel --> Canceled : cancel accepted
    PendingCancel --> Filled : in-flight fill (too late to cancel)
    PendingCancel --> PartiallyFilled : partial in-flight fill
    Filled --> [*]
    Canceled --> [*]
    Rejected --> [*]
```

The diagram simplifies the core of the FIX matrix (it omits `PendingReplace`, `Expired`, `DoneForDay`). Note the two race edges leaving `PendingCancel`: we will come back to them in 5.3.

5.2 Events ≠ states: the classic trap

5.2.1 ExecType (tag 150) vs OrdStatus (tag 39)

The entire lifecycle arrives through a single message, the `ExecutionReport` (MsgType `8`), with two fields that novice implementations confuse: `ExecType` (tag 150) describes *what just happened* —the event— while `OrdStatus` *always* identifies the current state —the destination node.[48] In automata vocabulary: `ExecType` is the input alphabet; `OrdStatus`, the set of states. The reference documentation is blunt: "This is where most implementations go wrong… A common mistake is treating them as the same field".[49]

The distinction is not pedantry: the same event `F` = Trade leads to `PartiallyFilled` or to `Filled` depending on the guard that compares `CumQty` (cumulative quantity, tag 14) with `OrderQty` (tag 38). The event alone does not determine the transition. Binance replicates this exact duality: every `executionReport` on its user data stream carries `x` (execution type) and `X` (order status).[50]

5.3 The cases that break bots

5.3.1 Races, zombies, and duplicates

The cancel-vs-fill race. An order can execute while your cancellation is in flight: FIX documents this in matrices D4/D5 of Appendix D, where the fill and the `CancelRequest` "cross on the wire" and the broker responds `CancelReject` with `CxlRejReason = 0` —"too late to cancel".[46] Consequence: after sending a cancel, the order is in `PendingCancel`, not `Canceled`, and the fills in that window are real. Alpaca draws these edges in its official diagram ("Original Order filled before cancel").[51]

Duplicate fills. Session recovery (WebSocket replay, `PossResend` in FIX) can deliver the same fill twice. The standard defense is idempotency by execution ID (`ExecID`, tag 17; `trade_id` in crypto). Nautilus Trader enforces it as a hard invariant: "This is the invariant that prevents double-counting executions".[52]

The zombie order. If the connection dies before the ACK, the order is left in an unknown state: it may never have arrived, it may be alive, or it may have executed. Cboe cancels open orders after two heartbeats without messages from the client, "to prevent orders from being stuck in an unknown state";[53] and MIAX warns that its cancel-on-disconnect is *best effort*: "Executions can occur while FOI is processing the ACOD event".[54] Without an honest in-flight state and reconciliation on reconnect, this case is unmanageable.

5.4 How the frameworks do it

5.4.1 From the explicit FSM to free-form strings

Framework	State alphabet	FSM	Transition verification
Nautilus Trader	`OrderStatus`, 14 states (Rust core)	Explicit + event sourcing	Yes: `Err(InvalidStateTransition)`
Hummingbot	`OrderState`, 11 states	Semi (enum + `ClientOrderTracker`)	Partial (`is_open`/`is_done` predicates)
Backtrader	9 integer constants	Implicit (lives in the simulated broker)	No
Freqtrade	Free-form strings (CCXT mirror) + `ft_is_open` flag	Implicit	No

The table orders the frameworks by increasing rigor from bottom to top. Nautilus Trader encodes the transition as a pure `match (state, event) -> Result` function in Rust: any unenumerated pair returns `InvalidStateTransition`, the order is rebuilt by replaying its event stream, and it even documents counterintuitive edges such as `Canceled → Filled` ("Real world possibility").[55] Hummingbot defines an enum with terminal states (`CANCELED`, `FILLED`, `FAILED`) separated from the active ones, but with no central verifier.[56] Backtrader spreads nine states across the broker and the strategy reacts in `notify_order`.[57] Freqtrade doesn't even have an enum: `Order.status` is a free-form string inherited from CCXT plus a boolean `ft_is_open`.[58] The cost of the implicit FSM is measurable: Hummingbot issue #7294 documents a spurious `MarketOrderFailureEvent` that triggered retries while the original order was in fact executing —duplicate orders caused by not modeling the in-flight state.[59] The mapping between venues is almost isomorphic: IBKR's `Submitted` covers `New` and (with `filled > 0`) `PartiallyFilled`; Alpaca and Binance name the states just like FIX; IBKR adds "pre-venue" states (`PendingSubmit`, `PreSubmitted`) that FIX collapses into `PendingNew`.[60]

And one closing structural observation: `Canceled` and `Rejected` are absorbing by construction in the FIX matrix —the same pattern as the strategy-level `KILL_SWITCH` that the next chapter exploits.

6. KILL_SWITCH and COOLDOWN: risk as topology

In chapter 2 we defined absorbing states as vertices with no outgoing edges. Here we give them their decisive use: risk control is not a feature you add to the strategy, it is a property of the graph's topology. Either your machine has a vertex you cannot leave without human permission, or your "kill switch" is a convention that the code can violate.

6.1 The kill switch as an absorbing state by construction

6.1.1 No outgoing edges except human reset

A well-built kill switch satisfies two provable properties over the graph: from any operational state there is a `kill` transition into KILL_SWITCH (reachability), and from KILL_SWITCH there is only one edge, `reset_manual`, which demands human intervention (absorption). Entry triggers an atomic sequence —stop accepting signals → cancel all live orders → liquidate positions if policy requires it → halt— that no one can interrupt halfway. MiFID II (RTS 6, Art. 12) demands exactly this —the immediate withdrawal of all orders, or of any subset, through a single decision, as we saw in chapter 1—.[61]

The contrast with Knight Capital is not the timeline already told, it is the architecture. The SEC order acknowledges in its footnote 7 that Knight did have automatic shutdown for *certain strategies of one of its groups*: the control existed, but it did not cover every order-routing path — and SMARS was left outside.[62] A partial kill switch is topologically equivalent to none for the uncovered paths. And its main risk tool, PMON, was a post-execution monitor that "relied entirely on human monitoring and did not generate automated alerts":[62] the halt lived outside the machine, in a person staring at a screen. A scattered `if` in your code has the same defect: it is not a state, and you cannot prove that nobody trades while it is active.

6.2 Triggers: the taxonomy of what must stop you

6.2.1 Which events should fire `kill`

Triggers are not invented; industrial practice and the regulatory record have cataloged them:

Trigger	What it detects	Documented precedent
Daily loss / drawdown	Slow ruin: the strategy or the market turned against you	Freqtrade's `MaxDrawdown`: "If the observed drawdown exceeds `max_allowed_drawdown`, trading will stop for `stop_duration`"[63]
Anomalous order rate	Emission loop: orders/second or cancel/fill ratio out of range	The exact Knight symptom: 212 parent orders → >4 million fills with no control comparing output to input[62]
Price deviation	Corrupt data or fat finger: price vs. mid, last, or external feed	Citigroup 2022: a $58M basket was entered as $444bn and $1.4bn actually got executed[64]
Rejection rate / unexpected fills	Desynchronization with the venue: your model of the book does not match reality	Knight's 97 "BNET rejects" emails were events emitted with no handler[62]

Trigger	What it detects	Documented precedent
Latency / connectivity	Slow ACKs, lost heartbeat, stale data: "stale data is more dangerous than no data"[65]	CME's Cancel on Disconnect: the venue cancels your resting orders if your session drops[66]
Logical integrity	NaN, negative prices, broken internal invariants	Citi's hard blocks stopped $248bn; the 711 *dismissible* warnings did not — total fine of £61.6M[64]

The table matters less for its content than for how you read it: each row is an event in your FSM's alphabet with a defined transition into KILL_SWITCH or COOLDOWN. The Citigroup lesson is that a warning that does not transition is not a control: 646 soft blocks appeared in a pop-up of which only 18 lines were visible, and the trader could close it and carry on.[64] The industrial reference is CME Group's Kill Switch: it blocks all new order entry and cancels every live order in under a second.[6667] And Rule 15c3-5 —under which Knight paid $12M in the first enforcement action— requires capital thresholds *linked* to automated pre-trade controls, measured over orders entered, not executed.[68] Translation into your graph: the guard lives in the transition function, before emitting, not in an after-the-fact report.

6.3 COOLDOWN as a state with a timed exit

6.3.1 Why it is a state and not a loose timestamp

COOLDOWN is a KILL_SWITCH with a timed exit: same mechanism (structural ban on re-entering), different outgoing edge (`cooldown_expirado` instead of `reset_manual`). The programmer's temptation is an `if time.now() > last_exit + delta` scattered through the code; the objection is that this loose timestamp is not testable as an invariant, does not appear in the diagram, and nothing prevents another path from ignoring it. Modeled as a state, the ban is a property of the graph: from COOLDOWN there is no edge into PENDING_ENTRY until the expiration event arrives.

Freqtrade implements it literally this way: `StoplossGuard` returns `ProtectionReturn(lock=True, until=until, reason=...)` — a lock state with its unlock timestamp embedded, "assuming that the bot needs some time to let markets recover".[6369] It is the honest version of an uncomfortable principle: after a losing streak your model of the market is less reliable, and the machine must impose on you the pause you would not impose on yourself.

With this, the piece that chapter 7 will exploit is in place: "no trading from KILL_SWITCH" and "no re-entry from COOLDOWN until `cooldown_expirado`" are not design aspirations, they are invariants over a finite graph — and an invariant over a finite graph is exhaustively testable.

7. Testing: the transition table is your test suite

Chapters 3 and 4 ended with an explicit machine: six states and a `TRANSICIONES` table that decides, for each (state, event) pair, whether there is a transition and where it goes. This chapter exploits the practical consequence of that finiteness: the behavior space is enumerable, and exhaustive coverage stops being a luxury and becomes the reasonable standard.

7.1 State and transition coverage

7.1.1 State coverage and transition coverage (0-switch/1-switch per ISTQB): a measurable coverage target, achievable at 100%

Over the FSM graph, the model-based testing literature defines gradual criteria: *state coverage* (every state visited at least once), *transition coverage* or **0-switch** (every transition exercised at least once), **1-switch** (every pair of consecutive transitions) and N-switch (every sequence of N+1 transitions).[70] The ISTQB CTAL-TA syllabus points out that 0-switch and 1-switch are the criteria commonly used in practice, and that 2-switch or higher is only justified under high risk, since the number of N-switches grows exponentially with N.[71] Terminological warning: some popular sources call state coverage 0-switch; here we follow the ISTQB convention, where 0-switch equals valid-transition coverage.[71]

The translation into your strategy is direct: with 6 states and 7 transitions plus `kill` and `reset_manual`, 100% 0-switch requires exercising exactly 9 edges, and a single 11-event script covers them all (`test_0_switch_todas_las_transiciones_ejercidas`, in the next block). ISTQB adds the *round-trip coverage* criterion —cycles that start and end in the same state—, very effective at detecting defects;[71] in trading, a round trip is literally `IDLE → IN_POSITION_LONG → IDLE`: one complete trade.

7.2 Invalid-transition tests

7.2.1 Negative state×event matrix: assert exception + no mutation + no side effects; runnable pytest code

Absent transitions are as important as present ones: every (state, event) pair without an edge is a promise of rejection. The correct negative test verifies three things: expected exception, unchanged state, and no side effects (position, stop, orders, and journal intact).[72] Since the table is data, the complete negative matrix —6 states × 8 events = 48 pairs, 35 illegal— is generated by parametrization, not by hand. The next block (pytest 8.x, Python 3.11+) redefines the minimal FSM to be self-contained and includes the 0-switch test and the replay test from section 7.4:

```
import hashlib
import itertools
import json

import pytest

IDLE, SCANNING, PENDING_ENTRY = "IDLE", "SCANNING", "PENDING_ENTRY"
IN_POSITION_LONG, COOLDOWN, KILL_SWITCH = "IN_POSITION_LONG", "COOLDOWN", "KILL_SWITCH"
ESTADOS = (IDLE, SCANNING, PENDING_ENTRY, IN_POSITION_LONG, COOLDOWN, KILL_SWITCH)
EVENTOS = ("setup_detectado", "setup_validado", "fill", "stop_tocado",
           "bar_close", "cooldown_expirado", "kill", "reset_manual")

TRANSICIONES = {
    (IDLE, "setup_detectado"): SCANNING,
    (SCANNING, "setup_validado"): PENDING_ENTRY,
    (SCANNING, "bar_close"): IDLE,
    (PENDING_ENTRY, "fill"): IN_POSITION_LONG,
    (PENDING_ENTRY, "bar_close"): SCANNING,
```

```
    (IN_POSITION_LONG, "stop_tocado"): COOLDOWN,
    (COOLDOWN, "cooldown_expirado"): IDLE,
    (KILL_SWITCH, "reset_manual"): IDLE,
}


class TransicionInvalida(Exception):
    pass


class EstrategiaFSM:
    """Minimal version of the FSM from chapters 3-4: table + extended state."""

    def __init__(self, estado=IDLE):
        self.estado = estado
        self.stop_price = None
        self.posicion_neta = 0
        self.ordenes_abiertas = 0
        self.journal = []

    def handle(self, evento, **payload):
        if evento == "kill" and self.estado != KILL_SWITCH:
            self.estado, self.ordenes_abiertas = KILL_SWITCH, 0  # cancels whatever is pending
            self.journal.append(evento)
            return
        destino = TRANSICIONES.get((self.estado, evento))
        if destino is None:
            raise TransicionInvalida(f"{evento} is not valid from {self.estado}")
        if evento == "setup_validado":
            self.ordenes_abiertas = 1
        elif evento == "fill":
            self.ordenes_abiertas = 0
            self.posicion_neta += payload["qty"]
            self.stop_price = payload["stop"]
        elif evento == "stop_tocado":
            self.posicion_neta, self.stop_price = 0, None
        elif self.estado == PENDING_ENTRY and evento == "bar_close":
            self.ordenes_abiertas = 0
        elif evento == "reset_manual":
            assert self.posicion_neta == 0, "reconcile the position before resetting"
        self.estado = destino
        self.journal.append(evento)


def pares_invalidos():
```

```
    validos = set(TRANSICIONES) | {(s, "kill") for s in ESTADOS if s != KILL_SWITCH}
    return [(s, e) for s, e in itertools.product(ESTADOS, EVENTOS) if (s, e) not in validos]


@pytest.mark.parametrize("estado,evento", pares_invalidos())
def test_transicion_invalida_no_muta_nada(estado, evento):
    fsm = EstrategiaFSM(estado=estado)
    fsm.posicion_neta, fsm.stop_price, fsm.ordenes_abiertas = 10, 95.0, 1
    antes = (fsm.estado, fsm.posicion_neta, fsm.stop_price,
             fsm.ordenes_abiertas, len(fsm.journal))
    with pytest.raises(TransicionInvalida):
        fsm.handle(evento, qty=10, stop=95.0)
    despues = (fsm.estado, fsm.posicion_neta, fsm.stop_price,
               fsm.ordenes_abiertas, len(fsm.journal))
    assert antes == despues


def test_0_switch_todas_las_transiciones_ejercidas():
    fsm = EstrategiaFSM()
    guion = ["setup_detectado", "setup_validado", "bar_close", "setup_validado",
             "fill", "stop_tocado", "cooldown_expirado", "setup_detectado",
             "bar_close", "kill", "reset_manual"]
    ejercidas = set()
    for evento in guion:
        origen = fsm.estado
        fsm.handle(evento, qty=10, stop=95.0)
        ejercidas.add((origen, evento))
    assert set(TRANSICIONES) <= ejercidas


def replay(log):
    fsm = EstrategiaFSM()
    for evento in log:
        fsm.handle(evento["tipo"], **evento.get("payload", {}))
    return fsm


def estado_hash(fsm):
    payload = json.dumps({"estado": fsm.estado, "posicion": fsm.posicion_neta,
                          "stop": fsm.stop_price, "ordenes": fsm.ordenes_abiertas},
                         sort_keys=True).encode()
    return hashlib.sha256(payload).hexdigest()


def test_replay_determinism_same_events_same_hash():
```

```
    log = [{"tipo": "setup_detectado"}, {"tipo": "setup_validado"},
           {"tipo": "fill", "payload": {"qty": 10, "stop": 95.0}},
           {"tipo": "stop_tocado"}, {"tipo": "cooldown_expirado"},
           {"tipo": "kill"}, {"tipo": "reset_manual"}]
    assert estado_hash(replay(log)) == estado_hash(replay(log))
```

Three design decisions matter here. First: the constructor accepts the initial state, so each test seeds its starting point instead of navigating to it; a previously broken transition does not contaminate the test under examination.[72] Second: the negative test asserts over all observable state, including the journal length —a rejected event that left a trace in the audit log would be a defect—. Third: `pares_invalidos()` is derived from the table itself, so adding a transition to the model updates the suite automatically: the table is literally the executable specification of the tests.

7.3 Invariants and property-based testing

7.3.1 Invariant catalog: "never IN_POSITION_LONG without a stop", "no trading from KILL_SWITCH", "net position consistent with fills"; Hypothesis RuleBasedStateMachine with @invariant and rules; runnable code

The previous tests verify transitions one by one; invariants verify properties that must hold after *every* step, in any sequence. Minimal catalog for this FSM, derived from the documented practice of OMSs and kill switches:[73][74]

- **Mandatory stop:** `IN_POSITION_LONG` implies `stop_price is not None`.
- **KILL_SWITCH is an operational sink:** in `KILL_SWITCH` there are no open orders, and the only exit is `reset_manual` after reconciling the position.
- **Position–fills consistency:** the net position matches the sum of signed fills recomputed from the journal; in flat states it is zero.

Hypothesis's stateful testing (documentation 6.161.5) generates sequences of operations, not just data: you declare rules (`@rule`) with arguments drawn from `st.*` strategies, and the engine searches for chains of rules that violate some `@invariant`, shrinking them to the minimal program that reproduces the failure.[75] `@precondition` filters out inapplicable rules before running them,[75] and since version 6.7.0 invariants are only checked after the `@initialize` steps.[76] The model-based pattern consists of keeping, alongside the system, a deliberately simple reference implementation (here, the `modelo_posicion` counter) and asserting that the two agree:[75]

```
import hypothesis.strategies as st
from hypothesis import settings
from hypothesis.stateful import (
    RuleBasedStateMachine, invariant, precondition, rule,
)

IDLE, SCANNING, PENDING_ENTRY = "IDLE", "SCANNING", "PENDING_ENTRY"
IN_POSITION_LONG, COOLDOWN, KILL_SWITCH = "IN_POSITION_LONG", "COOLDOWN", "KILL_SWITCH"

TRANSICIONES = {
    (IDLE, "setup_detectado"): SCANNING,
    (SCANNING, "setup_validado"): PENDING_ENTRY,
```

```
    (SCANNING, "bar_close"): IDLE,
    (PENDING_ENTRY, "fill"): IN_POSITION_LONG,
    (PENDING_ENTRY, "bar_close"): SCANNING,
    (IN_POSITION_LONG, "stop_tocado"): COOLDOWN,
    (COOLDOWN, "cooldown_expirado"): IDLE,
    (KILL_SWITCH, "reset_manual"): IDLE,
}


class TransicionInvalida(Exception):
    pass


class EstrategiaFSM:  # same machine as the previous block, repeated for self-containment
    def __init__(self, estado=IDLE):
        self.estado = estado
        self.stop_price = None
        self.posicion_neta = 0
        self.ordenes_abiertas = 0
        self.journal = []

    def handle(self, evento, **payload):
        if evento == "kill" and self.estado != KILL_SWITCH:
            self.estado, self.ordenes_abiertas = KILL_SWITCH, 0
            self.journal.append(evento)
            return
        destino = TRANSICIONES.get((self.estado, evento))
        if destino is None:
            raise TransicionInvalida(f"{evento} is not valid from {self.estado}")
        if evento == "setup_validado":
            self.ordenes_abiertas = 1
        elif evento == "fill":
            self.ordenes_abiertas = 0
            self.posicion_neta += payload["qty"]
            self.stop_price = payload["stop"]
        elif evento == "stop_tocado":
            self.posicion_neta, self.stop_price = 0, None
        elif self.estado == PENDING_ENTRY and evento == "bar_close":
            self.ordenes_abiertas = 0
        elif evento == "reset_manual":
            assert self.posicion_neta == 0, "reconcile the position before resetting"
        self.estado = destino
        self.journal.append(evento)
```

```
class EstrategiaMachine(RuleBasedStateMachine):
    """Hypothesis generates event sequences; the invariants are checked after each step."""

    def __init__(self):
        super().__init__()
        self.fsm = EstrategiaFSM()
        self.modelo_posicion = 0  # reference implementation (model-based)

    def _intentar(self, evento, **payload):
        try:
            self.fsm.handle(evento, **payload)
            return True
        except TransicionInvalida:
            return False  # illegal event in this state: rejection is already the contract

    @rule()
    def detectar(self):
        self._intentar("setup_detectado")

    @rule()
    def validar(self):
        self._intentar("setup_validado")

    @rule()
    def vela(self):
        self._intentar("bar_close")

    @rule()
    def enfriar(self):
        self._intentar("cooldown_expirado")

    @rule(qty=st.integers(min_value=1, max_value=100),
          stop=st.floats(min_value=1.0, max_value=100000.0))
    def llenar(self, qty, stop):
        if self._intentar("fill", qty=qty, stop=stop):
            self.modelo_posicion += qty

    @rule()
    def tocar_stop(self):
        if self._intentar("stop_tocado"):
            self.modelo_posicion = 0

    @rule()
    def matar(self):
        self._intentar("kill")
```

```
    @precondition(lambda self: self.fsm.estado == KILL_SWITCH
                  and self.fsm.posicion_neta == 0)
    @rule()
    def resetear(self):
        self.fsm.handle("reset_manual")

    @invariant()
    def nunca_en_posicion_sin_stop(self):
        if self.fsm.estado == IN_POSITION_LONG:
            assert self.fsm.stop_price is not None

    @invariant()
    def desde_kill_switch_no_se_opera(self):
        if self.fsm.estado == KILL_SWITCH:
            assert self.fsm.ordenes_abiertas == 0

    @invariant()
    def posicion_coherente_con_fills(self):
        assert self.fsm.posicion_neta == self.modelo_posicion
        if self.fsm.estado in (IDLE, SCANNING, PENDING_ENTRY, COOLDOWN):
            assert self.fsm.posicion_neta == 0


TestEstrategia = EstrategiaMachine.TestCase
TestEstrategia.settings = settings(max_examples=30, stateful_step_count=50,
                                   derandomize=True, deadline=None)
```

Note that the rules do not avoid illegal events: they fire them on purpose and treat rejection as part of the contract, so Hypothesis explores hostile interleavings —a `kill` mid-entry, a late `fill`, a double `stop_tocado`— that no manual script would have combined. The `resetear` precondition encodes a real business rule: the machine is not re-armed with an open, unreconciled position. `derandomize=True` pins the seed so CI is reproducible. We verified this code by mutation: if `kill` stops canceling orders, or if `fill` fails to record the stop, the corresponding invariants fail —the suite bites.

7.4 Replay: determinism as the golden test

7.4.1 replay_determinism_same_events_same_hash: event journal + pure FSM = backtest and live are the same program; divergence = impurities (hidden clock, side effects outside δ)

The last test closes the argument. If `handle()` is a pure transition function δ —its result depends only on the internal state and the event— and every event lands in the journal, re-running the journal is re-running the strategy. The pattern, used in production event-sourced trading engines, is literally named `replay_determinism_same_events_same_hash`: processing the same log twice through the full engine must produce an identical state hash.[77] It is the test included in the first block.

The implication is strong: a backtest is a replay of the journal over historical data, and live is the same program fed by market events; reusing the same strategy and risk code in both is precisely the advantage that the event-driven literature attributes to this design.[78] The reverse reading works as a diagnostic: if backtest and live diverge, the replay test turns that mystery into a bounded symptom —there is impurity in δ: a real clock consulted inside `handle()`, an unseeded `random`, a network call, a side effect outside the transition function—. Determinism is not declared; it is tested. And this serialized journal is the direct ancestor of the checkpointers that will appear in chapter 8 when we cross the bridge toward LangGraph.

8. When the FSM falls short: from states to graphs (bridge to LangGraph)

Throughout this guide we built a six-state strategy with an absorbing `KILL_SWITCH`, auditable journaling, and property tests. That machine is correct, deterministic, and verifiable. It also has a ceiling: this chapter delimits where that ceiling is, what the classic answers are, and why the next step —LangGraph's state graphs— is not a paradigm leap but a generalization of what you already know.

8.1 The three cracks of the flat FSM

8.1.1 State explosion, parallelism, and memory

The first crack is arithmetic. Every binary dimension you add to the behavior (trailing active? 50 % partial executed? break-even moved?) multiplies the state space: with n flags there are up to 2^n configurations, and with N independent positions of k states each, k^N global states. In the formal verification literature this has a name of its own: the *state explosion problem*, "the central problem of model checking" according to the reference text by Clarke, Grumberg and Peled.[79] With 6 exit-management flags that is already 64 flat states; with 10 symbols, more than 10^{18} configurations. And it is not only the states that explode: in a flat FSM transitions tend to grow exponentially along with them, as David Khourshid, creator of XState, points out.[80]

The second crack is structural: a classic FSM is in **a single state at a time**. Managing trailing-stop *and* take-profit *and* partials simultaneously demands the Cartesian product of all of them as compound states. The third is the absence of memory: if a position under management is interrupted (market halt, broker disconnection) and must resume in the exact substate where it was, the flat FSM forces you to encode a "last substate" variable with guards — covert memory in flags, precisely the antipattern Samek identifies as "the primary mechanism of architectural decay" in state machines.[81]

8.2 Statecharts: the classic answer

8.2.1 Harel 1987 and its current availability in Python

David Harel diagnosed these shortcomings in 1987 and defined statecharts as "state-diagrams + depth + orthogonality + broadcast-communication".[82] Hierarchy factors out common behavior: your `KILL_SWITCH` is defined **once** in the parent state and inherited by every substate of `IN_POSITION`, instead of repeating the transition in each one (forgetting one repetition is a real-risk bug). Orthogonality introduces parallel regions — one for stops, another for partials, another for the time-stop — that avoid the Cartesian product. *History states* solve resumption: a

compound state "remembers" which substate was active when it was exited.[83] And the extended state formalizes what we already did in this guide: the mode is qualitative (`TRAILING`), the stop level is quantitative (context, not state).

This is not museum theory: python-statemachine 3.2.0 supports StateCharts SCXML with compound, parallel, and history states, and transitions offers `HierarchicalMachine`.[84] If your problem is state explosion *within* a deterministic strategy, the answer is a statechart, not an agent graph.

8.3 The conceptual leap: an FSM is a labeled directed graph

8.3.1 FSM ⊂ graphs: three constraints that LangGraph relaxes

Formally, a finite automaton **is** a labeled directed graph: the vertices are the states and the edges have the form $q \to \delta(q, x)$.[85] What we call an FSM is a graph with three constraints: the transition function δ is a fixed table defined at design time (Samek: the topology must be "static and fixed at compile time"[81]), the edges are fixed, and the active state is an atomic token — a value from an enum.

LangGraph relaxes exactly those three. Its model is three primitives: a typed shared `State`, `Nodes` that are functions, and `Edges` that "determine which `Node` to execute next based on the current state".[86] First, *routing* is computed at runtime: `add_conditional_edges(node, routing_fn)` registers an arbitrary Python function — δ is *computed*, not *tabulated*, and it can even consult an LLM. Second, nodes can contain LLMs or "good ol' code".[86] Third, the state stops being an enum and becomes a typed object (`TypedDict`/Pydantic) with per-field *reducers* that merge each node's partial updates.[87] And the checkpointers persist a snapshot after every super-step with `thread_id`, time travel, and fault tolerance[88]: it is, in our interpretation, the journaling we built in chapters 4 and 7, industrialized.

```
stateDiagram-v2
    [*] --> IDLE
    IDLE --> SCANNING : bar_close
    SCANNING --> PENDING_ENTRY : setup_validado
    SCANNING --> IDLE : sin_setup
    PENDING_ENTRY --> IN_POSITION_LONG : fill
    PENDING_ENTRY --> SCANNING : cancelada
    IN_POSITION_LONG --> COOLDOWN : stop_tocado
    IN_POSITION_LONG --> COOLDOWN : take_profit
    COOLDOWN --> SCANNING : cooldown_expirado
    IDLE --> KILL_SWITCH : kill
    SCANNING --> KILL_SWITCH : kill
    PENDING_ENTRY --> KILL_SWITCH : kill
    IN_POSITION_LONG --> KILL_SWITCH : kill
    COOLDOWN --> KILL_SWITCH : kill
    KILL_SWITCH --> IDLE : reset_manual
```

The diagram above is our FSM from this guide seen through graph eyes: every state is a node, every event labels an edge. If you read `SCANNING → PENDING_ENTRY : setup_validado` as "the `SCANNING` node routes to the `PENDING_ENTRY` node when the routing function returns `setup_validado`", you already have the complete mental model of a `StateGraph`.

8.4 What's coming in the series

8.4.1 The bridge is official, and multi-agent trading is already built on it

We are not forcing an analogy. In the LangGraph launch post (January 2024), the LangChain team writes: "When talking about these more controlled flows, we internally refer to them as 'state machines'… LangGraph is a way to create these state machines by specifying them as graphs".[89] Moreover, in their taxonomy of cognitive architectures, *State Machine* (level 5: the code decides which steps to take, with cycles) is the rung immediately before *Agent* (level 6: the LLM decides).[89] Designing an FSM is, literally, the mental prerequisite for designing an agent.

The evidence in trading already exists. TradingAgents (arXiv 2412.20138, UCLA + MIT) implements a multi-agent trading firm — analysts, Bull/Bear debate, trader, risk committee, fund manager — built on LangGraph with a typed `AgentState` and `should_continue_debate` functions that inspect the state and return the next node:[90][91] it is $\delta(s) \to s'$ in pure form, with a round counter acting as a guard exactly as in our examples.

Dimension	Classic FSM (this series)	LangGraph StateGraph
What a "state" is	Value from an enum + auxiliary context	Complete typed object (`TypedDict`/Pydantic); modes are nodes
Transition function δ	Fixed table defined at design time	Routing function evaluated at runtime; can consult an LLM
Determinism	Deterministic and reproducible	Deterministic if nodes and routing are pure code; stochastic with LLMs
Context update	The handler mutates the context on the transition	The node returns a partial delta; the reducers merge it
Execution	One event → one transition, sequential	Pregel super-steps: nodes in parallel over an immutable snapshot
Persistence	Manual transition journaling	Checkpointers with `thread_id`, time travel, fault tolerance
Human pause	Explicit `AWAITING_*` state + manual event	Dynamic `interrupt()` + `Command(resume=...)`

Table of our own making from the cited official documentation. The important reading is not that one column replaces the other: the execution of each individual order will remain a deterministic, auditable FSM, because there determinism is a regulatory requirement, not a limitation. What the StateGraph adds is the upper orchestration layer — analyst debates, risk committees, human approvals — where dynamic routing and super-step parallelism do add value. The table also shows that almost every concept in this guide has a direct translation: whoever masters the left column already understands 80 % of the right one.

In the next quantarmy.com series on pythonparatrading.com we will do that translation step by step: we will reimplement this guide's state machine as a `StateGraph` with deterministic routing, add checkpointers as journaling, and culminate with a TradingAgents-style multi-agent analysis and risk system — with order execution delegated, as it should be, to an explicit FSM.

References

1. SEC, Order Instituting Administrative and Cease-and-Desist Proceedings, Release No. 34-70694 (In the Matter of Knight Capital Americas LLC), ¶¶1, 13–23 — https://www.sec.gov/files/litigation/admin/2013/34-70694.pdf (October 16, 2013) ↩↩↩↩↩↩↩

2. SEC, Order 34-70694, ¶18 — https://www.sec.gov/files/litigation/admin/2013/34-70694.pdf (October 16, 2013) ↩

3. SEC, Press Release 2013-222, "SEC Charges Knight Capital With Violations of Market Access Rule" — https://www.sec.gov/newsroom/press-releases/2013-222 (October 16, 2013) ↩↩

4. PRMIA, "The Knight Capital Algorithmic Trading Failure — A PRMIA Case Study" — https://prmia.org/common/Uploaded%20files/eAI/PRMIA%20Case%20study%20-%20Knight%20Trading.pdf ↩

5. SEC investor.gov, "Stock Market Circuit Breakers"; Nasdaq Trader, "Market Wide Circuit Breaker" — https://www.investor.gov/introduction-investing/investing-basics/glossary/stock-market-circuit-breakers ; https://www.nasdaqtrader.com/trader.aspx?id=CircuitBreaker ↩

6. Optiver, "5 things you should know about: market-wide circuit breakers" — https://www.optiver.com/insights/explainers/5-things-you-should-know-about-market-wide-circuit-breakers/ (March 2020) ↩

7. FINRA, "Guardrails for Market Volatility" — https://www.finra.org/investors/insights/guardrails-market-volatility (November 26, 2024) ↩

8. SEC-CFTC, "Findings Regarding the Market Events of May 6, 2010" — https://www.sec.gov/files/marketevents-report.pdf (September 30, 2010) ↩

9. NASDAQ OMX, "Market Access Rule FAQ" (Rule 15c3-5) — https://www.nasdaqtrader.com/content/productsservices/trading/ften/marketaccessrulefaq.pdf (2011) ↩

10. ESMA, "Q&A on MiFID II and MiFIR market structures topics" (Q17, kill functionality, RTS 6 Art. 12) — http://www.mfaalts.org/wp-content/uploads/2017/09/ESMA-QA-on-MiFID-II-and-MiFIR-Market-Structures-Topics.pdf (July 7, 2017) ↩

11. arXiv:2601.07525v1 — Def. 1 (citing M. Sipser, *Introduction to the Theory of Computation*, 1996, Def. 1.52) — https://arxiv.org/html/2601.07525v1 ↩

12. RFC 9293 — Transmission Control Protocol (STD 7, IETF, Aug-2022), §3.3.2; normative successor of RFC 793 (Sep-1981) — https://datatracker.ietf.org/doc/html/rfc9293 ↩↩

13. arXiv:2603.28903v1 — §II-A (δ: Q×Σ→2^Q, NFA); uniqueness of the reached state in arXiv:2605.28408v1 — https://arxiv.org/html/2603.28903v1 ; https://arxiv.org/html/2605.28408v1 ↩

14. arXiv:2604.11567v1 — §2, deterministic automaton (at most one successor) and complete (at least one) — https://arxiv.org/html/2604.11567v1 ↩

15. AcademiaLab — "Autómata finito determinista", section "Completo e incompleto" — https://academia-lab.com/enciclopedia/automata-finito-determinista/ ↩

16. Springer — "The Regular Languages" (2025), references: Mealy, *Bell System Technical Journal* 34(5), 1955; Moore, *Automata Studies*, Princeton UP, 1956 — https://link.springer.com/content/pdf/10.1007/978-3-031-84740-0_2 ↩

17. GeeksforGeeks — "Mealy and Moore Machines in TOC" (λ: Q→O vs λ: Q×Σ→O) — https://www.geeksforgeeks.org/theory-of-computation/mealy-and-moore-machines-in-toc/ ↩

18. D. Harel — "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming* 8 (1987), §2, p. 234 — https://www.state-machine.com/doc/Harel87.pdf ↩

19. K. Webber — "Mealy Machines, Moore Machines, and Why Event Sourcing Works" (2026) — https://kevinwebber.ca/blog/mealy-machines-and-event-sourcing/ ↩

20. QuantStart — "Event-Driven Backtesting with Python, Part I" (2014): same codebase for backtest and live — https://www.quantstart.com/articles/Event-Driven-Backtesting-with-Python-Part-I/ ↩

21. TutorialsPoint — "Deterministic Finite Automaton" (reduction: remove states unreachable from q0) — https://www.tutorialspoint.com/automata_theory/deterministic_finite_automaton.htm ↩

22. University of Mississippi, CSci 311 (based on P. Linz) — "A trap state is a state from which the automaton can never 'escape'" — https://john.cs.olemiss.edu/~hcc/csci311/notes/chap02/ch02.html ↩

23. fivenines.dev — "Design PFX: The order state machine" ("Terminal states … are absorbing — no arrows leave them") — https://fivenines.dev/problems/stock-exchange-the-order-state-machine ↩

24. Zaye Capital Markets — "The Turtle Trading Strategy: Rules, Results, and Why It Still Works Today" — https://zayecapitalmarkets.com/turtle-trading-strategy/ (2026-04-06) ↩

25. Uncoded — "The Circuit Breaker: Building Automated Fail-Safes" — https://uncoded.ch/blogs/the-circuit-breaker-building-automated-fail-safes (2026-06-09) ↩

26. python-statemachine — official documentation, "Transitions" — https://python-statemachine.readthedocs.io/en/latest/transitions.html (v3.2.0) ↩

27. python-statemachine — official documentation, "Conditions" — https://python-statemachine.readthedocs.io/en/latest/guards.html (v3.2.0) ↩

28. Miro Samek — "The Embedded Angle: Structuring State Machine Code" — https://www.state-machine.com/doc/Samek0312.pdf (2012) ↩

29. NexusFi — "Chandelier Exit: The Volatility-Adjusted Trailing Stop" — https://nexusfi.com/a/indicators/chandelier-exit (2026-06-01) ↩

30. python-statemachine — "Transitions" (internal transitions with `internal=True`) — https://python-statemachine.readthedocs.io/en/latest/transitions.html (v3.2.0) ↩

31. QuantifiedStrategies — "Fixed Fractional Position Sizing: Definition, Meaning And Examples" — https://www.quantifiedstrategies.com/fixed-fractional-position-sizing/ (2025-01-30) ↩

32. T3 Live — "How to Size Positions in Trading" — https://www.t3live.com/trading-position-sizes/ (n.d.) ↩

33. Altrady — "Turtle Trading Strategy: Turtle Trading Rules" — https://www.altrady.com/blog/crypto-trading-strategies/turtle-trading-strategy-rules (2026-04-27) ↩

34. Freqtrade — official documentation, "Plugins → Protections" (`CooldownPeriod`) — https://www.freqtrade.io/en/stable/plugins/ (accessed 2026) ↩

35. Trading Blox — Users Guide, "Turtle System Rules" ("Trade if Last is Winner" parameter) — https://www.tradingblox.com/Manuals/UsersGuideHTML/turtlesystem.htm (n.d.) ↩

36. Build a Finite State Machine in Python — https://belderbos.dev/blog/build-finite-state-machine-python/ (2026-04-07) ↩↩

37. transitions — official README (Machine, callbacks, conditions, GraphMachine, pickle) — https://github.com/pytransitions/transitions · https://pypi.org/project/transitions/ (release 0.9.3, 2025-07-02) ↩↩↩↩↩

38. Chandelier Exit: The Volatility-Adjusted Trailing Stop — https://nexusfi.com/a/indicators/chandelier-exit (2026-06-01) ↩

39. transitions README, "Check transitions" section (`may_<trigger>()` / `may_trigger()`) — https://github.com/pytransitions/transitions ↩↩

40. python-statemachine — repo and PyPI (v3.2.0, 2026-06-17; requires Python ≥3.10) — https://github.com/fgmacedo/python-statemachine · https://pypi.org/project/python-statemachine/ ↩↩

41. python-statemachine — "Conditions/Guards" documentation (declaration order, `enabled_events()`) — https://python-statemachine.readthedocs.io/en/latest/guards.html ↩↩↩↩

42. python-statemachine — "Transitions" documentation (`from_.any()`, `internal=True`) — https://python-statemachine.readthedocs.io/en/latest/transitions.html ↩

43. fgmacedo/python-statemachine releases (3.0.0 StateCharts SCXML/W3C; 3.2.0 safe IO for YAML/SCXML/JSON) — https://github.com/fgmacedo/python-statemachine/releases ↩↩↩↩

44. Issue #717: AsyncMachine runs callbacks with `asyncio.gather()` — https://github.com/pytransitions/transitions/issues/717 (2025-08-19) ↩

45. fiximate (FIX Trading Community) — FIX.4.4 Field #39 OrdStatus — https://fiximate.fixtrading.org/legacy/en/FIX.4.4/tag39.html ↩

46. FIX Protocol Ltd. — Appendix D: Order State Change Matrices, FIX 4.2 (b2bits mirror) — https://www.b2bits.com/fixopaedia/appendices/fix_42_appendix_d.html ↩↩

47. fiximate — FIX.Latest Field #39 OrdStatus ("Replaced — No longer used", deprecated FIX.4.3) — https://fiximate.fixtrading.org/en/FIX.Latest/tag39.html ↩

48. fiximate — FIX.4.4 Field #150 ExecType — https://fiximate.fixtrading.org/legacy/en/FIX.4.4/tag150.html ↩

49. fixsim.com — "FIX Execution Report (35=8) - Fields, States and Examples" — https://www.fixsim.com/fix-execution-report ↩

50. Binance — binance-spot-api-docs, `user-data-stream.md` (executionReport event, x/X fields) — https://github.com/binance/binance-spot-api-docs/blob/master/user-data-stream.md ↩

51. Alpaca — "Orders at Alpaca" (official order lifecycle diagram) — https://docs.alpaca.markets/docs/orders-at-alpaca ↩

52. NautilusTrader — "Execution" concepts (Overfills, deduplication by trade_id) — https://nautilustrader.io/docs/latest/concepts/execution/ ↩

53. Cboe — "Cboe Japan Equities SOR FIX Specification" v1.0.16, §6.1 Automatic Cancel on Disconnect (2025-06-09) — https://cdn.cboe.com/resources/membership/CXJ_SOR_Equities_FIX_Specification.pdf ↩

54. MIAX Sapphire — "Options Order Management using FIX Protocol", FOI v1.0 (2023-07-25) — https://www.miaxglobal.com/sites/default/files/page-files/Sapphire_FIX_Order_Interface_FOI_v1.0_0.pdf ↩

55. nautechsystems/nautilus_trader — `crates/model/src/orders/mod.rs` (develop), `impl OrderStatus { pub fn transition(...) }` — https://raw.githubusercontent.com/nautechsystems/nautilus_trader/develop/crates/model/src/orders/mod.rs ↩

56. hummingbot/hummingbot — `hummingbot/core/data_type/in_flight_order.py` (master) — https://raw.githubusercontent.com/hummingbot/hummingbot/master/hummingbot/core/data_type/in_flight_order.py ↩

57. mementum/backtrader — `backtrader/order.py` (master) — https://raw.githubusercontent.com/mementum/backtrader/master/backtrader/order.py ↩

58. freqtrade/freqtrade — `freqtrade/persistence/trade_model.py` (develop) — https://raw.githubusercontent.com/freqtrade/freqtrade/develop/freqtrade/persistence/trade_model.py ↩

59. hummingbot/hummingbot issue #7294 — "Failed Order Events followed by successful fills causing duplicate orders" (2024-11-13) — https://github.com/hummingbot/hummingbot/issues/7294 ↩

60. Interactive Brokers — TWS API "Placing Orders: Possible Order States" — https://interactivebrokers.github.io/tws-api/order_submission.html ↩

61. ESMA, "Q&A on MiFID II and MiFIR market structures topics" (Q17, kill functionality, RTS 6 Art. 12), 7-Jul-2017 — http://www.mfaalts.org/wp-content/uploads/2017/09/ESMA-QA-on-MiFID-II-and-MiFIR-Market-Structures-Topics.pdf ↩

62. SEC, Order Instituting Administrative and Cease-and-Desist Proceedings, Release No. 34-70694 (Knight Capital Americas LLC), ¶¶17, 19, 22–25 and footnote 7, 16-Oct-2013 — https://www.sec.gov/files/litigation/admin/2013/34-70694.pdf ↩↩↩↩

63. Freqtrade, official documentation, "Plugins → Protections" (MaxDrawdown, StoplossGuard) — https://docs.freqtrade.io/en/stable/plugins/ ↩↩

64. Bank of England / PRA, "Final Notice to Citigroup Global Markets Limited", May-2024 — https://www.bankofengland.co.uk/-/media/boe/files/prudential-regulation/regulatory-action/final-notice-from-boe-and-pra-to-citigroup-global-markets-limited.pdf ↩↩↩

65. algotradingdesk.com, "Automatic Kill-Switches in HFT Systems: The First Line of Survival", Feb-2026 — https://algotradingdesk.com/automatic-kill-switch-hft-risk-management/ ↩

66. CME Group Client Systems Wiki, "Risk Management Services" (Kill Switch, Cancel on Disconnect) — https://cmegroupclientsite.atlassian.net/wiki/display/EPICSANDBOX/Risk+Management+Services ↩↩

67. Katten / FIA L&C 2016, "Automated Trading Questions — Electronic Trading Risk Management Tools" (blocking <1 s) — https://katten.com/Files/151503_FIA%20Baltimore%20Automated%20Trading%20Questions%202016%20Presentation.pdf ↩

68. SEC, Press Release 2013-222, "SEC Charges Knight Capital With Violations of Market Access Rule", 16-Oct-2013 — https://www.sec.gov/newsroom/press-releases/2013-222 ↩

69. freqtrade/freqtrade, `freqtrade/plugins/protections/stoploss_guard.py` (develop branch, verbatim code) — https://raw.githubusercontent.com/freqtrade/freqtrade/develop/freqtrade/plugins/protections/stoploss_guard.py ↩

70. Novel Strategy Generating Variable-Length State Machine Test Paths — https://arxiv.org/pdf/2207.12172 (2022) ↩

71. ISTQB Certified Tester Advanced Level Test Analyst (CTAL-TA) Syllabus v4.0 — https://www.gasq.org/files/content/ISTQB2/ISTQB-CTAL-TA-Syllabus-v4.0-EN.pdf ↩↩↩

72. State transition testing — https://qajobfit.com/resources/state-transition-testing ↩↩

73. Kill Switch for Crypto Trading Bots: Circuit Breakers — https://vantixs.com/blog/kill-switches-and-circuit-breakers-crypto-bots (2026) ↩

74. Order Management Systems (OMS) — https://code-vb.com/order-management-systems-oms/ (2026) ↩

75. Hypothesis documentation — Stateful tests — https://hypothesis.readthedocs.io/en/latest/stateful.html ↩↩↩

76. Hypothesis changelog (6.7.0) — https://hypothesis.readthedocs.io/en/latest/changelog.html (2021) ↩

77. atomic-mesh: deterministic replay tests — https://github.com/davidakpele/atomic-mesh (2026) ↩

78. Should You Build Your Own Backtester? — QuantStart — https://www.quantstart.com/articles/Should-You-Build-Your-Own-Backtester/ ↩

79. Clarke, Grumberg, Peled — *Model Checking*, MIT Press (via Clarke, Klieber, Nováček, Zuliani, *Model Checking and the State Explosion Problem*) — https://pm.inf.ethz.ch/publications/Novacek12.pdf (2012) ↩

80. Khourshid (Stately/XState) — *You don't need a library for state machines* — https://stately.ai/blog/2021-01-20-you-dont-need-a-library-for-state-machines (2021-01-20) ↩

81. Samek — *The Embedded Angle: Structuring State Machine Code* — https://www.state-machine.com/doc/Samek0312.pdf (2012) ↩↩

82. Harel — *Statecharts: A Visual Formalism for Complex Systems*, Science of Computer Programming 8(3):231–274 — https://www.state-machine.com/doc/Harel87.pdf (1987) ↩

83. statecharts.dev — *History state* (glossary); UML 2.5.1 p. 309 — https://statecharts.dev/glossary/history-state.html ↩

84. python-statemachine 3.2.0 — official documentation (StateCharts SCXML) — https://python-statemachine.readthedocs.io/ (2026) ↩

85. Illinois CS 473 — *Basic Graph Properties* — https://courses.grainger.illinois.edu/cs473/fa2013/notes/17-graphs.pdf (2013) ↩

86. Graph API overview — Docs by LangChain — https://docs.langchain.com/oss/python/langgraph/graph-api (accessed 2026) ↩↩

87. Use the graph API — Docs by LangChain — https://docs.langchain.com/oss/python/langgraph/use-graph-api (accessed 2026) ↩

88. Persistence — Docs by LangChain — https://docs.langchain.com/oss/python/langgraph/persistence (accessed 2026) ↩

89. Chase, H. et al. — *LangGraph* (launch post, official LangChain blog) — https://blog.langchain.dev/langgraph/ (2024-01) ↩↩

90. Xiao, Sun, Luo, Wang — *TradingAgents: Multi-Agents LLM Financial Trading Framework*, arXiv:2412.20138 — https://arxiv.org/abs/2412.20138 (v1 2024-12-28) ↩

91. TauricResearch/TradingAgents — `tradingagents/graph/conditional_logic.py` — https://raw.githubusercontent.com/TauricResearch/TradingAgents/main/tradingagents/graph/conditional_logic.py ↩