TRADING CUANTITATIVO · ARQUITECTURA DE SISTEMAS

Máquinas de Estado Finito para Ejecución de Trading Cuantitativo en Python

De Knight Capital a LangGraph

Una guía técnica sobre el patrón que separa los bots que sobreviven de los que no: estado explícito, transiciones verificables y kill switches por construcción — con implementaciones ejecutables (manual, transitions y python-statemachine), el ciclo de vida de la orden FIX, testing property-based y el puente conceptual hacia LangGraph.

PUBLICADO EN
pythonparatrading.com

SERIE
Fundamentos de arquitectura — Parte 1

FECHA
Julio 2026

Índice

1. El día que 45 minutos costaron 460 millones

1.1 El desastre de Knight Capital como fallo de estado

1.1.1 Cronología oficial SEC: 212 órdenes padre → >4M ejecuciones en 154 acciones, >397M de títulos, posiciones +$3.5bn/−$3.15bn, pérdida >$460M en 45 minutos

El 1 de agosto de 2012, entre las 9:30 y las 10:15 a.m. ET, el router de órdenes SMARS de Knight Capital procesó 212 órdenes padre minoristas y, a partir de ellas, obtuvo más de 4 millones de ejecuciones en 154 acciones por más de 397 millones de títulos. Al detener el flujo, había acumulado una posición neta larga de unos $3.5 mil millones en 80 acciones y una neta corta de unos $3.15 mil millones en 74; la pérdida final superó los $460 millones según la orden de la SEC[1] (Knight anunció ~$440 millones antes de impuestos; ambas cifras son correctas en su contexto). En 75 acciones, sus ejecuciones superaron el 20% del volumen total del mercado[2]: el fallo interno de una firma se convirtió en riesgo para terceros.

Fecha / hora (ET)	Evento
27–31 jul 2012	Despliegue por etapas del código RLP; un técnico no copia el código a uno de los ocho servidores SMARS, sin revisión por un segundo técnico[1]
1 ago, 8:01	Un sistema interno empieza a generar 97 correos automáticos "BNET rejects" que mencionan SMARS y el error "Power Peg disabled"; nadie actúa[3]
1 ago, 9:30	Apertura. El octavo servidor interpreta un flag reutilizado como activación de Power Peg y entra en un bucle de envío de órdenes hijas[1]
9:30–10:15	212 órdenes padre → >4M ejecuciones, 154 acciones, >397M títulos[1]
10:15	Knight detiene el envío; posiciones no deseadas de +$3.5bn / −$3.15bn[1]
5–6 ago	~$400M de financiación de emergencia para evitar el colapso; en diciembre Knight acuerda su venta a Getco[4]
16 oct 2013	Multa de $12M: primera acción de enforcement de la SEC bajo la Regla 15c3-5[3]

La cronología muestra dos tiempos que conviene no confundir. El tiempo del incidente —45 minutos, unos $10 millones por minuto— es el que tarda una máquina sin estado de parada en destruir una firma. El tiempo de la señal —desde las 8:01, hora y media antes de la apertura— es el que el sistema estuvo avisando sin que existiera ninguna transición que conectara la señal con una acción. A esa tasa de pérdida, detectar, diagnosticar y decidir en comité cuesta más que el capital de la mayoría de las firmas: la contención tiene que ser estructural, no deliberativa.

1.1.2 La causa raíz no fue un bug de lógica sino de estado: código "muerto" reactivado, 97 correos de alerta ignorados, frase literal SEC: "halt SMARS's operations in response to its own aberrant activity"

La cadena causal, según la SEC, es una lección de teoría de estados disfrazada de incidente operativo. Knight dejó de usar la funcionalidad Power Peg en 2003 pero nunca la eliminó del código; en 2005 movió la función

de conteo acumulado de acciones a otro punto de la secuencia y no volvió a probarla[1]. En términos de máquina de estados: un estado que se creía eliminado del grafo seguía siendo alcanzable, y el nuevo código reutilizó el flag que lo activaba. El octavo servidor cayó en ese estado y entró en un bucle sin condición de salida, porque otra parte del sistema —que sí sabía que las órdenes padre estaban completadas— no se lo comunicaba a SMARS[1]: dos componentes con vistas divergentes del mismo estado.

Los 97 correos eran eventos emitidos sin handler. Y la orden SEC lo dice sin metáforas: Knight "did not have procedures in place to halt SMARS's operations in response to its own aberrant activity" [no tenía procedimientos para detener las operaciones de SMARS ante su propia actividad aberrante][1]. No faltaba código de parada en abstracto; faltaba que la parada fuera un estado del sistema.

1.2 El mercado ya es una jerarquía de máquinas de estado

1.2.1 Circuit breakers regulatorios como FSM literales: MWCB 7%/13%/20% sobre S&P 500 y LULD (Limit State 15s → pausa 5 min); Stop Logic de CME detuvo el Flash Crash 2010 con pausa de 5 segundos

Los reguladores aprendieron la lección antes que muchos desarrolladores. Los Market-Wide Circuit Breakers (MWCB), en su forma actual desde el 8 de abril de 2013, son una máquina de estados finitos (Finite State Machine, FSM) literal: caídas del S&P 500 del 7% (Nivel 1) o 13% (Nivel 2) antes de las 3:25 p.m. detienen todo el mercado 15 minutos; el 20% (Nivel 3) lo cierra por el día, y cada nivel solo se dispara una vez por sesión[5]. El Nivel 1 se ha activado cuatro veces: 9, 12, 16 y 18 de marzo de 2020[6]. Por valor individual, Limit Up-Limit Down (LULD) define otro autómata: si el precio toca la banda y no vuelve dentro de ella en 15 segundos ("Limit State"), la negociación se pausa 5 minutos[7].

El precedente empírico más fuerte es el Flash Crash del 6 de mayo de 2010: un vendedor fundamental descargó 75.000 contratos E-mini S&P 500 (~$4.1 mil millones) con un algoritmo sin límites de precio ni de tiempo. A las 2:45:28 p.m., la Stop Logic Functionality de CME pausó el E-mini cinco segundos; el informe conjunto SEC-CFTC documenta que esa pausa mínima rompió la cascada de ventas[8]. Una transición de estado de cinco segundos hizo lo que ningún humano pudo en veinte minutos.

1.2.2 SEC Rule 15c3-5 (Market Access): controles pre-trade y kill switch como obligación regulatoria; MiFID II RTS 6 Art. 12

La Regla 15c3-5, adoptada en 2010 y obligatoria desde julio de 2011, exige controles de riesgo automatizados y pre-trade —umbrales de crédito y capital, rechazo de órdenes erróneas o duplicadas— bajo "control directo y exclusivo" del broker-dealer[9]; la orden contra Knight fue su primera aplicación. En Europa, el RTS 6 de MiFID II (Artículo 12) obliga a las firmas de trading algorítmico a disponer de "kill functionality", y ESMA lo precisa: "a single decision of the investment firm should be able to result in an immediate withdrawal of all orders or any subset of them" [una única decisión debe poder resultar en la retirada inmediata de todas las órdenes o de cualquier subconjunto][10]. El kill switch no es una buena práctica: es una obligación legal con forma de estado absorbente.

1.3 Tesis del artículo

1.3.1 Tu bot ya tiene una máquina de estados: explícita (verificable) o implícita (flags e if/else dispersos); adelanto del hilo:

IDLE → SCANNING → PENDING_ENTRY → IN_POSITION_LONG → COOLDOWN → KILL_SWITCH

La tesis —interpretación nuestra a partir de los hechos citados— es que todo bot de trading ya tiene una máquina de estados. La única elección real es si es explícita —un grafo de estados y transiciones que puedes dibujar, testear y auditar— o implícita —una constelación de flags booleanos e if/else dispersos cuyos estados alcanzables nadie ha enumerado. Knight tenía la segunda: Power Peg era un estado fantasma, los 97 correos eran eventos sin transición y la parada era un procedimiento humano fuera de la máquina.

A lo largo de esta guía construiremos en Python la versión explícita: un ciclo IDLE → SCANNING → PENDING_ENTRY → IN_POSITION_LONG → COOLDOWN, más un KILL_SWITCH alcanzable desde cualquier estado y del que solo se sale por reset humano. El primer paso es dejar de hablar en metáforas: el capítulo 2 formaliza qué es exactamente una máquina de estados finitos, y esa formalización es lo que permite demostrar —no esperar— que tu bot no puede acabar donde acabó SMARS.

2. Qué es una FSM: el modelo matemático sin rodeos

El capítulo anterior mostró el coste de operar sin estado explícito. Este formaliza la alternativa, anclada en la definición que guía todo el artículo: **una Finite State Machine (FSM) no es un framework; es un patrón matemático: Estado actual + Evento → Función de Transición → Nuevo Estado**.

2.1 Definición formal

2.1.1 La quíntupla (Q, Σ, δ, q0, F); fórmula central: Estado + Evento → δ → Nuevo Estado

Un autómata finito se define como una 5-tupla:

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

donde Q es un conjunto finito de estados, Σ un alfabeto finito de entradas, $\delta : Q \times \Sigma \to Q$ la función de transición, $q_0 \in Q$ el estado inicial y $F \subseteq Q$ los estados de aceptación.[11] La fórmula central del artículo es la evaluación de δ en cada paso:

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

La traducción a trading es directa: Q son los estados de tu estrategia (FLAT, IN_POSITION_LONG, COOLDOWN…), Σ los eventos (setup_validado, fill, stop_tocado), q_0 el estado de arranque y δ la tabla de transiciones que tú escribes. Es la estructura con la que el RFC 9293 especifica TCP: 11 estados, con transiciones disparadas por llamadas de usuario, segmentos entrantes y timeouts.[12]

2.1.2 DFA vs NFA: determinismo exigido en ejecución; completitud: δ parcial = bot que "se cuelga" ante evento no contemplado

La única diferencia entre un Deterministic Finite Automaton (DFA) y uno no determinista (NFA) es la naturaleza de δ: en el NFA devuelve un *conjunto* de posibles sucesores ($\delta : Q \times \Sigma \to 2^Q$); en el DFA, exactamente uno, de modo que dado un estado y una secuencia de entradas existe un único estado alcanzado.[13] En ejecución real el no determinismo no es una opción: si ante (FLAT, fill) tu bot pudiera ir "a LONG o a SHORT según el caso", ningún auditor de riesgos podría razonar sobre él.

La otra cara es la **completitud**: un autómata es determinista si para todo (p, σ) existe *a lo sumo* un sucesor, y completo si existe *al menos* uno;[14] si δ es parcial, "cuando no se define ninguna transición, dicho autómata se detiene".[15] Un par (estado, evento) sin transición definida no es un caso raro que "nunca pasará": es comportamiento indefinido en producción. Si tu bot recibe un `fill` estando FLAT y la tabla no responde, la decisión la tomará el stack de excepciones de Python. La versión ingenieril de la completitud es una tabla exhaustiva, aunque muchas celdas resuelvan a "ignorar y loguear".

2.2 Mealy vs Moore: dónde viven las órdenes

2.2.1 Moore (salidas en estado) vs Mealy (salidas en transición); posición = Moore, orden de compra/venta = Mealy (se emite al cruzar la arista)

Para actuar, la FSM necesita salidas, y hay dos modelos clásicos — George Mealy (Bell Labs, 1955) y Edward Moore (Princeton, 1956)[16] — que difieren solo en la función de salida λ: en Moore, $\lambda : Q \to O$ — la salida depende solo del estado actual, es estable pero reacciona un ciclo tarde; en Mealy, $\lambda : Q \times \Sigma \to O$ — vive en la transición y reacciona en el mismo ciclo.[17]

El reparto en un motor de trading es asimétrico. La **posición neta** es una salida Moore: función exclusiva del estado (IN_POSITION_LONG $\Rightarrow$ +1 unidad), estable y auditable, lo que consultas para riesgo e informes. La **orden de compra o venta** es una salida Mealy: se emite *al cruzar la arista* — FLAT + setup_validado → LONG *con la acción* `send_order(BUY)` — porque enviarla un ciclo después introduce justo la latencia que Moore paga por su estabilidad. Harel etiquetó las flechas de sus statecharts con salidas "tipo Mealy" (*"Mealy-like outputs, or actions"*),[18] y el event sourcing moderno lo formula limpio: la transición devuelve el nuevo estado más una lista de acciones como datos que el entorno ejecuta después.[19] Esa separación — máquina pura que decide, efectos que actúan — es lo que hace testeable la estrategia sin bróker.

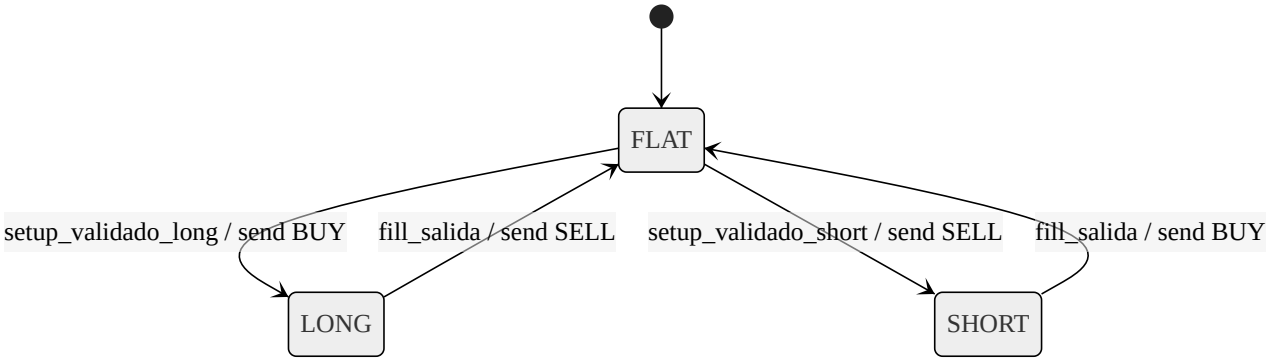
2.3 Propiedades que compras gratis

2.3.1 Determinismo → replay (backtest==live); alcanzabilidad (BFS); estados absorbentes → KILL_SWITCH sin aristas de salida excepto reset humano

Explicitar δ compra tres propiedades que el código con `if` anidados no tiene. Primera, **replay**: si la estrategia es una función pura $\delta(\text{estado}, \text{evento})$ y los eventos se journalizan, el backtest es literalmente un replay del live;[20] cuando divergen, el culpable es una impureza concreta (un reloj oculto, un side-effect fuera de δ). Segunda, **alcanzabilidad**: la tabla es un grafo, y un recorrido en anchura (BFS) desde q_0 detecta mecánicamente estados inalcanzables — código muerto o una arista olvidada.[21] Tercera, **estados absorbentes**: un estado trampa es aquel del que el autómata "nunca puede escapar",[22] y los matching engines los usan deliberadamente: los estados terminales de una orden son absorbentes por construcción, "sin flechas que salgan de ellos".[23] KILL_SWITCH es exactamente eso: un estado sin aristas de salida excepto un `reset_manual` humano. La seguridad deja de ser convención y pasa a ser un teorema sobre el grafo.

2.3.2 Diagrama Mermaid mínimo FLAT/LONG/SHORT + tabla de transiciones como especificación ejecutable

Cerramos con la máquina mínima que gobernará los ejemplos (la versión de seis estados llega en el capítulo 3):



El diagrama es legible, pero la especificación ejecutable es la tabla estado × evento, el mismo formato con el que el RFC de TCP especifica qué hacer en cada estado ante cada segmento:[12]

Estado \ Evento	setup_validado_long	setup_validado_short	fill_salida	stop_tocado
FLAT	→ LONG / `send BUY`	→ SHORT / `send SELL`	— (¿y si llega?)	— (¿y si llega?)
LONG	— (¿doble entrada?)	— (¿giro directo?)	→ FLAT / `send SELL`	→ FLAT / `send SELL`
SHORT	— (¿giro directo?)	— (¿doble entrada?)	→ FLAT / `send BUY`	→ FLAT / `send BUY`

Léela en dos pasadas. Como diseñador: cada celda con flecha es una transición con su acción Mealy, y el patrón simétrico de salidas (cerrar con la operación inversa a abrir) queda visible de un vistazo. Como auditor: las celdas con "—" son los pares donde δ es parcial, y cada una es una pregunta de riesgo concreta — un `fill_salida` llegando a FLAT suele ser un fill duplicado o tardío del bróker; un setup validado estando LONG plantea si permites piramidar o girar sin pasar por FLAT. Completar la tabla es tomar esas decisiones en diseño, no en producción.

3. Anatomía de una estrategia real: los seis estados

El capítulo 2 definió la máquina como una quíntupla y la tabla de transiciones como su representación operativa. Aquí instanciamos ese formalismo con una estrategia concreta y documentada: una ruptura de canal Donchian de 20 días al estilo del sistema Turtle (System 1), cuyas reglas mecánicas —entrada en nuevo máximo de 20 días, filtro de re-entrada, salida en canal opuesto de 10 días— están publicadas.[24] Al ser un sistema totalmente mecánico, cada regla se traduce sin ambigüedad a un estado, un evento o una guarda.

3.1 El mapa de estados del sistema

3.1.1 Catálogo de los 6 estados con su responsabilidad exacta

Cada estado tiene una única responsabilidad y un conjunto cerrado de eventos que consume; un evento que llega a un estado que no lo contempla se ignora o es un bug —esa es la propiedad que hace testeable la máquina.

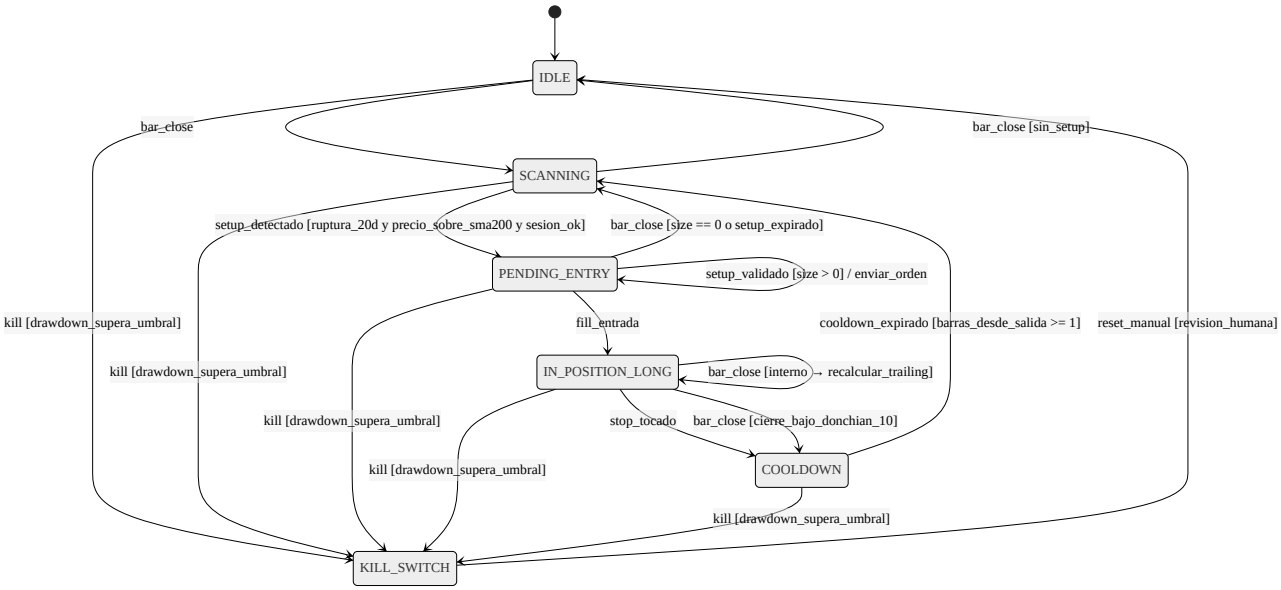
Estado	Responsabilidad exacta	Eventos que consume	Transiciones de salida
IDLE	Sin posición ni vigilancia activa; espera la siguiente barra	`bar_close`, `kill`	→ SCANNING; → KILL_SWITCH
SCANNING	Evaluar si el precio rompe el canal Donchian de 20 días	`setup_detectado`, `bar_close`, `kill`	→ PENDING_ENTRY; → IDLE; → KILL_SWITCH
PENDING_ENTRY	Setup validado: calcular sizing, fijar stop y enviar la orden	`setup_validado`, `fill_entrada`, `bar_close`, `kill`	→ IN_POSITION_LONG; → SCANNING (rechazo); → KILL_SWITCH
IN_POSITION_LONG	Gestionar la posición: trailing stop, invalidación, time-stop	`bar_close` (interno), `stop_tocado`, `kill`	→ COOLDOWN; → KILL_SWITCH
COOLDOWN	Bloquear la re-entrada durante 1 barra (o N segundos) tras el cierre	`cooldown_expirado`, `kill`	→ SCANNING; → KILL_SWITCH
KILL_SWITCH	Circuit breaker: no operar bajo ninguna señal de mercado	`reset_manual`	→ IDLE

La tabla codifica tres decisiones de diseño no obvias. Primera: SCANNING y PENDING_ENTRY son estados distintos aunque ambos estén "sin posición"; separarlos obliga a calcular el tamaño en frío, antes de la entrada, y no con la posición viva y el PnL fluctuando. Segunda: IN_POSITION_LONG consume `bar_close` como evento *interno* —recalcula el trailing stop sin cambiar de estado— mientras `stop_tocado` es transición de salida; la sección 3.3 desarrolla esta distinción, la más sutil del diseño. Tercera: KILL_SWITCH es el único estado alcanzable desde todos los demás y el único cuya salida exige un evento humano (`reset_manual`), nunca una señal de mercado: un circuit breaker "completely disables the

algorithm's ability to open new positions" [deshabilita por completo la capacidad del algoritmo de abrir nuevas posiciones] hasta una auditoría manual.[25]

3.1.2 Diagrama Mermaid completo con transiciones etiquetadas por evento y guarda

El diagrama es el contrato del sistema: cada arista lleva su evento y, entre corchetes, la guarda. El código del capítulo 4 implementará exactamente estas transiciones, ni una más ni una menos.



`kill` aparece cinco veces: en una FSM plana, un evento global exige una arista por estado origen (las librerías con jerarquía permiten factorizarlo como transición "desde cualquier estado").[26] Y fíjate en lo que *no* está: no hay arista IDLE → IN_POSITION_LONG. Nadie entra en una posición sin pasar por SCANNING y PENDING_ENTRY; la topología prohíbe estructuralmente el atajo.

3.2 Las guardas: la frontera entre elegancia y spaghetti

3.2.1 Guardas como filtros de régimen (solo long si precio > SMA200), horarios y volatilidad

Una guarda es un predicado booleano sobre el contexto que se evalúa cuando llega el evento: si falla, la transición no ocurre.[27] En la arista SCANNING → PENDING_ENTRY coexisten tres guardas canónicas: régimen (solo largos si el precio supera la SMA200, un filtro que no crea alpha sino que elimina trades malos), horario (no operar fuera de la sesión líquida) y volatilidad (no abrir si el ATR supera un múltiplo de su media). Son legítimas; el peligro está en su abuso. Miro Samek advierte que "abuse of extended state variables and guards is the primary mechanism of architectural decay" [el abuso de variables de estado extendido y guardas es el principal mecanismo de decaimiento arquitectónico]: las guardas terminan convirtiéndose en los mismos if/else que la máquina venía a eliminar.[28]

La heurística de diseño que proponemos (interpretación nuestra, no de Samek): **cada flag booleano que consulta una guarda es un estado encubierto; dos flags booleanos son cuatro estados encubiertos**. Si tu transición exige `in_regime and not recently_stopped`, mantienes $2^2 = 4$ configuraciones implícitas

sin nombre ni topología. Alguna —¿régimen alcista con cooldown activo?— ya tiene nombre en tu dominio: promuévela a estado explícito.

3.3 Eventos internos vs transiciones de salida

3.3.1 Trailing stop Chandelier como evento interno vs stop, invalidación y time-stop como salidas

Dentro de IN_POSITION_LONG conviven dos clases de sucesos que no conviene confundir. El **evento interno** actualiza el contexto sin cambiar de estado: es el caso del trailing stop Chandelier, $\text{stop} = \text{HH}(22) - 3 \times \text{ATR}$, donde $\text{HH}(22)$ es el máximo de las últimas 22 barras. Su propiedad clave es el trinquete monótono —el stop solo sube, nunca baja—; sin ella degenera en una banda ruidosa que te expulsa en cada pullback.[29] Cada `bar_close` recalcula el nivel mediante una transición interna que no ejecuta las acciones de entrada/salida del estado.[30] Las **transiciones de salida**, en cambio, destruyen la posición: `stop_tocado` (el precio alcanza el nivel), invalidación de tesis (cierre bajo el canal Donchian de 10 días: la idea deja de ser válida aunque el stop no se haya tocado) y time-stop (el movimiento esperado no llega en su ventana). Las tres llevan a COOLDOWN, pero conviene registrar el motivo: son diagnósticos distintos sobre la salud de la estrategia.

3.4 Sizing en PENDING_ENTRY

3.4.1 Fixed fractional (0.5–2 % del equity) y N-units Turtle, con ejemplo numérico reproducible

PENDING_ENTRY responde una única pregunta: ¿cuánto compro? El esquema estándar es el *fixed fractional*: arriesgar una fracción fija del equity por trade, típicamente entre el 0.5 % y el 2 %, de modo que el tamaño se deriva de la distancia al stop y no al revés.[31]

$$\text{size} = \left\lfloor \frac{ff \times \text{Equity}}{\text{riesgo\_trade}} \right\rfloor$$

con redondeo a la baja a la acción entera más próxima.[32] La variante Turtle (N-units) normaliza por volatilidad: una "unidad" es la cantidad tal que un movimiento de 1 ATR equivale al 1 % del equity, $\text{unidad} = (0.01 \times \text{Equity}) / \text{ATR}$.[33]

Ejemplo autocontenido. Equity = 100,000 €, $ff = 1\%$, ATR(20) = 2.50 €, entrada en ruptura a 48.00 € y stop inicial a $2 \times \text{ATR}$, es decir en $48.00 - 5.00 = 43.00$ €. Riesgo por acción: 5.00 €.

- Fixed fractional: $(0.01 \times 100{,}000)/5.00 = 1{,}000/5.00 = 200$ acciones. Capital inmovilizado: $200 \times 48.00 = 9{,}600$ €; pérdida si salta el stop: $200 \times 5.00 = 1{,}000$ €, exactamente el 1 %.
- N-units Turtle: $(0.01 \times 100{,}000)/2.50 = 400$ acciones. Un movimiento adverso de 1 ATR cuesta $400 \times 2.50 = 1{,}000$ € (1 %), pero con stop a 2 ATR la pérdida potencial es el 2 %: conviene verificar qué versión de la fórmula se está usando, pues algunas fuentes incluyen el $2 \times \text{ATR}$ en el denominador. Y si el cálculo diera 0 acciones, la guarda `size > 0` falla y la máquina vuelve a SCANNING: setup rechazado sin orden enviada.

Dos precisiones sobre el COOLDOWN que cierra el ciclo. El cooldown *temporal* —bloquear la re-entrada de un activo durante N velas o minutos tras cerrar— está documentado en Freqtrade como la protección `CooldownPeriod`.[34] La regla Turtle original, en cambio, era *condicional*, no temporal: si la última ruptura de 20 días fue ganadora (se operara o no), la siguiente señal se saltaba.[35] Son mecanismos distintos —uno espera un tiempo, el otro consulta el historial— y en nuestra FSM el segundo es una guarda sobre `setup_detectado`, no un estado.

4. Implementación en Python: de 50 líneas a librería profesional

El capítulo 3 fijó el contrato: seis estados (IDLE, SCANNING, PENDING_ENTRY, IN_POSITION_LONG, COOLDOWN, KILL_SWITCH) y un alfabeto de eventos (setup_detectado, setup_validado, fill_entrada, stop_tocado, bar_close, cooldown_expirado, kill, reset_manual). Aquí implementamos ese contrato tres veces — a mano, con `transitions` y con `python-statemachine` —, con una única simplificación respecto al diagrama: la auto-transición `setup_validado` sobre PENDING_ENTRY se pliega en `setup_detectado` para mantener los ejemplos mínimos. Así puedes comparar qué compra cada librería y qué precio cobra. Los tres códigos son ejecutables y producen exactamente el mismo recorrido de estados ante la misma secuencia de eventos.

4.1 Versión cero: Enum + tabla de transiciones

4.1.1 Implementación manual completa (~50 líneas): StrEnum de estados, dict (estado, evento) → estado, excepción InvalidTransition, historial append-only; máximo determinismo, cero dependencias

La implementación mínima seria cabe en una pantalla: un `StrEnum` para los estados, un diccionario que materializa la función de transición $\delta(\text{estado}, \text{evento}) \to \text{estado}$, una excepción para los pares no definidos y un historial append-only[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):
    """Se lanza cuando un evento no es válido desde el estado actual."""

# delta: (estado, evento) -> estado destino
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 caduca
    (Estado.IN_POSITION_LONG, "stop_tocado"): Estado.COOLDOWN,
    (Estado.COOLDOWN, "cooldown_expirado"): Estado.SCANNING,
    (Estado.KILL_SWITCH, "reset_manual"): Estado.IDLE,
}
# kill es alcanzable desde cualquier estado no absorbente
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: (estado, evento)

    def enviar(self, evento):
        clave = (self.estado, evento)
        if clave not in TRANSICIONES:
            raise InvalidTransition(f"{evento} no es valido desde {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 hay orden pendiente
    except InvalidTransition as exc:
        print(f"InvalidTransition: {exc}")
    fsm.enviar("kill")
    fsm.enviar("reset_manual")
    print("historial:", fsm.historial)
```

Tres decisiones de diseño importan más que el número de líneas. Primera: el diccionario *es* la tabla del capítulo 3, así que cualquier par (estado, evento) no listado es ilegal por construcción — el KILL_SWITCH solo admite `reset_manual` porque así lo dice la tabla, no por un comentario. Segunda: la política de error es *fail-fast* (`InvalidTransition`), la opción correcta para un motor de ejecución donde ignorar un evento inesperado esconde bugs. Tercera: el historial append-only convierte la estrategia en replayable: guarda los eventos y reconstruyes cualquier estado posterior, que es exactamente la propiedad que hace que backtest y live ejecuten el mismo código. Lo que esta versión no tiene es guardas (condiciones sobre datos, como "precio > SMA200") ni callbacks de entrada/salida: en cuanto los añadas como `if` dentro de `enviar`, estarás reimplementando una librería, mal. Ese es el límite honesto del enfoque manual[36].

4.2 La misma estrategia con `transitions`

4.2.1 Código ejecutable con Machine: callbacks prepare/before/after, conditions como guardas, may_trigger() para introspección, GraphMachine para exportar el diagrama

`transitions` (0.9.3) separa **modelo** (tu objeto, que guarda el estado y los datos) de **máquina** (la clase `Machine`, que inyecta los triggers como métodos del modelo)[37]. Las transiciones se declaran como datos — listas de dicts — lo que encaja bien con generar el grafo desde configuración:

```
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"},          # transicion interna
    {"trigger": "stop_tocado", "source": "IN_POSITION_LONG",
     "dest": "COOLDOWN", "after": "registrar_salida"},
    {"trigger": "bar_close", "source": "PENDING_ENTRY",
     "dest": "SCANNING"},                            # setup caduca
    {"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   # filtro de regimen (dato inyectado)
        self.stop = 0.0
        self.log = []

    # ---- guards: devuelven bool, sin efectos laterales ----
    def regimen_ok(self):
        return self.precio_sobre_sma200

    # ---- callbacks de accion ----
    def registrar_entrada(self):
        self.stop = 95.0
        self.log.append("entrada: stop inicial 95.0")

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

    def registrar_salida(self):
```

```
        self.log.append(f"salida por stop en {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("puede entrar?", e.may_trigger("setup_detectado"))  # False (guard)
    e.setup_detectado()                            # falla en silencio
    print("estado tras guard fallida:", e.state)   # sigue SCANNING

    e.precio_sobre_sma200 = True
    e.setup_detectado()
    e.fill_entrada()
    e.actualizar_trailing(maximo_barra=101.0)      # interna: no re-entra
    print("estado:", e.state, "| stop:", e.stop)
    e.actualizar_trailing(maximo_barra=99.0)       # el trinquete no baja
    print("stop tras 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)
```

Cuatro cosas que el código demuestra. Los callbacks siguen un orden fijo (`prepare` → `conditions` → `before` → cambio de estado → `after`), así que una transición es una secuencia de fases determinista[37]. El trailing stop se modela como transición interna (`dest=None`): actualiza datos sin salir ni re-entrar al estado, que es la semántica exacta de un stop con trinquete monótono (solo sube, nunca baja)[38]. El kill switch se declara con `source="*"`, alcanzable desde cualquier estado. Y `may_trigger()` da introspección sin mutar: un risk-check puede preguntar "¿puedo entrar ahora?" evaluando las guardas sin transicionar[39]. Advertencia operativa: cuando una `condition` falla, `transitions` **falla en silencio** (devuelve `False`, no lanza); tus tests deben asertar el estado resultante, no solo llamar al trigger[37].

Bonus de ecosistema: con `GraphMachine` y `graph_engine="mermaid"` exportas el diagrama directamente al formato que usamos en el capítulo 3, sin instalar 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):     # guard de regimen (siempre True en la 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)   # escribe el diagrama en formato mermaid
    print(buf.getvalue().decode())
```

La salida es un bloque `stateDiagram-v2` que incluye las guardas como etiquetas (`setup_detectado [regimen_ok]`) y marca `actualizar_trailing` como `[internal]` — documentación viva generada desde el propio código[37].

4.3 La misma estrategia con `python-statemachine`

4.3.1 Código ejecutable declarativo: guardas cond/unless con prioridad por orden de declaración, transición interna (internal=True) para el trailing stop, KILL_SWITCH accesible desde cualquier estado (from_.any() o patrón equivalente), enabled_events() para introspección; menciona StateChart v3 (estados compuestos) como avance

`python-statemachine` (3.2.0, requiere Python ≥3.10) adopta el estilo contrario: la máquina *es* una clase y las transiciones son atributos declarativos, auto-documentados[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")          # transicion interna
    stop_tocado = IN_POSITION_LONG.to(COOLDOWN)
    cooldown_expirado = COOLDOWN.to(SCANNING)
    kill = KILL_SWITCH.from_.any()                # desde cualquier estado
    reset_manual = KILL_SWITCH.to(IDLE)

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

    # guard: sin efectos laterales; si falla, la transicion no se ejecuta
    def regimen_ok(self):
        return self.precio_sobre_sma200

    def on_fill_entrada(self):
        self.stop = 95.0
        self.log.append("entrada: stop inicial 95.0")

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

    def on_stop_tocado(self):
        self.log.append(f"salida por stop en {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("habilitados sin regimen:",
          [e.id for e in sm.enabled_events()])          # solo kill
    sm.precio_sobre_sma200 = True
    print("habilitados con regimen:",
          [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)  # interna: sin exit/enter
    print("estado:", estado(sm), "| stop:", sm.stop)
    sm.send("stop_tocado")
    print("estado:", estado(sm))
    sm.send("kill")                                     # valido desde COOLDOWN
    print("estado:", estado(sm),
          "| KILL_SWITCH activo:", sm.KILL_SWITCH.is_active)
    sm.send("reset_manual")
    print("estado:", estado(sm))
    print("log:", sm.log)
```

Tres diferencias semánticas respecto a `transitions` que cambian cómo testeas. Primera: la política por defecto es estricta — un evento sin transición habilitada (o cuya guarda falla) lanza `TransitionNotAllowed`, el comportamiento fail-fast que queremos en ejecución[41]. Segunda: cuando varias transiciones comparten evento, el motor las evalúa **en orden de declaración y gana la primera cuya guarda pasa**, una semántica determinista ideal para codificar prioridades de riesgo (el kill-switch primero)[41]. Tercera: la introspección es `enabled_events()`, que devuelve los eventos con guardas satisfechas *ahora* — nótese en la demo cómo desaparece `setup_detectado` al apagar el filtro de régimen mientras `kill` permanece siempre habilitado[41]. El patrón `KILL_SWITCH.from_.any()` crea la transición global desde todo estado no final en una sola línea[42]. Por último, un avance: desde la v3.0 la librería incluye `StateChart`, con estados compuestos, paralelos e historia bajo semántica SCXML (estándar W3C); si tu FSM crece hacia subestados (p. ej. IN_POSITION_LONG con fases de piramidado), ese es el camino nativo[43].

4.4 Cuál elegir

4.4.1 Tabla comparativa: transitions 0.9.3 (6.6k★) vs python-statemachine 3.2.0 (1.3k★, StateCharts SCXML, Py≥3.10) vs manual; criterios: determinismo, testabilidad, serialización de estado, async, introspección; advertencia AsyncMachine de transitions (callbacks con asyncio.gather, issue #717) para motores live

Criterio	Manual (Enum + dict)	transitions 0.9.3	python-statemachine 3.2.0

Estilo	Tabla de datos + bucle propio	Data-driven: dicts + modelo	Declarativo: clase + `a.to(b)`
Evento inválido	`InvalidTransition` (tú decides)	Falla en silencio por defecto	`TransitionNotAllowed` (estricto)
Guardas	Manuales	`conditions`/`unless`	`cond`/`unless`, prioridad por orden de declaración
Transición interna	Lógica aparte	`dest=None`	`to.itself(internal=True)`
Introspección	`TRANSICIONES[estado]` directo	`may_trigger()` / `may_<trigger>()`[39]	`enabled_events()`[41]
Serialización de estado	Trivial (el estado es un string)	`model.state`; máquina pickleable[37]	`configuration_values`; definición exportable a YAML/SCXML/JSON[43]
Async	A mano	`AsyncMachine` (clase aparte)	Engine async autodetectado, callbacks secuenciales
Jerarquía	No	`HierarchicalMachine` (extensión)	`StateChart`: compuestos, paralelos, historia (SCXML)[43]
Salud del proyecto	Sin dependencia	6.6k★, release 2025-07, cadencia lenta	1.3k★, v3.2.0 (2026-06), muy activo, Py ≥3.10[40]

Nuestra recomendación por caso de uso. Para backtesting determinista y para aprender, la versión manual o `transitions`: la tabla de transiciones es inspeccionable de un vistazo, el estado se serializa como un string y `may_trigger()` facilita tests que no mutan nada. Para un motor live con asyncio, jerarquía y auditoría, `python-statemachine` 3.2: su engine async ejecuta los callbacks secuencialmente dentro del modelo run-to-completion, mientras que la `AsyncMachine` de `transitions` lanza los callbacks de una misma fase con `asyncio.gather` — ejecución concurrente que rompe supuestos de orden al gestionar recursos, un problema abierto en su issue #717 que exige subclasear si dos callbacks deben correr en orden (registrar fill → actualizar PnL)[44]. Además, la semántica SCXML con tests de conformidad W3C y la definición cargable desde YAML hacen el grafo auditable como dato, no solo como código[43]. La opción manual sigue siendo la referencia de oro para verificar por replay que las dos librerías implementan el mismo contrato: mismos eventos, mismo historial.

Con la FSM de estrategia implementada y verificada, queda pendiente la otra máquina de estados del sistema: la del ciclo de vida de cada orden, que es donde la ejecución real se encuentra con el bróker. Es el tema del capítulo 5.

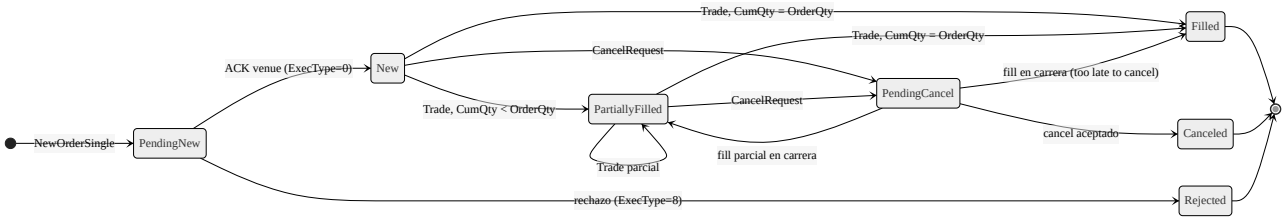
5. La segunda máquina: el ciclo de vida de la orden

La FSM de estrategia del capítulo 4 decide *cuándo* actuar. Pero al llamar a `buy()` aparece una segunda máquina, independiente de la primera: cada orden viva tiene su propio estado, eventos y transiciones. Esta máquina no tienes que inventarla: el protocolo FIX (Financial Information eXchange) la estandarizó en 1992, y cada API moderna —Binance, Alpaca, Interactive Brokers (IBKR)— es un dialecto del mismo grafo. Tu trabajo no es diseñar el alfabeto, sino hacerlo explícito en tu Order Management System (OMS) en lugar de sufrirlo implícito.

5.1 El alfabeto de estados de una orden

5.1.1 FIX OrdStatus (tag 39)

FIX define el estado de una orden como un único carácter en el tag 39, `OrdStatus`: `0` = New, `1` = PartiallyFilled, `2` = Filled, `4` = Canceled, `8` = Rejected, `A` = PendingNew, `6` = PendingCancel, `E` = PendingReplace, entre otros.[45] La especificación incluye una matriz oficial de transiciones (Appendix D): filas = estado actual, columnas = siguiente estado reportado, y cada estado lleva un valor de precedencia que decide qué reportar cuando la orden está en varios estados a la vez (parcialmente ejecutada *y* con cancelación pendiente).[46] Dos detalles son lecciones de diseño. Primero: las filas de `Canceled` y `Rejected` están vacías —estados absorbentes. Segundo: `Replaced` existió como estado (`5`) en FIX 4.2, pero se eliminó en FIX 4.4 y hoy consta como "No longer used":[47] el estándar corrigió su propia FSM, degradando "replaced" de estado a *evento* (`ExecType=5`); tras un replace, la orden vuelve a `New` o `PartiallyFilled`.



El diagrama simplifica el núcleo de la matriz FIX (omite `PendingReplace`, `Expired`, `DoneForDay`). Fíjate en las dos aristas de carrera que salen de `PendingCancel`: volveremos a ellas en 5.3.

5.2 Eventos ≠ estados: la trampa clásica

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

Todo el ciclo de vida llega por un único mensaje, el `ExecutionReport` (MsgType `8`), con dos campos que las implementaciones novatas confunden: `ExecType` (tag 150) describe *qué acaba de ocurrir* —el evento—, mientras `OrdStatus` identifica *siempre* el estado actual —el nodo destino.[48] En vocabulario de autómatas: `ExecType` es el alfabeto de entrada; `OrdStatus`, el conjunto de estados. La documentación de referencia es tajante: "This is where most implementations go wrong… A common mistake is treating them as the same

field" [Aquí es donde la mayoría de las implementaciones fallan… un error común es tratarlos como el mismo campo].[49]

La distinción no es pedantería: el mismo evento `F` = Trade conduce a `PartiallyFilled` o a `Filled` según la guarda que compara `CumQty` (cantidad acumulada, tag 14) con `OrderQty` (tag 38). El evento por sí solo no determina la transición. Binance replica esta dualidad exacta: cada `executionReport` de su user data stream lleva `x` (execution type) y `X` (order status).[50]

5.3 Los casos que rompen bots

5.3.1 Carreras, zombies y duplicados

Carrera cancel-vs-fill. Una orden puede ejecutarse mientras tu cancelación está en vuelo: FIX lo documenta en las matrices D4/D5 del Appendix D, donde el fill y el `CancelRequest` "se cruzan en la conexión" y el bróker responde `Cancel Reject` con `CxlRejReason = 0` —"too late to cancel".[46] Consecuencia: tras enviar un cancel, la orden está en `PendingCancel`, no en `Canceled`, y los fills de esa ventana son reales. Alpaca dibuja estas aristas en su diagrama oficial ("Original Order filled before cancel").[51]

Fills duplicados. La recuperación de sesión (replay de WebSocket, `PossResend` en FIX) puede entregar el mismo fill dos veces. La defensa estándar es la idempotencia por ID de ejecución (`ExecID`, tag 17; `trade_id` en crypto). Nautilus Trader lo impone como invariante duro: "This is the invariant that prevents double-counting executions" [Esta es la invariante que evita el doble conteo de ejecuciones].[52]

La orden zombie. Si la conexión muere antes del ACK, la orden queda en estado desconocido: puede no haber llegado, estar viva o haberse ejecutado. Cboe cancela las órdenes abiertas tras dos heartbeats sin mensajes del cliente, "to prevent orders from being stuck in an unknown state" [para evitar que las órdenes queden atrapadas en un estado desconocido];[53] y MIAX advierte que su cancel-on-disconnect es *best effort*: "Executions can occur while FOI is processing the ACOD event" [Pueden ocurrir ejecuciones mientras se procesa el evento ACOD].[54] Sin un estado in-flight honesto y reconciliación al reconectar, este caso es inmanejable.

5.4 Cómo lo hacen los frameworks

5.4.1 De la FSM explícita a los strings libres

Framework	Alfabeto de estados	FSM	Verificación de transiciones
Nautilus Trader	`OrderStatus`, 14 estados (núcleo Rust)	Explícita + event sourcing	Sí: `Err(InvalidStateTransition)`
Hummingbot	`OrderState`, 11 estados	Semi (enum + `ClientOrderTracker`)	Parcial (predicados `is_open`/`is_done`)
Backtrader	9 constantes enteras	Implícita (vive en el bróker simulado)	No

Freqtrade	Strings libres (espejo de CCXT) + flag `ft_is_open`	Implícita	No

La tabla ordena los frameworks por rigor creciente de abajo arriba. Nautilus Trader codifica la transición como función pura `match (estado, evento) -> Result` en Rust: cualquier par no enumerado devuelve `InvalidStateTransition`, la orden se reconstruye reproduciendo su stream de eventos, e incluso documenta aristas contraintuitivas como `Canceled → Filled` ("Real world possibility").[55] Hummingbot define un enum con estados terminales (`CANCELED`, `FILLED`, `FAILED`) separados de los activos, pero sin verificador central.[56] Backtrader reparte nueve estados por el bróker y la estrategia reacciona en `notify_order`.[57] Freqtrade ni siquiera tiene enum: `Order.status` es un string libre heredado de CCXT más un booleano `ft_is_open`.[58] El coste de la FSM implícita es medible: el issue #7294 de Hummingbot documenta un falso `MarketOrderFailureEvent` que disparó reintentos mientras la orden original sí se ejecutaba —órdenes duplicadas por no modelar el estado in-flight.[59] El mapeo entre venues es casi isomorfo: `Submitted` de IBKR cubre `New` y (con `filled > 0`) `PartiallyFilled`; Alpaca y Binance nombran los estados igual que FIX; IBKR añade estados "pre-venue" (`PendingSubmit`, `PreSubmitted`) que FIX colapsa en `PendingNew`.[60]

Y una observación estructural de cierre: `Canceled` y `Rejected` son absorbentes por construcción en la matriz FIX —el mismo patrón que el `KILL_SWITCH` a nivel de estrategia que explota el próximo capítulo.

6. KILL_SWITCH y COOLDOWN: el riesgo como topología

En el capítulo 2 definimos los estados absorbentes como vértices sin aristas de salida. Aquí les damos su uso decisivo: el control de riesgo no es una funcionalidad que se añade a la estrategia, es una propiedad de la topología del grafo. O tu máquina tiene un vértice del que no se sale sin permiso humano, o tu "kill switch" es una convención que el código puede violar.

6.1 El kill switch como estado absorbente por construcción

6.1.1 Sin aristas de salida excepto reset humano

Un kill switch bien construido cumple dos propiedades demostrables sobre el grafo: desde cualquier estado operativo existe una transición `kill` hacia KILL_SWITCH (alcanzabilidad), y desde KILL_SWITCH solo existe una arista, `reset_manual`, que exige intervención humana (absorción). La entrada dispara una secuencia atómica —dejar de aceptar señales → cancelar todas las órdenes vivas → liquidar posiciones si la política lo exige → halt— que nadie puede interrumpir a mitad. MiFID II (RTS 6, Art. 12) exige exactamente esto —la retirada inmediata de todas las órdenes o de cualquier subconjunto por una única decisión, como vimos en el capítulo 1—.[61]

El contraste con Knight Capital no es la cronología ya contada, es la arquitectura. La orden de la SEC reconoce en su nota al pie 7 que Knight sí tenía apagado automático para *ciertas estrategias de uno de sus grupos*: el control existía, pero no cubría todos los caminos de emisión de órdenes — y SMARS quedó fuera.[62] Un kill switch parcial es topológicamente equivalente a ninguno para los caminos no cubiertos. Y su herramienta principal de riesgo, PMON, era un monitor post-ejecución que "relied entirely on human monitoring and did not generate automated alerts" [dependía por completo de la supervisión humana y no generaba alertas automáticas]:[62] la parada vivía fuera de la máquina, en una persona mirando una pantalla. Un `if` disperso en tu código tiene el mismo defecto: no es un estado, y no se puede demostrar que nadie opere mientras está activo.

6.2 Triggers: la taxonomía de lo que debe detenerte

6.2.1 Qué eventos deben disparar `kill`

Los triggers no se inventan; la práctica industrial y el expediente regulatorio los han catalogado:

Trigger	Qué detecta	Precedente documentado
Pérdida diaria / drawdown	Ruina lenta: la estrategia o el mercado se volvieron contra ti	`MaxDrawdown` de Freqtrade: "If the observed drawdown exceeds `max_allowed_drawdown`, trading will stop for `stop_duration`"[63]
Tasa de órdenes anómala	Bucle de emisión: órdenes/segundo o ratio cancelaciones/ejecuciones fuera de rango	Es el síntoma exacto de Knight: 212 órdenes padre → >4 millones de ejecuciones sin ningún control que comparara salida con entrada[62]

Desviación de precio	Datos corruptos o fat finger: precio vs. mid, último o feed externo	Citigroup 2022: una cesta de $58M se capturó como $444bn y $1.4bn llegaron a ejecutarse[64]
Tasa de rechazos / fills inesperados	Desincronización con el venue: tu modelo del libro no coincide con la realidad	Los 97 correos "BNET rejects" de Knight fueron eventos emitidos sin handler[62]
Latencia / conectividad	ACKs lentos, heartbeat perdido, datos obsoletos: "stale data is more dangerous than no data" [los datos obsoletos son más peligrosos que no tener datos][65]	Cancel on Disconnect de CME: el venue cancela tus órdenes resting si tu sesión cae[66]
Integridad lógica	NaN, precios negativos, invariantes internos rotos	Los hard blocks de Citi bloquearon $248bn; los 711 warnings *ignorables* no — multa total de £61.6M[64]

La tabla importa menos por su contenido que por su criterio de lectura: cada fila es un evento del alfabeto de tu FSM con una transición definida hacia KILL_SWITCH o COOLDOWN. La lección de Citigroup es que un warning que no transiciona no es un control: 646 soft blocks aparecieron en un pop-up del que solo se veían 18 líneas, y el trader pudo cerrarlo y continuar.[64] La referencia industrial es el Kill Switch de CME Group: bloquea toda entrada de órdenes nuevas y cancela todas las vivas en menos de un segundo.[66][67] Y la Regla 15c3-5 —bajo la que Knight pagó $12M en la primera acción de enforcement— exige umbrales de capital *vinculados* a controles automatizados pre-trade, medidos sobre órdenes ingresadas, no ejecutadas.[68] Traducción a tu grafo: la guarda vive en la función de transición, antes de emitir, no en un informe posterior.

6.3 COOLDOWN como estado con salida temporizada

6.3.1 Por qué es un estado y no un timestamp suelto

COOLDOWN es un KILL_SWITCH con salida temporizada: mismo mecanismo (prohibición estructural de re-entrar), distinta arista de salida (`cooldown_expirado` en lugar de `reset_manual`). La tentación del programador es un `if time.now() > last_exit + delta` esparcido por el código; la objeción es que ese timestamp suelto no es testeable como invariante, no aparece en el diagrama y nada impide que otro camino lo ignore. Modelado como estado, la prohibición es una propiedad del grafo: desde COOLDOWN no existe arista hacia PENDING_ENTRY hasta que llega el evento de expiración.

Freqtrade lo implementa literalmente así: `StoplossGuard` devuelve `ProtectionReturn(lock=True, until=until, reason=...)` — un estado de bloqueo con su timestamp de desbloqueo embebido, "assuming that the bot needs some time to let markets recover" [asumiendo que el bot necesita tiempo para que el mercado se recupere].[63][69] Es la versión honesta de un principio incómodo: tras una racha de pérdidas tu modelo del mercado es menos fiable, y la máquina debe imponerte la pausa que tú no te impondrías.

Con esto queda montada la pieza que explotará el capítulo 7: "desde KILL_SWITCH no se opera" y "desde COOLDOWN no se re-entra hasta `cooldown_expirado`" no son aspiraciones de diseño, son invariantes sobre un grafo finito — y una invariante sobre un grafo finito es testeable de forma exhaustiva.

7. Testing: la tabla de transiciones es tu suite de tests

Los capítulos 3 y 4 terminaron con una máquina explícita: seis estados y una tabla `TRANSICIONES` que decide, para cada par (estado, evento), si hay transición y a dónde. Este capítulo explota la consecuencia práctica de esa finitud: el espacio de comportamientos es enumerable, y la cobertura exhaustiva deja de ser un lujo para convertirse en el estándar razonable.

7.1 Cobertura de estados y transiciones

7.1.1 State coverage y transition coverage (0-switch/1-switch según ISTQB): objetivo de cobertura medible y alcanzable al 100%

Sobre el grafo de la FSM, la literatura de model-based testing define criterios graduales: *state coverage* (todo estado visitado al menos una vez), *transition coverage* o **0-switch** (toda transición ejercida al menos una vez), **1-switch** (todo par de transiciones consecutivas) y N-switch (toda secuencia de N+1 transiciones).[70] El syllabus ISTQB CTAL-TA señala que 0-switch y 1-switch son los criterios frecuentes en la práctica, y que 2-switch o superior solo se justifica ante alto riesgo, pues el número de N-switches crece exponencialmente con N.[71] Advertencia terminológica: algunas fuentes divulgativas llaman 0-switch a la cobertura de estados; aquí seguimos la convención ISTQB, donde 0-switch equivale a cobertura de transiciones válidas.[71]

La traducción a tu estrategia es directa: con 6 estados y 7 transiciones más `kill` y `reset_manual`, el 100% de 0-switch exige ejercer exactamente 9 aristas, y un único guion de 11 eventos las cubre todas (`test_0_switch_todas_las_transiciones_ejercidas`, en el bloque siguiente). ISTQB añade el criterio *round-trip coverage* —ciclos que empiezan y terminan en el mismo estado—, muy efectivo detectando defectos;[71] en trading, un round trip es literalmente `IDLE → IN_POSITION_LONG → IDLE`: una operación completa.

7.2 Tests de transiciones inválidas

7.2.1 Matriz negativa estado×evento: asertar excepción + no mutación + sin efectos laterales; código pytest ejecutable

Las transiciones ausentes son tan importantes como las presentes: cada par (estado, evento) sin arista es una promesa de rechazo. El test negativo correcto verifica tres cosas: excepción esperada, estado sin cambios y ausencia de efectos laterales (posición, stop, órdenes y journal intactos).[72] Como la tabla es un dato, la matriz negativa completa —6 estados × 8 eventos = 48 pares, 35 ilegales— se genera por parametrización, no a mano. El siguiente bloque (pytest 8.x, Python 3.11+) redefine la FSM mínima para ser autocontenido e incluye el test de 0-switch y el de replay de la sección 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:
    """Version minima de la FSM de los capitulos 3-4: tabla + estado extendido."""

    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  # cancela lo pendiente
            self.journal.append(evento)
```

```
            return
        destino = TRANSICIONES.get((self.estado, evento))
        if destino is None:
            raise TransicionInvalida(f"{evento} no es valido desde {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, "reconcilia la posicion antes de resetear"
        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))
```

Tres decisiones de diseño importan aquí. Primera: el constructor acepta el estado inicial, así que cada test siembra su punto de partida en lugar de navegar hasta él; una transición previa rota no contamina el test bajo examen.[72] Segunda: el test negativo aserta sobre todo el estado observable, incluida la longitud del journal —un evento rechazado que dejara rastro en la auditoría sería un defecto—. Tercera: `pares_invalidos()` se deriva de la propia tabla, de modo que añadir una transición al modelo actualiza la suite automáticamente: la tabla es literalmente la especificación ejecutable de los tests.

7.3 Invariantes y property-based testing

7.3.1 Catálogo de invariantes: "nunca IN_POSITION_LONG sin stop", "desde KILL_SWITCH no se opera", "posición neta coherente con fills"; Hypothesis RuleBasedStateMachine con @invariant y reglas; código ejecutable

Los tests anteriores verifican transiciones una a una; las invariantes verifican propiedades que deben cumplirse tras *cada* paso, en cualquier secuencia. Catálogo mínimo para esta FSM, derivado de la práctica documentada de OMS y kill switches:[73][74]

- **Stop obligatorio:** `IN_POSITION_LONG` implica `stop_price is not None`.
- **KILL_SWITCH es sumidero operativo:** en `KILL_SWITCH` no hay órdenes abiertas, y la única salida es `reset_manual` tras reconciliar la posición.

- **Coherencia posición–fills:** la posición neta coincide con la suma de fills firmados recomputada desde el journal; en estados planos es cero.

El stateful testing de Hypothesis (documentación 6.161.5) genera secuencias de operaciones, no solo datos: declaras reglas (`@rule`) con argumentos extraídos de estrategias `st.*`, y el motor busca cadenas de reglas que violen algún `@invariant`, reduciéndolas al programa mínimo que reproduce el fallo.[75] `@precondition` filtra las reglas inaplicables antes de ejecutarlas,[75] y desde la versión 6.7.0 las invariantes solo se comprueban tras los `@initialize`.[76] El patrón model-based consiste en mantener junto al sistema una implementación de referencia deliberadamente simple (aquí, el contador `modelo_posicion`) y asertar que ambas concuerdan:[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:  # misma maquina del bloque anterior, repetida por autocontencion
    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} no es valido desde {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, "reconcilia la posicion antes de resetear"
        self.estado = destino
        self.journal.append(evento)


class EstrategiaMachine(RuleBasedStateMachine):
    """Hypothesis genera secuencias de eventos; las invariantes se verifican tras cada paso."""

    def __init__(self):
        super().__init__()
        self.fsm = EstrategiaFSM()
        self.modelo_posicion = 0  # implementacion de referencia (model-based)

    def _intentar(self, evento, **payload):
        try:
            self.fsm.handle(evento, **payload)
            return True
        except TransicionInvalida:
            return False  # evento ilegal en este estado: el rechazo ya es el contrato

    @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)
```

Nótese que las reglas no evitan los eventos ilegales: los disparan a propósito y tratan el rechazo como parte del contrato, de modo que Hypothesis explora intercalados hostiles —`kill` a mitad de una entrada, `fill` tardío, doble `stop_tocado`— que ningún guion manual habría combinado. La precondición de `resetear` codifica una regla de negocio real: no se rearma la máquina con posición abierta sin reconciliar. `derandomize=True` fija la semilla para que CI sea reproducible. Verificamos este código por mutación: si `kill` deja de cancelar órdenes, o si `fill` no registra el stop, las invariantes correspondientes fallan —la suite muerde.

7.4 Replay: determinismo como test de oro

7.4.1 replay_determinism_same_events_same_hash: journal de eventos + FSM pura = backtest y live son el mismo programa; divergencia = impurezas (reloj oculto, side-effects fuera de δ)

El último test cierra el argumento. Si `handle()` es una función de transición δ pura —su resultado depende solo del estado interno y del evento— y cada evento queda en el journal, reejecutar el journal es reejecutar la estrategia. El patrón, usado en motores de trading event-sourced de producción, se nombra literalmente `replay_determinism_same_events_same_hash`: procesar el mismo log dos veces por el motor completo debe producir un hash de estado idéntico.[77] Es el test incluido en el primer bloque.

La implicación es fuerte: un backtest es un replay del journal sobre datos históricos y el live es el mismo programa alimentado por eventos del mercado; reutilizar el mismo código de estrategia y riesgo en ambos es justo la ventaja que la literatura event-driven atribuye a este diseño.[78] La lectura inversa sirve de diagnóstico: si backtest y live divergen, el test de replay convierte ese misterio en un síntoma acotable —hay impureza en δ: un reloj real consultado dentro de `handle()`, un `random` sin semilla, una llamada de red, un efecto lateral fuera de la función de transición—. El determinismo no se declara; se testea. Y este journal serializado es el antepasado directo de los checkpointers que aparecerán en el capítulo 8 al cruzar el puente hacia LangGraph.

8. Cuando la FSM se queda corta: de estados a grafos (puente a LangGraph)

A lo largo de esta guía construimos una estrategia de seis estados con `KILL_SWITCH` absorbente, journaling auditable y tests de propiedades. Esa máquina es correcta, determinista y verificable. También tiene un techo: este capítulo delimita dónde está, qué respuestas clásicas existen y por qué el siguiente paso —los grafos de estado de LangGraph— no es un salto de paradigma sino una generalización de lo que ya sabes.

8.1 Las tres grietas de la FSM plana

8.1.1 Explosión de estados, paralelismo y memoria

La primera grieta es aritmética. Cada dimensión binaria que añades al comportamiento (¿trailing activo? ¿parcial del 50 % ejecutado? ¿break-even movido?) multiplica el espacio de estados: con n flags hay hasta 2^n configuraciones, y con N posiciones independientes de k estados cada una, k^N estados globales. En la literatura de verificación formal esto tiene nombre propio: el *state explosion problem*, "el problema central del model checking" según el texto de referencia de Clarke, Grumberg y Peled.[79] Con 6 flags de gestión de salida ya son 64 estados planos; con 10 símbolos, más de 10^{18} configuraciones. Y no solo explotan los estados: en una FSM plana las transiciones tienden a crecer exponencialmente con ellos, como señala David Khourshid, creador de XState.[80]

La segunda grieta es estructural: una FSM clásica está en **un solo estado a la vez**. Gestionar trailing-stop *y* take-profit *y* parciales simultáneamente exige el producto cartesiano de todos ellos como estados compuestos. La tercera es la ausencia de memoria: si una posición en gestión se interrumpe (halt del mercado, desconexión del bróker) y debe reanudarse en el subestado exacto donde estaba, la FSM plana obliga a codificar una variable "último subestado" con guardas — memoria encubierta en flags, precisamente el antipatrón que Samek identifica como "el principal mecanismo de decaimiento arquitectónico" de las máquinas de estados.[81]

8.2 Statecharts: la respuesta clásica

8.2.1 Harel 1987 y su disponibilidad actual en Python

David Harel diagnosticó estas carencias en 1987 y definió los statecharts como "state-diagrams + depth + orthogonality + broadcast-communication".[82] La jerarquía factoriza comportamiento común: tu `KILL_SWITCH` se define **una vez** en el estado padre y lo heredan todos los subestados de `IN_POSITION`, en lugar de repetir la transición en cada uno (olvidar una repetición es un bug de riesgo real). La ortogonalidad introduce regiones paralelas — una para stops, otra para parciales, otra para el time-stop — que evitan el producto cartesiano. Los *history states* resuelven la reanudación: un estado compuesto "recuerda" qué subestado estaba activo al salir.[83] Y el estado extendido formaliza lo que ya hicimos en esta guía: el modo es cualitativo (`TRAILING`), el nivel del stop es cuantitativo (contexto, no estado).

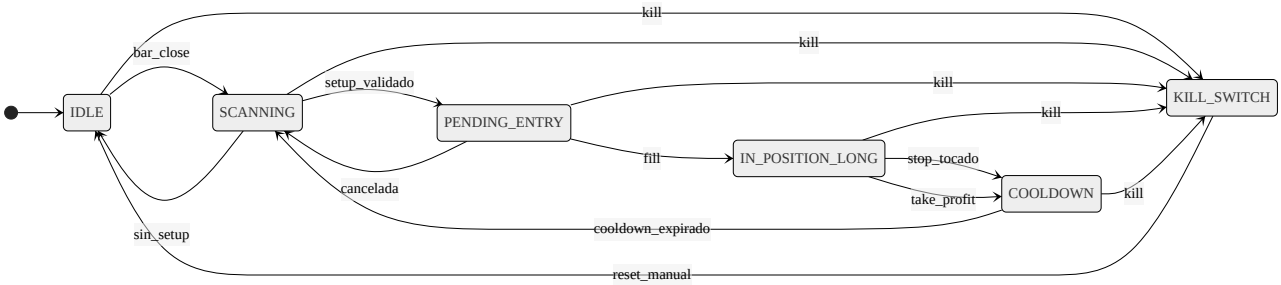
Esto no es teoría de museo: python-statemachine 3.2.0 soporta StateCharts SCXML con estados compuestos, paralelos e history, y transitions ofrece `HierarchicalMachine`.[84] Si tu problema es explosión de estados *dentro* de una estrategia determinista, la respuesta es un statechart, no un grafo de agentes.

8.3 El salto conceptual: una FSM es un grafo dirigido etiquetado

8.3.1 FSM ⊂ grafos: tres restricciones que LangGraph relaja

Formalmente, un autómata finito **es** un grafo dirigido etiquetado: los vértices son los estados y las aristas tienen la forma $q \to \delta(q, x)$.[85] Lo que llamamos FSM es un grafo con tres restricciones: la función de transición δ es una tabla fija definida en tiempo de diseño (Samek: la topología debe ser "static and fixed at compile time"[81]), las aristas son fijas, y el estado activo es un token atómico — un valor de un enum.

LangGraph relaja exactamente las tres. Su modelo son tres primitivas: un `State` compartido tipado, `Nodes` que son funciones, y `Edges` que "determine which `Node` to execute next based on the current state".[86] Primero, el *routing* se calcula en runtime: `add_conditional_edges(nodo, routing_fn)` registra una función Python arbitraria — δ se *computa*, no se *tabula*, y puede incluso consultar un LLM. Segundo, los nodos pueden contener LLMs o "good ol' code".[86] Tercero, el estado deja de ser un enum para ser un objeto tipado (`TypedDict`/Pydantic) con *reducers* por campo que fusionan las actualizaciones parciales de cada nodo.[87] Y los checkpointers persisten un snapshot tras cada super-step con `thread_id`, time travel y tolerancia a fallos[88]: es, interpretación nuestra, el journaling que construimos en los capítulos 4 y 7, industrializado.



El diagrama anterior es nuestra FSM de esta guía vista con ojos de grafo: cada estado es un nodo, cada evento etiqueta una arista. Si lees `SCANNING → PENDING_ENTRY : setup_validado` como "el nodo `SCANNING` enruta al nodo `PENDING_ENTRY` cuando la función de routing devuelve `setup_validado`", ya tienes el modelo mental completo de un `StateGraph`.

8.4 Lo que viene en la serie

8.4.1 El puente es oficial, y ya hay trading multiagente sobre él

No estamos forzando una analogía. En el post de lanzamiento de LangGraph (enero de 2024), el equipo de LangChain escribe: "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] Además,

en su taxonomía de arquitecturas cognitivas, *State Machine* (nivel 5: el código decide qué pasos tomar, con ciclos) es el peldaño inmediatamente anterior a *Agent* (nivel 6: el LLM decide).[89] Diseñar una FSM es, literalmente, el prerrequisito mental para diseñar un agente.

La evidencia en trading ya existe. TradingAgents (arXiv 2412.20138, UCLA + MIT) implementa una firma de trading multiagente — analistas, debate Bull/Bear, trader, comité de riesgo, gestor de fondos — construida sobre LangGraph con un `AgentState` tipado y funciones `should_continue_debate` que inspeccionan el estado y devuelven el siguiente nodo:[90][91] es $\delta(s) \to s'$ en forma pura, con un contador de rondas haciendo de guarda exactamente como en nuestros ejemplos.

Dimensión	FSM clásica (esta serie)	StateGraph de LangGraph
Qué es un "estado"	Valor de un enum + contexto auxiliar	Objeto tipado completo (`TypedDict`/Pydantic); los modos son nodos
Función de transición δ	Tabla fija definida en diseño	Función de routing evaluada en runtime; puede consultar un LLM
Determinismo	Determinista y reproducible	Determinista si nodos y routing son código puro; estocástico con LLMs
Actualización del contexto	El handler muta el contexto en la transición	El nodo devuelve un delta parcial; los reducers lo fusionan
Ejecución	Un evento → una transición, secuencial	Super-steps Pregel: nodos en paralelo sobre snapshot inmutable
Persistencia	Journaling manual de transiciones	Checkpointers con `thread_id`, time travel, fault tolerance
Pausa humana	Estado explícito `AWAITING_*` + evento manual	`interrupt()` dinámico + `Command(resume=...)`

Tabla de elaboración propia a partir de la documentación oficial citada. La lectura importante no es que una columna sustituya a la otra: la ejecución de cada orden individual seguirá siendo una FSM determinista y auditable, porque ahí el determinismo es un requisito regulatorio, no una limitación. Lo que el StateGraph añade es la capa de orquestación superior — debates entre analistas, comités de riesgo, aprobaciones humanas — donde el routing dinámico y el paralelismo de super-steps sí aportan. La tabla también muestra que casi todo concepto de esta guía tiene traducción directa: quien domina la columna izquierda ya entiende el 80 % de la derecha.

En la próxima serie de quantarmy.com en pythonparatrading.com haremos esa traducción paso a paso: reimplementaremos la máquina de estados de esta guía como un `StateGraph` con routing determinista, añadiremos checkpointers como journaling, y culminaremos con un sistema multiagente de análisis y riesgo al estilo de TradingAgents — con la ejecución de órdenes delegada, como debe ser, en una FSM explícita.

Referencias

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 (16 de octubre de 2013)
2. SEC, Order 34-70694, ¶18 — https://www.sec.gov/files/litigation/admin/2013/34-70694.pdf (16 de octubre de 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 (16 de octubre de 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/ (marzo de 2020)
7. FINRA, "Guardrails for Market Volatility" — https://www.finra.org/investors/insights/guardrails-market-volatility (26 de noviembre de 2024)
8. SEC-CFTC, "Findings Regarding the Market Events of May 6, 2010" — https://www.sec.gov/files/marketevents-report.pdf (30 de septiembre de 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 (7 de julio de 2017)
11. arXiv:2601.07525v1 — Def. 1 (cita 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, ago-2022), §3.3.2; sucesor normativo de RFC 793 (sep-1981) — https://datatracker.ietf.org/doc/html/rfc9293
13. arXiv:2603.28903v1 — §II-A (δ: Q×Σ→2^Q, NFA); unicidad del estado alcanzado en arXiv:2605.28408v1 — https://arxiv.org/html/2603.28903v1 ; https://arxiv.org/html/2605.28408v1
14. arXiv:2604.11567v1 — §2, autómata determinista (a lo sumo un sucesor) y completo (al menos uno) — https://arxiv.org/html/2604.11567v1
15. AcademiaLab — "Autómata finito determinista", sección "Completo e incompleto" — https://academia-lab.com/enciclopedia/automata-finito-determinista/
16. Springer — "The Regular Languages" (2025), referencias: 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): misma base de código para backtest y live — https://www.quantstart.com/articles/Event-Driven-Backtesting-with-Python-Part-I/
21. TutorialsPoint — "Deterministic Finite Automaton" (reducción: eliminar estados inalcanzables desde q0) — https://www.tutorialspoint.com/automata_theory/deterministic_finite_automaton.htm

22. University of Mississippi, CSci 311 (basado en 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 — documentación oficial, "Transitions" — https://python-statemachine.readthedocs.io/en/latest/transitions.html (v3.2.0)
27. python-statemachine — documentación oficial, "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" (transiciones internas con `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/ (s/f)
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 — documentación oficial, "Plugins → Protections" (`CooldownPeriod`) — https://www.freqtrade.io/en/stable/plugins/ (consultada 2026)
35. Trading Blox — Users Guide, "Turtle System Rules" (parámetro "Trade if Last is Winner") — https://www.tradingblox.com/Manuals/UsersGuideHTML/turtlesystem.htm (s/f)
36. Build a Finite State Machine in Python — https://belderbos.dev/blog/build-finite-state-machine-python/ (2026-04-07)
37. transitions — README oficial (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, sección "Check transitions" (`may_<trigger>()` / `may_trigger()`) — https://github.com/pytransitions/transitions
40. python-statemachine — repo y PyPI (v3.2.0, 2026-06-17; requiere Python ≥3.10) — https://github.com/fgmacedo/python-statemachine · https://pypi.org/project/python-statemachine/
41. python-statemachine — documentación "Conditions/Guards" (orden de declaración, `enabled_events()`) — https://python-statemachine.readthedocs.io/en/latest/guards.html
42. python-statemachine — documentación "Transitions" (`from_.any()`, `internal=True`) — https://python-statemachine.readthedocs.io/en/latest/transitions.html
43. Releases de fgmacedo/python-statemachine (3.0.0 StateCharts SCXML/W3C; 3.2.0 IO seguro YAML/SCXML/JSON) — https://github.com/fgmacedo/python-statemachine/releases
44. Issue #717: AsyncMachine ejecuta callbacks con `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 (espejo b2bits) — 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` (evento executionReport, campos x/X) — https://github.com/binance/binance-spot-api-docs/blob/master/user-data-stream.md
51. Alpaca — "Orders at Alpaca" (diagrama oficial del ciclo de vida de la orden) — https://docs.alpaca.markets/docs/orders-at-alpaca
52. NautilusTrader — "Execution" concepts (Overfills, deduplicación por 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 y nota al pie 7, 16-oct-2013 — https://www.sec.gov/files/litigation/admin/2013/34-70694.pdf
63. Freqtrade, documentación oficial, "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" (bloqueo <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` (rama develop, código verbatim) — 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 (vía 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* (glosario); UML 2.5.1 p. 309 — https://statecharts.dev/glossary/history-state.html
84. python-statemachine 3.2.0 — documentación oficial (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 (consultada 2026)
87. Use the graph API — Docs by LangChain — https://docs.langchain.com/oss/python/langgraph/use-graph-api (consultada 2026)
88. Persistence — Docs by LangChain — https://docs.langchain.com/oss/python/langgraph/persistence (consultada 2026)
89. Chase, H. et al. — *LangGraph* (post de lanzamiento, blog oficial LangChain) — 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