Índice

[📕 PARTE I • VISÃO GERAL, OBJETIVO E FRAMEWORK DAS LAYERS 5](#_Toc216076002)

[CAPÍTULO 1 • INTRODUÇÃO E PROPÓSITO DESTA EDIÇÃO 5](#_Toc216076003)

[CAPÍTULO 2 • A ESCADA INSTITUCIONAL (LAYERS 0 → 4) 10](#_Toc216076004)

[📕 PARTE II • FUNDAMENTOS INSTITUCIONAIS DO MERCADO 15](#_Toc216076005)

[CAPÍTULO 3 • COMO OS MERCADOS FINANCEIROS FUNCIONAM NA REALIDADE 15](#_Toc216076006)

[CAPÍTULO 4 • MICROESTRUTURA DE MERCADO EM PROFUNDIDADE 19](#_Toc216076007)

[CAPÍTULO 5 • ORDERFLOW INSTITUCIONAL 24](#_Toc216076008)

[📕 PARTE III • PREPARAÇÃO TÉCNICA DO PROJETO 30](#_Toc216076009)

[CAPÍTULO 6 • SETUP INSTITUCIONAL DE DESENVOLVIMENTO (PYTHON + VS CODE + ARQUITETURA BASE) 30](#_Toc216076010)

[CAPÍTULO 7 • ARQUITETURA PROFISSIONAL DE SOFTWARE PARA TRADING 36](#_Toc216076011)

[📕 PARTE IV • IBKR: A PORTA PARA O MUNDO INSTITUCIONAL 44](#_Toc216076012)

[CAPÍTULO 8 • ENTENDER O ECOSSISTEMA IBKR E PREPARAR O ACESSO INSTITUCIONAL 44](#_Toc216076013)

[CAPÍTULO 9 • CONSTRUIR O CONECTOR IBKR PROFISSIONAL (EClient + EWrapper) 50](#_Toc216076014)

[CAPÍTULO 10 • MODELOS DE DADOS INSTITUCIONAIS (TRADEEVENT, DOMEVENT, TICKEVENT, DELTAEVENT…) 56](#_Toc216076015)

[CAPÍTULO 11 • DESENHAR O EVENT BUS INSTITUCIONAL (PUB/SUB INTERNO) 65](#_Toc216076016)

[📕 PARTE V • ORDERFLOW ENGINE 72](#_Toc216076017)

[CAPÍTULO 12 • DOM ENGINE (LIVRO DE ORDENS INSTITUCIONAL) 72](#_Toc216076018)

[CAPÍTULO 13 • DELTA ENGINE (DELTA, CVD, SPEED OF TAPE) 77](#_Toc216076019)

[CAPÍTULO 14 • FOOTPRINT ENGINE (VOLUME POR NÍVEL, IMBALANCES, PONTO DE CONTROLO) 84](#_Toc216076020)

[CAPÍTULO 15 • SIGNAL ENGINE (ABSORÇÃO, SPOOFING, IMBALANCES, BURSTS, FALHA DE BREAKOUT) 91](#_Toc216076021)

[📕 PARTE VI • EXECUÇÃO PROFISSIONAL 97](#_Toc216076022)

[CAPÍTULO 16 • EXECUTION ENGINE (IBKR, MT5, MULTI-BROKER) 97](#_Toc216076023)

[Capítulo 17 – UI Institucional (Heatmap, Footprint, DOM, Delta, CVD) 103](#_Toc216076024)

[📕 PARTE VII — FEEDS E ABSTRAÇÕES 112](#_Toc216076025)

[Capítulo 18 — Data Provider Abstraction Layer (Multi-Feed Institucional) 112](#_Toc216076026)

[Capítulo 19 — Execution Abstraction Layer (IBKR, MT5, Dual Execution) 120](#_Toc216076027)

[Capítulo 20 — Price Normalization & Cross-Market Mapping (GC → XAUUSD para MT5) 126](#_Toc216076028)

[Capítulo 21 — Multi-Broker Architecture (IBKR + MT5 + Dual Execution Pipeline) 132](#_Toc216076029)

[📘 PARTE VIII — FEEDS PROFISSIONAIS 139](#_Toc216076030)

[Capítulo 22 — Rithmic Integration Layer (Bridge C# → Python + Orderflow Feed Profissional) 139](#_Toc216076031)

[Capítulo 23 — dxFeed Integration Layer (WebSocket Institucional + Histórico e Replay Avançado) 145](#_Toc216076032)

[📘 PARTE IX — MOTORES INSTITUCIONAIS AVANÇADOS 153](#_Toc216076033)

[Capítulo 24 — Historical Reconstruction Engine (DOM, Footprint, Delta e T&S a partir de dados brutos dxFeed) 153](#_Toc216076034)

[Capítulo 25 — Replay Engine (Modo Playback Institucional com DOM, Footprint e Delta em Tempo Real) 159](#_Toc216076035)

[📘 PARTE X — RISCO E GOVERNANÇA 165](#_Toc216076036)

[Capítulo 26 — Risk Engine Institucional (Daily DD, Exposure, Hard Limits, Kill Switch, Circuit Breakers) 165](#_Toc216076037)

[Capítulo 27 — Position Monitor & Trade Lifecycle (Gestão completa da posição em ambiente multi-broker) 170](#_Toc216076038)

[Capítulo 28 — Order Manager & Execution Controller 176](#_Toc216076039)

[Capítulo 29 — Latency Monitor & Throughput Management 182](#_Toc216076040)

[Capítulo 30 — Estratégias de Execução Algorítmica com Microestrutura (Tape + Orderflow) 186](#_Toc216076041)

[⭐ \*\*Capítulo 31 — Modelos Baseados em Microestrutura 192](#_Toc216076042)

[⭐ \*\*Capítulo 32 — Detecção de Padrões Institucionais 198](#_Toc216076043)

[⭐ \*\*Capítulo 33 — Machine Learning para Microestrutura 204](#_Toc216076044)

[Capítulo 33 — Machine Learning para Microestrutura (Transformers, Sequence Models, Prediction Engines) 205](#_Toc216076045)

[⭐ \*\*Capítulo 34 — Feature Engineering Institucional 211](#_Toc216076046)

[Capítulo 35 Dataset Builder Institucional 216](#_Toc216076047)

[📘 PARTE XI — MODELOS E PIPELINES AVANÇADOS 226](#_Toc216076048)

[⭐ \*\*Capítulo 36 — Training Pipeline Institucional 226](#_Toc216076049)

[📘 PARTE XII — DEPLOY, INFRA, REDUNDÂNCIA E PRODUÇÃO 233](#_Toc216076050)

[⭐ \*\*Capítulo 37 — Deploy Institucional 233](#_Toc216076051)

[📘 PARTE XIII — GOVERNANÇA, SEGURANÇA E CONTROLO 242](#_Toc216076052)

[⭐ \*\*Capítulo 38 — Segurança e Governança 242](#_Toc216076053)

[📘 PARTE XIV — PROFISSIONALIZAÇÃO 249](#_Toc216076054)

[⭐ \*\*Capítulo 39 — Caminho para Profissionalização 249](#_Toc216076055)

[📦 APÊNDICE A – GLOSSÁRIO INSTITUCIONAL COMPLETO 255](#_Toc216076056)

[📦 APÊNDICE B - Esquemas UML, Mermaid e PlantUML 263](#_Toc216076057)

[B.2 - Diagramas Mermaid de Arquitetura 263](#_Toc216076058)

[B.3 - Diagramas UML em PlantUML 268](#_Toc216076059)

[B.4 - Recomendações finais de uso 269](#_Toc216076060)

[📦 APÊNDICE C – CHECKLISTS PROFISSIONAIS 269](#_Toc216076061)

[🟥 C.1 – Checklist de Preparação da Sessão (Pré-Mercado) 269](#_Toc216076062)

[🟧 C.2 – Checklist de Microestrutura (Durante Sessão) 270](#_Toc216076063)

[🟨 C.3 – Checklist de Execução (Antes de enviar qualquer ordem) 271](#_Toc216076064)

[🟩 C.4 – Checklist Diário do Risk Officer 272](#_Toc216076065)

[🟦 C.5 – Checklist de Deploy Institucional (Nova versão) 272](#_Toc216076066)

[🟫 C.6 – Checklist de Segurança e Governança 273](#_Toc216076067)

[🟪 C.7 – Checklist do Replay Engine 273](#_Toc216076068)

[🟧 C.8 – Checklist de Treino de Modelos ML 274](#_Toc216076069)

[🟨 C.9 – Checklist de Final de Sessão 274](#_Toc216076070)

[📦 APÊNDICE D – TEMPLATES PROFISSIONAIS (YAML, .env, ML, RISCO, EXECUÇÃO, PROVIDERS) 275](#_Toc216076071)

[📦 APÊNDICE E – TABELAS TÉCNICAS INSTITUCIONAIS 283](#_Toc216076072)

[🟥 E.1 – Tabela de Regimes de Volatilidade (Volatility Regimes) 283](#_Toc216076073)

[🟧 E.2 – Tabela de Imbalances (Desequilíbrios Bid/Ask) 283](#_Toc216076074)

[🟨 E.3 – Tabela de Padrões Institucionais (Pattern Detector) 283](#_Toc216076075)

[🟩 E.4 – Tabela de Delta e CVD 284](#_Toc216076076)

[🟦 E.5 – Tabela de Speed of Tape (SoT) 285](#_Toc216076077)

[🟫 E.6 – Tabela de DOM (Depth of Market) 285](#_Toc216076078)

[🟥 E.7 – Tabela de Execução (IBKR + MT5) 286](#_Toc216076079)

[🟧 E.8 – Tabela de Risco – Safety Layer 287](#_Toc216076080)

[🟨 E.9 – Tabela de Qualidade de Feed 287](#_Toc216076081)

[🟩 E.10 – Tabela de ML – Thresholds Operacionais 287](#_Toc216076082)

[🟦 E.11 – Tabela de Modelos ML – Comparação Rápida 288](#_Toc216076083)

[🟪 E.12 – Tabela de Parâmetros de Footprint 288](#_Toc216076084)

[🟫 E.13 – Tabela de Conversão Futuro → CFD (GC → XAUUSD) 288](#_Toc216076085)

[📦 APÊNDICE F – EXEMPLOS REAIS E ESTUDOS DE CASO INSTITUCIONAIS 289](#_Toc216076086)

[🟥 F.1 – Absorção em Máximos (Reversão Direcional) 289](#_Toc216076087)

[🟧 F.2 – Spoofing no DOM (Manipulação de Curto Prazo) 290](#_Toc216076088)

[🟨 F.3 – Delta Divergence em Continuação 291](#_Toc216076089)

[🟩 F.4 – Burst + Pullback (Entrada Institucional Oculta) 292](#_Toc216076090)

[🟦 F.5 – Failed Breakout (Quebra Falsa com Reversão Forte) 293](#_Toc216076091)

[🟫 F.6 – Liquidity Backfill (Reposição de Liquidez) 294](#_Toc216076092)

[🟥 F.7 – Case Study Completo: Entrada SELL Institucional 295](#_Toc216076093)

[🟧 F.8 – Case Study Completo: Entrada BUY Institucional 296](#_Toc216076094)

[🟨 F.9 – Cenário Negativo: Quando o Bot NÃO Deve Entrar 297](#_Toc216076095)

[🟪 F.10 – Exemplo Completo: Mapeamento GC → XAUUSD 297](#_Toc216076096)

[📦 APÊNDICE G – MindMaps 298](#_Toc216076097)

[🌳 MAPA MENTAL 1 — MICROESTRUTURA (MERMAID) 298](#_Toc216076098)

[🌳 MAPA MENTAL 2 — ARQUITETURA DO BOT (MERMAID) 298](#_Toc216076099)

[🌳 MAPA MENTAL 3 — PIPELINE DE DECISÃO (MERMAID) 299](#_Toc216076100)

[🌳 MAPA MENTAL 4 — UI INSTITUCIONAL (MERMAID) 299](#_Toc216076101)

[🌳 MAPA MENTAL 5 — PIPELINE DE ML (MERMAID) 300](#_Toc216076102)

[📦 APÊNDICE H – Fluxos UML 300](#_Toc216076103)

## 📕 PARTE I • VISÃO GERAL, OBJETIVO E FRAMEWORK DAS LAYERS

### CAPÍTULO 1 • INTRODUÇÃO E PROPÓSITO DESTA EDIÇÃO

Esta segunda edição tem como objetivo formar o leitor para operar, analisar e desenvolver tecnologia na camada mais avançada possível fora do ambiente bancário, atingindo a **Layer 2.5**, a zona intermédia entre o domínio completo da microestrutura (Layer 2) e os princípios fundamentais da execução algorítmica profissional (Layer 3).

Este capítulo estabelece o enquadramento conceptual da obra, explica porque esta edição é necessária, apresenta a diferença entre visão retail e institucional, define a meta técnica, clarifica limites e descreve como utilizar o livro para maximizar resultados.

#### 1. PORQUE ESTA EDIÇÃO EXISTE

A primeira edição focou-se em price action institucional, estrutura do mercado e automação através do MT5.  
Esse conhecimento foi suficiente para sair do mundo retail tradicional, mas **não permite observar o mercado real**.  
O leitor passou a compreender liquidez e comportamento do preço, porém continuou limitado à superfície da ação do mercado.

A Layer 2 (*ex:MT5*), a camada da microestrutura, requer acesso e interpretação de dados que o universo retail típico não vê:

• fluxo de ordens  
• agressão vs passividade  
• movimentação interna do livro  
• delta  
• absorção  
• spoofing  
• times & sales  
• depth of market  
• footprints de volume  
• velocidade do fluxo

Esta segunda edição existe para fornecer ao leitor estas ferramentas e capacidades, permitindo-lhe operar no mesmo paradigma cognitivo e técnico que profissionais de prop firms e mesas de trading de bancos.

Existem três motivos centrais para esta edição existir:

**➤ 1) O retail não vê nem compreende dados reais**

Os instrumentos mais populares do mundo retail (TradingView, MT4, MT5 e CFDs) **não fornecem dados institucionais**. Não exibem o fluxo verdadeiro do mercado, mas sim reconstruções artificiais derivadas de um feed simplificado.

Sem dados reais o trader fica preso a representações secundárias.  
Velas e indicadores são sempre efeitos, nunca causas.

**➤ 2) O trader que domina apenas Layers 0 e 1 nunca conseguirá pensamento institucional**

Layer 0 corresponde aos indicadores e price action básico.  
Layer 1 corresponde ao price action institucional (SMC/ICT).

Ambas as camadas são úteis, mas insuficientes.  
O trader institucional trabalha com fluxos, não com padrões.

**➤ 3) Nunca foi disponibilizado ao público um guia completo para criar uma aplicação institucional real**

O conhecimento existe em ambientes restritos, dentro de bancos e mesas de negociação.  
O objetivo desta edição é criar a ponte entre esse mundo e o leitor individual, através de:

• dados reais via IBKR  
• construção completa de uma aplicação  
• motor profissional de orderflow  
• interface gráfica institucional  
• execução opcional  
• telemetria  
• processos de engenharia profissional

Nenhum livro junta tudo isto.  
Até agora.

#### 2. VISÃO INSTITUCIONAL VS VISÃO RETAIL

A diferença entre estas duas realidades é profunda.  
Não se trata apenas de conhecimento adicional, mas de paradigmas completamente diferentes.

**➤ A visão retail**

O trader retail interpreta o mercado através de elementos derivados:

• candles  
• indicadores  
• zonas de liquidez  
• suportes e resistências  
• padrões visuais  
• estrutura do preço  
• divergências  
• moving averages  
• sinais de confluência

O retail observa **efeitos**.  
Acredita que está a ler o mercado, mas está apenas a consumir imagens simplificadas.

Tudo o que o retail vê é consequência.

**➤ A visão institucional**

O trader institucional interpreta **causas** diretamente a partir do fluxo:

• quem está a vender e quem está a comprar  
• quem bate no ask e quem bate no bid  
• onde há tamanho escondido  
• onde há absorção  
• quais níveis estão a ser defendidos  
• quão rápido o tape acelera  
• se há spoofing ou manipulação  
• se a liquidez sobe ou recua  
• qual lado está a exercer pressão  
• qual lado está a desistir

A análise institucional é feita **no tempo presente**.  
Não depende de candles ou indicadores.  
Depende do comportamento interno do mercado.

É esta transição cognitiva que o leitor fará ao longo deste livro.

#### 3. O OBJETIVO: ATINGIR A LAYER 2.5

A Layer 2 corresponde ao domínio da microestrutura e orderflow.  
A **Layer 3** corresponde à execução institucional, FIX-style, algoritmos de routing, **infraestruturas de baixa latência e mecanismos que apenas bancos possuem.**

O objetivo desta edição é conduzir o leitor até à **Layer 2.5**, que inclui:

**✓ Microestrutura completa**

Delta, CVD, footprint, DOM, times & sales, manipulação de liquidez, acções algorítmicas.

**✓ Execução algorítmica sólida**

Ordens inteligentes, gestão de risco, lógica de controlo de posição e validação institucional.

**✓ Sem necessidade de hardware especial ou acessos bancários**

Tudo construído com Python, IBKR e PySide6.

**✓ Nível profissional e aplicável na vida real**

É o nível esperado de um Quant Junior ou Trader Institucional Júnior que inicia carreira numa mesa de execução.

Esta edição não pretende ensinar estratégias de “melhorar trades”.  
Pretende ensinar **como funciona o mercado real** e como construir ferramentas institucionais com capacidade de leitura avançada e execução fiável.

#### 4. O QUE VAMOS CONSTRUIR PASSO A PASSO

Esta edição não é conceptual.  
É um projeto completo.

O leitor irá construir:

**➤ A) Um conector profissional IBKR**

Com capacidade para:

• processamento de tick-by-tick  
• leitura de DOM  
• receção de Times & Sales  
• identificação de agressão  
• reconexão automática  
• buffers seguros  
• logging estruturado  
• distribuição interna via event bus  
• subscrição de futuros CME (GC, ES, NQ, 6E)

**➤ B) Um Orderflow Engine institucional**

Este módulo analisará o fluxo em tempo real:

• cálculo de delta bid/ask  
• agregadores temporais  
• CVD  
• footprint engine  
• DOM engine  
• deteção de absorção  
• spoof detection  
• desequilíbrios 2:1, 3:1, 4:1  
• velocidade do tape  
• bursts de fluxo  
• criação de sinais internos

**➤ C) Uma interface gráfica profissional**

Uma aplicação completa construída em PySide6, com:

• painéis de DOM  
• heatmap  
• footprint  
• gráfico de delta  
• Times & Sales  
• navegação fluida  
• zoom e scrolling  
• tema dark institucional  
• sincronização entre componentes  
• alto desempenho

**➤ D) Execução algorítmica opcional**

Para os leitores que pretendem automatizar decisões:

• ordens market, limit e stop  
• bracket orders  
• validação de risco  
• regras de execução  
• gestão de posições  
• logging de execução ao estilo FIX

**➤ E) Telemetria e datasets**

Criando:

• gravação de DOM  
• gravação de trades  
• gravação de delta  
• datasets históricos  
• ferramentas de replay  
• logs estruturados

**➤ F) Deploy real**

Com:

• IB Gateway  
• watchdog scripts  
• mini-PC dedicado  
• backups  
• procedimentos de fiabilidade  
• execução contínua

#### 5. O QUE ESTA EDIÇÃO NÃO PODE ENSINAR

Algumas camadas só são acessíveis através de instituições financeiras devido a limitações legais, custos avassaladores e necessidades de infraestrutura.

Não poderão ser ensinados neste livro:

❌ colocation em datacenters financeiros  
❌ FIX engines industriais proprietários  
❌ Smart Order Routing multi-venue real  
❌ HFT (operar em microsegundos)  
❌ FPGA ou kernel bypass  
❌ acesso a dark pools  
❌ feeds institucionais privados

Este livro alcança o limite técnico permitido fora desses ambientes: **Layer 2.5**.

#### 6. COMO LER ESTE LIVRO

Este manual funciona como um programa de formação progressivo.

**▸ Passo 1: Ler os capítulos na ordem**

Cada secção depende do conhecimento acumulado anteriormente.

**▸ Passo 2: Escrever todo o código à mão**

Copiar impedirá a transformação técnica pretendida.

**▸ Passo 3: Testar cada módulo isoladamente**

Primeiro o conector, depois o engine, depois a interface, depois a execução.

**▸ Passo 4: Integrar apenas quando tudo estiver sólido**

A união final só ocorre quando cada componente estiver funcional e robusto.

**▸ Passo 5: Criar a versão pessoal da aplicação**

O leitor adaptará cores, funcionalidades, estratégias e módulos.

**▸ Passo 6: Observar dados reais**

A app não é teórica.  
Funciona com mercado real em tempo real.

### CAPÍTULO 2 • A ESCADA INSTITUCIONAL (LAYERS 0 → 4)

O mercado financeiro não é um ambiente homogéneo.  
Existem camadas de acesso, conhecimento, tecnologia e velocidade totalmente diferentes entre si.  
Cada camada corresponde a um nível de **acesso**, **capacidade técnica**, **informação** e **poder de execução**.

O trader retail e o trader institucional não estão a jogar o mesmo jogo.  
Nem sequer estão no mesmo campo.

Este capítulo explica, de forma clara e detalhada, **as cinco layers fundamentais**, como funcionam, quem opera nelas e onde este livro coloca o leitor.

#### 1. O QUE SÃO AS LAYERS DO MERCADO

As layers representam os níveis de profundidade e acesso dentro do ecossistema financeiro.

Cada layer tem:

• tipos de dados  
• capacidades técnicas  
• velocidade de execução  
• ferramentas  
• restrições  
• riscos  
• oportunidades

Quanto mais alta a layer, mais perto se está de **como o mercado realmente funciona**.

A Layer 0 é a camada superficial.  
A Layer 4 é o núcleo do mercado institucional avançado.

#### 2. LAYER 0 • RETAIL BÁSICO

**O mundo dos indicadores, vídeos de YouTube e ilusões estatísticas**

Esta é a camada onde se encontra a maioria absoluta dos traders do mundo.

Nesta layer o trader utiliza ferramentas que **não mostram o mercado real**, como:

• indicadores tradicionais (RSI, MACD, etc.)  
• setups visuais  
• padrões gráficos  
• cruzamentos de médias  
• sinais automáticos genéricos  
• estratégias baseadas em “regras fixas”

Características principais:

➤ Dados atrasados e derivados  
Os indicadores são cálculos sobre cálculos.  
Nada aqui representa o fluxo real.

➤ Ausência total de contexto institucional  
O trader não sabe quem está a comprar ou vender.

➤ Zero contacto com microestrutura  
DOM, delta, footprint ou T&S não existem nesta layer.

➤ Forte influência emocional  
A estratégia muda todos os dias.  
O trader não tem consistência conceptual.

A Layer 0 é a superfície.  
O acesso aqui é fácil, mas completamente inútil a longo prazo.

#### 3. LAYER 1 • PRICE ACTION INSTITUCIONAL (SMC/ICT)

**A primeira camada acima do ruído**

Nesta layer o trader começa a entender o comportamento do preço:

• liquidez  
• manipulação  
• FVG  
• BOS e CHOCH  
• zonas de interesse  
• mitigação  
• estrutura do mercado

É muito superior à Layer 0 porque:

• identifica quem está a ser manipulado  
• entende a busca de liquidez  
• compreende “o porquê” do movimento

Mas ainda tem limitações importantes:

➤ O trader ainda só vê efeitos, não causas  
Mesmo com ICT/SMC, o trader só interpreta a consequência do que o fluxo fez.

➤ As operações são baseadas no gráfico  
E não no fluxo real.

➤ Não há leitura do comportamento interno da vela  
Toda a dinâmica entre bid/ask está oculta.

A Layer 1 é essencial, mas insuficiente para operar institucionalmente.

#### 4. LAYER 2 • MICROESTRUTURA E ORDERFLOW REAL

**Aqui começa o mercado verdadeiro**

Nesta layer o trader observa **o motor real** que gera as velas.

Ferramentas da Layer 2:

• DOM (Depth of Market)  
• Times & Sales  
• Delta  
• CVD  
• Footprint  
• Volume agressor e passivo  
• Liquidez resting  
• Absorção  
• Spoofing  
• Pulling & Stacking  
• Velocidade do tape

Nesta layer o trader já não adivinha o que está a acontecer:  
ele **vê**.

Características da Layer 2:

➤ A leitura é baseada em comportamento, não em padrões  
O trader observa ações, não desenhos.

➤ A análise é no tempo presente  
Enquanto a Layer 1 observa o passado recente, a Layer 2 observa o agora.

➤ O trader vê intenção, não apenas resultado  
Consegue distinguir quem está a controlar o mercado.

Esta layer representa 80% do conhecimento necessário para operar como profissional.

#### 5. LAYER 2.5 • EXECUÇÃO ALGORÍTMICA PROFISSIONAL

**A camada máxima possível fora de um banco**

Esta é a camada alvo deste livro.  
É o nível atingível com Python, IBKR, dados reais, acesso doméstico e software profissional.

Capacidades desta layer:

➤ Construção de aplicações completas  
DOM, footprint, delta, T&S, gráficos institucionais.

➤ Orderflow Engine  
Processamento avançado de fluxo para gerar insights e sinais.

➤ Execução algorítmica sólida  
Ordens limit, market, stop e bracket com validação de risco.

➤ Arquitetura event-driven  
O sistema reage a eventos de mercado como sistemas institucionais.

➤ Telemetria e datasets  
Gravação do fluxo para análise posterior (replay engine).

➤ UI institucional  
Interface ao nível de Bookmap, Jigsaw ou ATAS.

Representa a ponte entre o mundo retail avançado e o mundo institucional verdadeiro.

#### 6. LAYER 3 • EXECUÇÃO INSTITUCIONAL E ROUTING AVANÇADO

**O domínio dos bancos, prop firms e market makers**

O trader nesta layer utiliza ferramentas e infraestruturas que o público não consegue replicar.

Tecnologia da Layer 3:

• FIX engines industriais  
• Smart Order Routing (SOR)  
• acesso direto a múltiplas bolsas  
• latência sub-milisegundo  
• redes privadas  
• servidores dedicados  
• otimização de execução  
• gestão avançada da fila (queue position)

Esta layer é fechada.  
Só instituições têm acesso.

#### 7. LAYER 4 • HIGH-FREQUENCY TRADING E INFRAESTRUTURA ULTRA-BAIXA LATÊNCIA

**A camada que move o mercado global**

A Layer 4 é o território das empresas HFT como Jump Trading, Hudson River, Citadel Securities, DRW, Tower Research e outras.

Tecnologias e requisitos:

• FPGA  
• kernel bypass  
• trading em nanosegundos  
• colocação física no data center da bolsa  
• acesso direto ao matching engine  
• linhas dedicadas  
• otimização eletrónica avançada  
• equipas multidisciplinares de engenharia

Nenhum trader fora de um banco ou empresa HFT chega a esta layer.  
Nem é necessário para ser lucrativo.

#### 8. EM QUE LAYER O LEITOR VAI FICAR APÓS ESTE LIVRO?

O leitor terminará este livro com competências claras de:

✓ Layer 2 (microestrutura completa)  
✓ Layer 2.5 (execução algorítmica + aplicação profissional)

O leitor ficará no patamar necessário para:

• ser contratado como quant junior  
• criar ferramentas internas para fundos ou mesas de trading  
• operar com leitura profissional do mercado  
• automatizar fluxos reais  
• compreender como o mercado se move  
• deixar de depender de price action

Este livro leva o leitor ao nível mais elevado possível **sem acesso bancário**.

#### 9. POR QUE ESTE CAPÍTULO É ESSENCIAL?

Porque todas as escolhas de arquitetura, software, design, execução e análise que faremos ao longo deste livro são fundamentadas nestas layers.

O leitor agora sabe:

• onde está  
• para onde vai  
• o que é possível  
• o que não é possível  
• quais ferramentas fazem sentido  
• qual o destino final desta edição

Sem esta estrutura conceptual, qualquer tentativa de criar um bot institucional resultaria num sistema incoerente, mal desenhado ou simplesmente inútil.

## 📕 PARTE II • FUNDAMENTOS INSTITUCIONAIS DO MERCADO

### CAPÍTULO 3 • COMO OS MERCADOS FINANCEIROS FUNCIONAM NA REALIDADE

Para compreender microestrutura, orderflow e execução algorítmica, é necessário corrigir primeiro uma ilusão comum: **o mercado que o trader retail vê não é o mercado real**.

O mercado real é um sistema altamente organizado, hierárquico, técnico e regulado, composto por:

• exchanges reais  
• order books  
• matching engines  
• provedores de liquidez  
• algoritmos institucionais  
• fluxos eletrónicos  
• camadas de routing  
• market makers  
• HFTs  
• brokers e clearing houses

Nada disto é visível no TradingView ou no MT5.

Este capítulo explica com profundidade como o mercado realmente funciona.

#### 1. O QUE É UM MERCADO FINANCEIRO?

Um mercado financeiro é um sistema **order-driven**, movido por ordens reais de compra e venda, introduzidas por participantes institucionais e retail.

O preço NÃO é decidido por previsões, indicadores, notícias ou sentimento.  
O preço é decidido por:

✓ ordens que entram  
✓ ordens que são retiradas  
✓ ordens que são executadas  
✓ ordens que consomem liquidez

O mercado é um **sistema de fila**, governado por prioridades, profundidade e tempo.

Elementos essenciais:

• compradores e vendedores (bid e ask)  
• profundidade do livro (DOM)  
• fluxos agressivos e passivos  
• matching engine  
• liquidez resting  
• liquidez oculta  
• ordens algorítmicas

Tudo isto ocorre milhares de vezes por segundo.

#### 2. PORQUE FOREX SPOT NÃO REPRESENTA O MERCADO REAL

É crucial entender que **Forex Spot não é uma exchange centralizada**.  
Não existe um "livro de ordens universal" no Forex.

Principais características do Forex Spot:

• mercado descentralizado  
• múltiplos provedores de liquidez privados  
• livros independentes  
• preços fragmentados  
• volume não real  
• ausência de DOM universal  
• ausência de Times & Sales verdadeiro  
• brokers que sintetizam dados

O que o trader vê num broker é uma **versão artificial** do mercado interbancário.

**O resultado é simples:**

✓ sem ordem real  
✓ sem volume real  
✓ sem footprint  
✓ sem delta  
✓ sem DOM  
✓ sem microestrutura  
✓ sem orderflow verdadeiro

Por isso, esta edição usa **futuros CME via IBKR**, que são 100% transparentes.

#### 3. PORQUE FUTUROS (CME) SÃO O PADRÃO INSTITUCIONAL

As bolsas de futuros, como CME (Chicago Mercantile Exchange), oferecem:

✓ volume real  
✓ delta real  
✓ order book verdadeiro  
✓ Times & Sales verdadeiro  
✓ matching engine central  
✓ regras de prioridade FIFO  
✓ dados consistentes para todos  
✓ microestrutura transparente

Cada contrato futuro tem um único livro de ordens que todos os participantes veem.

O mercado que vamos estudar nesta edição é o dos futuros, especialmente:

• GC (ouro)  
• ES (S&P 500)  
• NQ (Nasdaq)  
• 6E (Euro futures)

São mercados extremamente líquidos e altamente representativos da microestrutura institucional.

#### 4. COMO FUNCIONA UM ORDER BOOK (DOM)

O DOM é a estrutura central do mercado.  
É onde vivem as ordens limit de todos os participantes.

O DOM mostra:

• níveis de preço  
• quantidade disponível para comprar  
• quantidade disponível para vender  
• profundidade da liquidez  
• alterações instantâneas pelas ordens das instituições

O DOM é organizado em duas metades:

• BID (interesse em comprar)  
• ASK (interesse em vender)

Cada lado contém múltiplos níveis de preço e quantidades.

Quando alguém envia uma ordem **limit**, ela entra no DOM.  
Quando alguém envia uma ordem **market**, ela consome o DOM.

#### 5. COMO AS ORDENS REALMENTE MOVEM O MERCADO

O movimento do preço é causado por **agressão**, não por intenção.

Existem dois tipos de fluxo:

**➤ Ordens Passivas**

São ordens que descansam no DOM:

• limit buy  
• limit sell

Elas fornecem liquidez mas **não movem o mercado**.

**➤ Ordens Agressivas**

São ordens que atravessam o livro:

• market buy (bate no ask)  
• market sell (bate no bid)

Estas ordens **consomem liquidez** e movem o preço.

Regra fundamental:

**✓ Só ordens agressivas movem o preço.**

**✓ Ordens passivas absorvem agressão e criam zonas de defesa.**

#### 6. O MATCHING ENGINE • O “CÉREBRO” DO MERCADO

O matching engine é o sistema eletrónico que:

• recebe ordens  
• ordena as ordens limit por preço e tempo  
• executa ordens market contra o livro  
• gere prioridades e posições na fila  
• mantém a integridade do mercado

A lógica padrão é FIFO (First In, First Out):

• primeiro a colocar a ordem  
• primeiro a ser executado

Algumas bolsas usam Pro-Rata, mas CME usa FIFO.

#### 7. O PAPEL DOS PARTICIPANTES DO MERCADO

Cada tipo de participante executa ações diferentes:

**➤ Retail (Layer 0 e 1)**

• entra tarde  
• opera sem informação real  
• usa derivativos visuais

**➤ Hedge Funds e grandes institucionais**

• usam ordens algorítmicas para disfarçar intenções  
• gerem risco institucional  
• executam ordens grandes com o menor impacto

**➤ Market Makers**

• fornecem liquidez  
• estabilizam o livro  
• usam modelos probabilísticos

**➤ HFTs (Layer 4)**

• arbitragens  
• market making automatizado  
• execução em microssegundos

**➤ Prop Traders (Layer 2 e 2.5)**

• analisam microestrutura  
• operam com mistura de execução algorítmica e discricionária

#### 8. O CICLO REAL DO PREÇO

O preço só tem três comportamentos reais:

**➤ 1) Pressão Agressiva**

Ordens market puxam o preço numa direção.

**➤ 2) Absorção**

Ordens passivas travam ou invertem esse movimento.

**➤ 3) Recuo de Liquidez**

Retirada de ordens passivas permite aceleração.

Não existe mais nada.  
Todos os padrões visuais derivam destes mecanismos.

#### 9. O PAPEL DA LIQUIDEZ

Liquidez é o combustível do mercado.

• Onde há liquidez resting, há zonas de defesa  
• Onde há falta de liquidez, há aceleração  
• Onde há spoofing, há intenção falsa  
• Onde há absorção, há inversões  
• Onde há desequilíbrio, há deslocação agressiva

A leitura correta da liquidez permite prever **o que vai acontecer**, não apenas interpretar o que aconteceu.

#### 10. O QUE ISTO SIGNIFICA PARA TI

Compreender este capítulo permite-te:

✓ ler o mercado como um profissional  
✓ abandonar ilusões de indicadores  
✓ interpretar o fluxo em tempo real  
✓ perceber porque o preço se move  
✓ antecipar absorções, breakouts reais ou falsos  
✓ entender a lógica dos grandes players  
✓ analisar o comportamento do livro

É a base da microestrutura.  
Sem este capítulo, não é possível avançar para delta, footprint, DOM ou execução avançada.

### CAPÍTULO 4 • MICROESTRUTURA DE MERCADO EM PROFUNDIDADE

A microestrutura é o estudo de **como ordens reais interagem dentro do livro para formar cada movimento do preço**.  
Enquanto a análise técnica tradicional observa o resultado (as velas), a microestrutura observa os mecanismos internos que geram esse resultado.

Compreender microestrutura significa ver:

• intenção  
• agressão  
• defesa  
• desequilíbrios  
• manipulação  
• aceleração  
• absorção  
• enfraquecimento  
• colapso de liquidez

Depois de dominar este capítulo, o leitor nunca mais olhará para um gráfico da mesma forma.

#### 1. ESTRUTURA FUNDAMENTAL DO MERCADO: BID, ASK E MIDPRICE

O mercado é um sistema organizado em torno de dois valores fundamentais:

• BID → melhor preço onde existe alguém disposto a comprar  
• ASK → melhor preço onde existe alguém disposto a vender

Entre eles existe o MIDPRICE, que representa o ponto central.

A microestrutura é governada pela dinâmica entre estes três pontos.

Princípios fundamentais:

➤ Um trade só acontece quando alguém aceita o preço do outro lado  
➤ O preço não se move de forma espontânea  
➤ O BID e ASK são a “parede frontal” da liquidez  
➤ Quem inicia a transação é sempre o agressor  
➤ Quem fornece liquidez é sempre o passivo

A agressão e a passividade constituem o equilíbrio primordial da microestrutura.

#### 2. QUEUE POSITION • A FILA INVISÍVEL QUE DECIDE TUDO

Todas as ordens limit que entram no DOM são colocadas numa fila ordenada por:

1. preço
2. tempo de chegada

Este mecanismo é chamado FIFO (First In, First Out).

A Queue Position é crítica porque determina:

• a probabilidade de execução  
• a prioridade sobre outros traders  
• o risco de ser ultrapassado por ordens mais rápidas  
• a capacidade de fornecer liquidez

A microestrutura é, em grande parte, um jogo de gestão de posição na fila.

Participantes institucionais utilizam algoritmos especializados para:

• entrar no livro sem revelar intenção  
• sair do livro ao primeiro sinal de risco  
• “esconder-se” dentro da fila  
• manipular o volume aparente

#### 3. LIQUIDEZ RESTING VS LIQUIDEZ AGRESSIVA

Existem dois tipos de liquidez fundamentais:

**Liquidez Resting**

São ordens limit colocadas no livro:

• não movem o preço  
• absorvem agressão  
• funcionam como barreiras  
• revelam intenção passiva

Quando uma grande quantidade de resting liquidity aparece num nível, é um sinal de defesa.

**Liquidez Agressiva**

São ordens market enviadas para consumir o livro:

• batem no ask  
• batem no bid  
• movem o preço  
• revelam urgência

Quanto maior a agressão, maior o deslocamento do preço.

Princípio essencial:

**Liquidez passiva controla.**

**Liquidez agressiva desloca.**

A microestrutura é o equilíbrio dinâmico destas duas forças.

#### 4. ABSORÇÃO • O SINAL INSTITUCIONAL MAIS IMPORTANTE

Absorção ocorre quando:

• grandes quantidades de ordens agressivas atingem um nível de preço  
• mas o preço não avança  
• porque existem ordens passivas suficientes para “segurar” o fluxo

Em linguagem simples:

➤ compradores agressivos batem no ask  
➤ vendedores passivos absorvem tudo  
➤ o preço não sobe  
➤ uma inversão pode estar iminente

O mesmo vale no sentido inverso.

Absorção indica:

• interesse institucional forte  
• defesa ativa de um nível  
• possível reversão ou manipulação  
• fadiga do lado agressor

É a base do tape reading avançado.

#### 5. SPOOFING • MANIPULAÇÃO DE LIQUIDEZ

Spoofing é quando uma instituição coloca grande liquidez resting que **não pretende executar**, apenas para influenciar o fluxo.

O spoofing funciona assim:

• são colocadas ordens grandes (muitas vezes falsas)  
• o livro parece forte num lado  
• o mercado reage psicologicamente  
• a liquidez falsa é removida no último momento  
• o preço desloca-se na direção oposta

O spoofing é **proibido por lei**, mas continua a ser usado por algoritmos sofisticados.

Características do spoofing:

• aparece de repente  
• aumenta drasticamente a liquidez aparente  
• desaparece rapidamente  
• altera a leitura do DOM  
• cria armadilhas para traders agressivos

O leitor aprenderá a detectá-lo mais adiante no livro.

#### 6. IMBALANCES • DESEQUILÍBRIOS CRÍTICOS DO FLUXO

Um imbalance ocorre quando há discrepância significativa entre agressão de compra e agressão de venda num mesmo nível ou candle.

Exemplos:

• muito mais volume a bater no ask do que no bid  
• níveis consecutivos de footprint com predominância clara  
• delta extremamente assimétrico

Indicadores de imbalance:

✓ footprint com razões de 2:1, 3:1 ou 4:1  
✓ delta extremamente positivo ou negativo  
✓ pressão unilateral consistente

Imbalance revela:

• tendência emergente  
• exaustão de um lado  
• oportunidade de continuação  
• potencial de breakout real

Irão ser usados na construção do motor de orderflow.

#### 7. PULLING & STACKING • MOVIMENTO DA LIQUIDEZ

Dinâmica crítica no DOM:

**Pulling**

Remoção súbita de liquidez resting.  
Indica:

• medo institucional  
• fuga de posição  
• aceleração iminente  
• desistência de defesa

**Stacking**

Aumento súbito de liquidez resting.  
Indica:

• defesa institucional  
• interesse claro no nível  
• possível absorção  
• travagem de movimento

Estas dinâmicas são sinais avançados utilizados por traders profissionais.

#### 8. DELTA E AGRESSÃO

Delta mede a diferença entre:

• volume batido no ask (compradores agressivos)  
• volume batido no bid (vendedores agressivos)

Delta positivo:

✓ compradores estão a controlar  
✓ pressão de compra maior

Delta negativo:

✓ vendedores estão a controlar  
✓ pressão de venda maior

Delta é o indicador mais puro do fluxo agressor.  
Será um pilar central do motor de orderflow.

#### 9. SPEED OF TAPE • VELOCIDADE DO FLUXO

Trata-se da velocidade a que trades são executados.

Alta velocidade:

• momentum  
• agressão coordenada  
• algorítmos ativos  
• breakout real

Baixa velocidade:

• indecisão  
• consolidação  
• ausência de interesse institucional

Instituições usam isto para identificar oportunidades e risco.

#### 10. MICROESTRUTURA É A LÍNGUA DOS INSTITUCIONAIS

A microestrutura é a base de:

• execução algorítmica  
• trading institucional  
• leitura profissional do mercado  
• compreensão real de tendências  
• identificação de reversões legítimas  
• entendimento de manipulações

Um leitor que domina microestrutura não depende de:

✘ indicadores  
✘ padrões  
✘ zonas marcadas  
✘ suposições visuais  
✘ narrativas emocionais

Ele lê:

✓ intenção  
✓ força  
✓ fraqueza  
✓ manipulação  
✓ liquidez  
✓ urgência  
✓ comportamento

Microestrutura é o idioma do mercado real.

### CAPÍTULO 5 • ORDERFLOW INSTITUCIONAL

Orderflow é o estudo e interpretação do fluxo real de ordens que atravessa o livro.  
Em vez de olhar para o preço final (a vela), o trader observa **como** esse preço é criado, **quem** o moveu e **com que intensidade**.

Enquanto a análise técnica tenta prever o que poderá acontecer, o orderflow mostra **o que está a acontecer neste exato momento**.

#### 1. ORDERFLOW É A LEITURA DAS CAUSAS DO MERCADO

A vela é o resumo.  
O footprint é o conteúdo.  
O DOM é a dinâmica interna.  
O Times & Sales é o pulso.  
O delta é a força.  
A microestrutura é o contexto.

Quando combinados, estes elementos criam uma leitura completa:

• quem está no controlo  
• quando o controlo muda  
• quando há manipulação  
• quando há defesa  
• quando há absorção  
• quando há aceleração  
• quando há exaustão  
• quando há oportunidade real

Orderflow remove a incerteza visual das velas e substitui-a por análise comportamental.

#### 2. ELEMENTOS FUNDAMENTAIS DO ORDERFLOW

Existem cinco pilares que compõem o núcleo do orderflow institucional:

• Delta (bid vs ask)  
• Times & Sales (agressão real)  
• DOM (livro de ordens)  
• Footprint (volume por nível)  
• CVD (cumulative volume delta)

Cada um deles oferece uma perspetiva específica.

A seguir estudamos cada elemento com profundidade.

#### 3. TIMES & SALES • O PULSO DO MERCADO

O Times & Sales (T&S) mostra:

• cada trade executado  
• o tamanho da ordem  
• o preço  
• o lado agressor (bid ou ask)  
• a velocidade dos prints

É o único sítio onde se vê **quem realmente tomou a iniciativa**.

Sinais observáveis no T&S:

**➤ Agressão ordenada**

Sequência rápida de trades todos no ask → compradores agressivos.

**➤ Agressão dispersa**

Trades pequenos sem direção → mercado indeciso.

**➤ Iceberg absorver**

Tamanho aparente pequeno mas muitos prints no mesmo preço.

**➤ Bursts**

Fluxos explosivos de prints, normalmente feitos por algoritmos institucionais.

O T&S indica **intenção e urgência**.

#### 4. DOM • A ESTRUTURA DA LIQUIDEZ

O Depth of Market mostra:

• liquidez resting  
• mudanças de posição  
• alterações rápidas de tamanho  
• remoção de ordens  
• adição de ordens  
• possíveis spoofings  
• concentração de interesse

A leitura do DOM revela:

**➤ Força defensiva**

Liquidez grande que permanece apesar da agressão.

**➤ Fraqueza defensiva**

Liquidez que desaparece ao primeiro sinal.

**➤ Interesse institucional**

Tamanhos anómalos em níveis estratégicos.

**➤ Manipulação**

Ordes grandes que aparecem e desaparecem repetidamente.

**➤ Intenção oculta**

Pequenas alterações no livro que revelam planos dos algos.

O DOM mostra a **arquitetura interna do mercado**.

#### 5. DELTA • O BALANÇO ENTRE COMPRADORES E VENDEDORES

O delta é calculado como:

volume\_agressor\_ask − volume\_agressor\_bid

Delta positivo → compradores agressivos no controlo  
Delta negativo → vendedores agressivos no controlo

A leitura avançada de delta inclui:

**➤ Delta divergente**

O preço sobe mas o delta cai → fraqueza na compra.

**➤ Delta exausto**

Agressão alta sem deslocamento → absorção.

**➤ Delta climático**

Explosão de agressão → final de movimento.

**➤ Delta sequencial**

Vários candles com delta forte → continuação provável.

#### 6. CVD • UMA LEITURA DE CONTINUIDADE

O CVD (Cumulative Volume Delta) acumula o delta ao longo do tempo.  
É uma leitura contínua da pressão agressora.

Cenários importantes:

**➤ CVD sobe enquanto o preço cai**

Compradores absorvidos → possível inversão.

**➤ CVD cai enquanto o preço sobe**

Vendedores absorvidos → manipulação antes do breakout.

**➤ CVD dispara**

Intervenção institucional clara.

**➤ CVD lateral**

Equilíbrio de forças.

O CVD oferece contexto estrutural ao delta.

#### 7. FOOTPRINT • A IMAGEM INTERNA DA VELA

O footprint divide cada candle em múltiplos níveis, mostrando:

• volume negociado no bid  
• volume negociado no ask  
• desequilíbrios por nível  
• absorções internas  
• níveis de agressão máxima

O footprint explica **porque a vela ficou daquela forma**.

Elementos críticos:

**➤ Imbalances**

Níveis onde o volume agressor é 2, 3 ou 4 vezes maior que o oposto.

**➤ Absorção interna**

Muito volume mas pouco deslocamento.

**➤ Ponto de controlo**

Preço com maior concentração de volume.

**➤ Zona de decisão**

Blocos de volume alinhados verticalmente.

#### 8. ABSORÇÃO • A DEFESA INSTITUCIONAL

No orderflow a absorção é lida como:

• agressão contínua contra um nível  
• preço que não progride  
• prints repetidos  
• liquidez passiva que se regenera  
• grandes players escondidos no livro

Absorção é um dos sinais mais importantes:

✓ precede reversões  
✓ precede quebras falsas  
✓ confirma zonas defendidas  
✓ representa controlo institucional

Será implementado no motor de orderflow na parte prática.

#### 9. SPOOFING • O ENGANO PROFISSIONAL

O spoofing, explicado em microestrutura, ganha profundidade no orderflow:

• aparece liquidez enorme repentinamente  
• tenta influenciar perceção  
• reage rapidamente à agressão  
• desaparece antes da execução

Sinais de spoofing:

• liquidez aparece longe do preço para “empurrar”  
• liquidez aparece perto do preço para simular defesa  
• desaparecimentos sincronizados com prints no T&S

Detetar spoofing é essencial para evitar armadilhas algorítmicas.

#### 10. BURSTS • ATIVIDADE ALGORÍTMICA

Bursts são sequências rápidas de prints:

• todos agressivos  
• todos no mesmo lado  
• com velocidade superior ao normal

Indicam:

• algos institucionais a atuarem  
• execução acelerada  
• momentum verdadeiro  
• zonas de rutura

O motor de ordemflow irá identificar bursts automaticamente.

#### 11. COMO OS PROFISSIONAIS LÊEM ORDERFLOW

Traders profissionais observam:

✓ se há interesse real  
✓ se há ou não absorção  
✓ se a agressão está a falhar  
✓ se a liquidez está a desaparecer  
✓ se existe intenção oculta  
✓ se a velocidade apoia o movimento  
✓ se os desequilíbrios são estruturais  
✓ se há participantes grandes escondidos

É uma leitura comportamental, não visual.  
É a leitura que usaremos para construir sinais internos, como:

• absorção confirmada  
• desequilíbrio dominante  
• inversão por exaustão  
• continuação por delta alinhado  
• manipulação por spoofing

#### 12. A LIGAÇÃO ENTRE ORDERFLOW E A CONSTRUÇÃO DA APP

O entendimento deste capítulo permite ao leitor perceber:

• porque vamos criar buffers de ticks  
• porque precisamos de snapshots do DOM  
• porque vamos tratar bid e ask separadamente  
• porque precisamos de um event bus  
• porque vamos criar um motor separado para delta  
• porque o footprint é calculado no engine e não na UI  
• porque certos sinais exigem análise sequencial

A aplicação terá literalmente todos os componentes explicados aqui.

## 📕 PARTE III • PREPARAÇÃO TÉCNICA DO PROJETO

### CAPÍTULO 6 • SETUP INSTITUCIONAL DE DESENVOLVIMENTO (PYTHON + VS CODE + ARQUITETURA BASE)

Antes de escrever a primeira linha do conector IBKR, o leitor precisa de montar um ecossistema de desenvolvimento que imite as boas práticas do mundo institucional.

O objetivo é transformar o computador doméstico numa **mini-mesa de trading institucional**, com:

• ambiente isolado  
• dependências controladas  
• arquitetura modular  
• logging profissional  
• testes estruturados  
• execução repetível

Este capítulo explica passo a passo como preparar tudo.

#### 1. PORQUE O SETUP IMPORTA NO TRADING INSTITUCIONAL

Ambientes profissionais não toleram:

✘ scripts desorganizados  
✘ ficheiros soltos  
✘ código misturado  
✘ dependências conflitantes  
✘ ausência de logging  
✘ ambientes instáveis  
✘ erros silenciosos

Em instituições financeiras:

✓ tudo é modular  
✓ tudo é testado  
✓ tudo é versionado  
✓ tudo é isolado  
✓ tudo é auditável  
✓ tudo é replicável

Vamos replicar esse padrão aqui.

#### 2. INSTALAR PYTHON DA FORMA CORRETA

A instalação tem de cumprir três requisitos:

✓ deve permitir múltiplos ambientes  
✓ deve ser compatível com bibliotecas de UI (Qt/PySide6)  
✓ deve ser compatível com bibliotecas de networking e threading

Requisitos mínimos:

• Python 3.10, 3.11 ou 3.12  
• pip e venv disponíveis  
• adicionar Python ao PATH

Recomendação:

• instalar via Python.org  
• evitar instalar pelo Windows Store  
• nunca desenvolver sem ambiente virtual

#### 3. INSTALAR VS CODE OU PYCHARM

Ambos funcionam perfeitamente.  
A escolha depende do estilo do programador.

**VS Code**

✓ leve  
✓ rápido  
✓ extensões úteis  
✓ ideal para modularidade  
✓ ótimo para Python + UI + IBKR

**PyCharm**

✓ ambiente de IDE mais pesado  
✓ ferramentas integradas  
✓ bom para debugging profundo  
✓ ótimas ferramentas de refatoração

Conclusão:  
**VS Code é preferido** nesta edição porque permite visualizar múltiplos módulos e ficheiros simultaneamente, algo essencial para arquiteturas profissionais.

#### 4. CRIAR A ESTRUTURA DO PROJETO

Criamos uma pasta chamada:

**tape\_reading\_app**

E dentro dela construímos a estrutura institucional:

tape\_reading\_app/

config/

core/

engine/

orderflow/

dom/

footprint/

delta/

utils/

ibkr/

ui/

data/

logs/

tests/

main.py

Explicação das pastas:

**config/**

Ficheiros de configuração, tokens, settings, parâmetros de ligações.

**core/**

O núcleo do sistema, com:

• motor de eventos  
• buffers  
• cálculo de delta  
• cálculo de footprint  
• lógica interna de sinais  
• ligação com DOM e T&S

**ibkr/**

Tudo relacionado ao conector:

• EClient  
• EWrapper  
• callbacks  
• reconexão  
• normalização de dados  
• distribuição de eventos para o engine

**ui/**

Componentes visuais PySide6:

• DOM visual  
• footprint  
• delta chart  
• T&S  
• heatmap  
• navegação e sincronização

**data/**

Datasets gravados:

• DOM  
• delta  
• trades  
• replay

**logs/**

Logs estruturados:

• eventos  
• execuções  
• erros  
• telemetria

**tests/**

Testes unitários e de integração:

• teste de conector  
• teste de buffers  
• teste de cálculo  
• teste de UI

**main.py**

Ponto de entrada da aplicação.

#### 5. CRIAR O AMBIENTE VIRTUAL

No terminal:

python -m venv venv

Ativar:

• Windows

venv\Scripts\activate

• Linux/Mac

source venv/bin/activate

Dentro do ambiente virtual vamos instalar bibliotecas:

pip install ibapi

pip install PySide6

pip install numpy

pip install pandas

pip install pydantic

pip install loguru

pip install matplotlib

pip install pyqtgraph

pip install pytest

pip install websockets

pip install python-dotenv

Cada biblioteca tem função específica que usaremos em capítulos posteriores.

#### 6. ESCOLHA DO MODELO DE ARQUITETURA

Existem três modelos possíveis:

**Modelo 1: Procedural (péssimo para este projeto)**

Código solto, funções globais, difícil de escalar.

**Modelo 2: OOP Tradicional (aceitável)**

Classes organizadas, modularidade razoável.

**Modelo 3: Arquitetura Event-Driven (ideal)**

O mercado funciona por eventos.  
IBKR funciona por callbacks.  
Logo, a tua aplicação também deve funcionar assim.

A arquitetura event-driven permite:

✓ processamento em tempo real  
✓ baixo acoplamento  
✓ alta performance  
✓ separação clara de responsabilidades  
✓ escalabilidade  
✓ integração com UI sem bloquear o loop

É o padrão das mesas institucionais.

#### 7. O EVENT BUS • A ESTRUTURA QUE LIGA TUDO

Vamos criar um sistema simples:

✓ o conector IBKR publica eventos (ticks, DOM, trades)  
✓ o motor de orderflow subscreve esses eventos  
✓ a UI também subscreve esses eventos  
✓ todos os módulos recebem os dados em tempo real

A arquitetura ficará assim:

IBKR → EVENT BUS → ORDERFLOW ENGINE → UI  
IBKR → EVENT BUS → EXECUTION ENGINE  
IBKR → EVENT BUS → LOGGING

Toda a app gira em torno deste sistema.

#### 8. LOGGING INSTITUCIONAL

Em bancos, tudo é logado.  
Não existe:

✘ print()  
✘ exceções silenciosas  
✘ perdas de informação

A biblioteca recomendada é:

loguru

Permitindo:

• logging de microestruturas  
• logging de erros  
• logging de execução  
• logs rotativos  
• timestamps precisos  
• ficheiros segmentados

Exemplo simples:

**from** loguru **import** logger

logger.**add**("logs/app.log", rotation="10 MB")

logger.info("Aplicação iniciada")

Mais tarde usaremos logs estruturados (JSON).

#### 9. PREPARAR O IB GATEWAY

É essencial preparar o ambiente institucional de dados:

1. instalar IB Gateway
2. configurar login automático
3. ativar market data permissions
4. ativar depth of market
5. ativar tick-by-tick
6. confirmar permissão para futuros CME

A ligação final será:

Python → IBAPI → IB Gateway → IBKR → CME

Este canal é extremamente estável e amplamente usado profissionalmente.

#### 10. PRIMEIRAS REGRAS TÉCNICAS DO PROJETO

Estes princípios serão obrigatórios:

✓ nunca bloquear o event loop  
✓ nunca recalcular tudo; apenas incremental  
✓ nunca processar dados na thread da UI  
✓ nunca pedir dados em polling; apenas callbacks  
✓ nunca misturar lógica de UI com lógica de cálculo  
✓ nunca gravar ficheiros na thread principal

O objetivo é construir software robusto e escalável.

### CAPÍTULO 7 • ARQUITETURA PROFISSIONAL DE SOFTWARE PARA TRADING

Uma aplicação institucional de trading não é um conjunto de scripts colados ao acaso.  
É um sistema vivo, modular, previsível, testável e resiliente, capaz de lidar com falhas de ligação, picos de dados e decisões críticas sem se desintegrar.

Este capítulo define a arquitetura profissional que iremos usar ao longo do livro.  
Tudo o que vais programar depois encaixa neste modelo.

#### 1. OBJETIVO DA ARQUITETURA

A arquitetura tem de garantir cinco coisas essenciais:

1. Separação de responsabilidades
2. Baixo acoplamento entre componentes
3. Alta coesão dentro de cada módulo
4. Escalabilidade e capacidade de evolução
5. Resiliência a falhas e erros

Se qualquer uma destas dimensões falhar, a app torna-se:

✘ frágil  
✘ difícil de manter  
✘ impossível de testar  
✘ perigosa para operar com dinheiro real

A arquitetura que vais usar é inspirada em sistemas que bancos e prop firms utilizam para construir ferramentas internas.

#### 2. VISÃO GLOBAL DA ARQUITETURA

A aplicação será dividida em seis blocos principais:

1. Conector de dados
2. Event Bus
3. Orderflow Engine
4. Execution Engine (opcional)
5. UI institucional
6. Logging, armazenamento e telemetria

Fluxo lógico de dados:

IBKR (dados de mercado)  
→ Conector IBKR  
→ Event Bus  
→ Orderflow Engine  
→ UI

Quando existir execução:

UI ou lógica algorítmica  
→ Execution Engine  
→ Conector IBKR  
→ IBKR (ordens)

Tudo isto monitorizado por:

→ módulo de logging  
→ módulo de armazenamento de dados

#### 3. PRINCÍPIOS DE ENGENHARIA UTILIZADOS

Vamos usar princípios de engenharia de software típicos do ambiente institucional.

##### 3.1 Separação de responsabilidades

Cada módulo faz apenas uma coisa:

• o conector só fala com o IBKR  
• o event bus só distribui eventos  
• o engine só processa dados  
• a UI só mostra informação  
• o execution só cria e gere ordens  
• o logger só regista eventos

Nada de misturar UI com lógica de negócio ou conector com cálculo de delta.

##### 3.2 Baixo acoplamento

Os módulos comunicam entre si através de eventos e interfaces bem definidas, não por acesso direto a variáveis internas de outros módulos.

Quanto menos módulos souberem detalhes internos uns dos outros, mais fácil é:

• alterar lógica  
• trocar componentes  
• testar isoladamente  
• evitar bugs em cascata

##### 3.3 Alta coesão

Cada módulo trata de um conjunto muito bem definido de responsabilidades.  
Se um ficheiro ou classe faz coisas a mais, deve ser dividido.

Alta coesão significa que:

• tudo o que está num módulo pertence mesmo àquele domínio  
• não há funções aleatórias misturadas  
• não há classes “faz tudo”

##### 3.4 Event driven

Os mercados são governados por eventos.  
IBKR envia callbacks.  
Logo, a tua app também deve ser orientada a eventos.

A arquitetura event driven:

• responde a novos ticks  
• reage a atualizações de DOM  
• reage a novas trades no T and S  
• reage a comandos da UI  
• reage a sinais do engine  
• responde a erros de rede

Em vez de o teu código “perguntar” constantemente o que está a acontecer, ele “ouve” o que está a acontecer.

#### 4. BLOCOS PRINCIPAIS DA ARQUITETURA

##### 4.1 Conector IBKR

Responsabilidade:

• ligar ao IB Gateway  
• subscrever instrumentos  
• receber dados de mercado  
• normalizar eventos recebidos  
• reenviar os dados para o event bus

Nunca deve:

✘ calcular delta  
✘ mexer em footprint  
✘ desenhar gráficos  
✘ decidir sinais de trading

Ele é apenas a porta de entrada.

Características desejáveis:

• reconexão automática  
• timeouts controlados  
• gestão de IDs de pedidos  
• tratamento de erros e códigos IBKR  
• thread própria ou mecanismo de loop próprio

##### 4.2 Event Bus

Responsabilidade:

• distribuir eventos para quem os quer receber  
• permitir que múltiplos subscritores recebam o mesmo evento  
• manter o sistema desacoplado

Exemplo de eventos:

• MarketTickEvent  
• DomUpdateEvent  
• TradeEvent (T and S)  
• DeltaUpdateEvent  
• FootprintUpdateEvent  
• SignalEvent  
• OrderEvent  
• ErrorEvent

O event bus é a “rede interna” do sistema.

Características:

• leve  
• simples  
• thread safe  
• extensível

##### 4.3 Orderflow Engine

Responsabilidade:

• receber eventos brutos do mercado  
• transformar dados em informação estruturada  
• calcular delta e CVD  
• montar footprint  
• analisar DOM  
• detetar absorção, spoofing, imbalances, bursts  
• emitir sinais internos para a UI e para execução

É o cérebro analítico da aplicação.

Submódulos típicos:

• delta\_engine  
• footprint\_engine  
• dom\_engine  
• signal\_engine

Cada um destes submódulos trata de um aspecto específico do fluxo.

##### 4.4 Execution Engine (opcional)

Responsabilidade:

• transformar sinais em ordens  
• aplicar regras de risco  
• definir lógica de entrada e saída  
• criar ordens market e limit  
• enviar ordens para o conector IBKR  
• receber confirmações de execução  
• gerir ordens ativas e posições

O execution engine é uma camada extra.  
A app pode funcionar como ferramenta de leitura sem ele.

Regras importantes:

• nunca enviar ordens sem passar pelo módulo de risco  
• nunca bloquear a UI  
• nunca assumir que a ordem foi executada até receber confirmação  
• registar todas as ações em log

##### 4.5 Interface de Utilizador (UI)

Responsabilidade:

• mostrar DOM  
• mostrar footprint  
• mostrar delta  
• mostrar T and S  
• mostrar sinais e estados  
• receber comandos do utilizador

A UI é cliente do event bus:

• subscreve eventos  
• atualiza componentes visuais  
• envia eventos de input (por exemplo, clique num botão de buy ou sell)

A UI não calcula microestrutura.  
Apenas apresenta resultados.

##### 4.6 Logging, Armazenamento e Telemetria

Responsabilidade:

• registar o que aconteceu  
• fornecer histórico para análise  
• permitir debugging  
• gerar datasets para replay

Dados gravados:

• logs de sistema  
• ticks  
• DOM snapshots  
• trades  
• delta  
• sinais  
• execuções

Mais tarde será possível:

• voltar atrás no tempo  
• reproduzir sessões  
• treinar modelos  
• auditar decisões

#### 5. CICLO COMPLETO DE UM EVENTO DE MERCADO

Um evento típico flui assim:

1. A bolsa executa uma trade
2. IBKR envia callback com essa trade
3. O conector IBKR converte para um TradeEvent
4. O TradeEvent é publicado no event bus
5. O Orderflow Engine recebe o TradeEvent
6. O engine atualiza delta, CVD e footprint
7. O engine gera um DeltaUpdateEvent e um FootprintUpdateEvent
8. A UI recebe esses eventos e atualiza gráficos
9. Opcionalmente, o engine também gera um SignalEvent
10. O Execution Engine recebe o SignalEvent e pode decidir enviar uma ordem

Cada passo está bem definido.  
Cada módulo sabe exatamente o seu papel.

#### 6. CICLO COMPLETO DE UMA ORDEM

Quando a app também gere execução:

1. O utilizador clica num botão na UI ou a lógica algorítmica gera um sinal
2. A UI ou o módulo de lógica cria um OrderRequestEvent
3. O OrderRequestEvent é enviado para o Execution Engine
4. O Execution Engine valida risco e condições
5. Se tudo estiver em conformidade, envia a ordem para o conector IBKR
6. O conector envia a ordem pela IBAPI
7. IBKR devolve um estado de ordem (pendente, parcialmente preenchida, completa, cancelada)
8. O conector cria um OrderStatusEvent
9. O Execution Engine e a UI recebem o OrderStatusEvent
10. A UI mostra o estado e o logger regista

Mais uma vez, tudo é baseado em eventos.

#### 7. THREADS, CONCORRÊNCIA E PERFORMANCE

Uma aplicação destas precisa de lidar com:

• dados em tempo real  
• interface responsiva  
• cálculo contínuo  
• disco e logs

Se tudo correr numa única thread, a interface fica lenta e pode congelar.

Padrão recomendado:

• uma thread para IBKR e callbacks  
• uma thread principal para UI  
• event bus thread safe  
• cálculos pesados delegados a worker threads ou executors  
• gravação de dados em thread separada

Objetivo:

✓ UI fluida  
✓ sem perda de dados  
✓ sem bloqueios  
✓ sem interdependências perigosas

#### 8. ERROS, FALHAS E RESILIÊNCIA

Qualquer sistema de trading deve assumir que:

• a ligação à corretora pode falhar  
• a internet pode oscilar  
• a API pode devolver erros  
• a máquina pode aquecer ou abrandar  
• o utilizador pode cometer erros

A arquitetura deve prever:

• reconexão automática ao IBKR  
• retentativas controladas  
• timeouts  
• deteção de perda de dados  
• mecanismos de safe shutdown  
• logs detalhados para diagnóstico

O software nunca deve:

✘ crashar silenciosamente  
✘ ignorar exceções  
✘ deixar ordens abertas sem registo  
✘ bloquear sem feedback

Robustez é uma característica obrigatória num sistema institucional.

#### 9. TESTES E QUALIDADE

Uma arquitetura profissional permite testar:

• o conector isolado  
• o event bus isolado  
• o engine com dados gravados  
• a UI com dados simulados  
• o execution com ordens simuladas

Podem ser criados:

• testes unitários para funções críticas  
• testes de integração para fluxos de eventos  
• testes de carga para ver performance sob stress

Sempre que algo falhar, deve ficar claro:

• qual módulo falhou  
• em que condições  
• qual evento originou o problema

Sem arquitetura clara, isto torna-se quase impossível.

#### 10. RESUMO DA ARQUITETURA ALVO

No final deste livro, a aplicação terá:

• um conector IBKR robusto  
• um event bus interno simples e eficaz  
• um orderflow engine modular e poderoso  
• uma UI institucional completa  
• um optional execution engine  
• logging e datasets de nível profissional

Tudo isto baseado numa arquitetura event driven, modular e resiliente.

Esta é a base que diferencia um “projeto de hobby” de uma ferramenta institucional.

## 📕 PARTE IV • IBKR: A PORTA PARA O MUNDO INSTITUCIONAL

### CAPÍTULO 8 • ENTENDER O ECOSSISTEMA IBKR E PREPARAR O ACESSO INSTITUCIONAL

A Interactive Brokers é uma das poucas corretoras no mundo que disponibiliza:

• dados verdadeiros de bolsa (CME, NASDAQ, NYSE, EUREX)  
• acesso à API oficial  
• feed tick-by-tick real  
• DOM profundo e atualizável  
• infraestrutura institucional  
• execução robusta  
• disponibilidade global

O ecossistema da IBKR foi criado para profissionais e fundos, não para traders de retalho.  
É isso que a torna a melhor porta de entrada para a **Layer 2.5**.

Este capítulo explica:

• como funciona a IBAPI  
• porque precisas do IB Gateway (e não TWS)  
• que permissões são necessárias  
• como a IB processa dados  
• como a API entrega eventos  
• como configurar tudo na prática

#### 1. OS TRÊS COMPONENTES ESSENCIAIS DA IBKR

Para interagir com IBKR a partir de Python precisas de três partes independentes:

**1) IB Gateway (ou TWS)**

É o cliente oficial que mantém a ligação ao servidor da IBKR.

**2) A API oficial (IBAPI)**

Conjunto de bibliotecas usadas para comunicação programática.

**3) O teu programa Python**

A aplicação que vais construir, que se liga ao Gateway e recebe dados.

Fluxo:

Python app → IBAPI → IB Gateway → Servidores IBKR → Bolsas  
Bolsas → IBKR → Gateway → IBAPI → Python app

Este fluxo garante:

• segurança  
• autenticação  
• controlo  
• sincronização

#### 2. DIFERENÇA ENTRE TWS E IB GATEWAY

Existem dois clientes oficiais da IBKR:

**➤ TWS (Trader Workstation)**

• interface gráfica completa  
• visual, pesado, estável mas lento  
• consome mais RAM  
• pensado para utilizadores humanos

**➤ IB Gateway**

• interface mínima  
• sem gráficos  
• sem janelas  
• baixa necessidade de RAM  
• ideal para bots  
• está sempre ligado  
• reinicia automaticamente

**Conclusão**

Para uma aplicação institucional, **IB Gateway é a escolha correta**.

Todas as empresas, fundos e bots profissionais usam o Gateway, não o TWS.

#### 3. COMO A IBKR ENTREGA OS DADOS DE MERCADO

Para compreender como construir o nosso engine, é essencial perceber como a IB envia dados.

**A IBKR envia dados de forma assíncrona**

Não existe “pedido repetido” ou polling.  
A IB envia event-driven:

• ticks  
• trades  
• variações no DOM  
• snapshots de preço  
• erros  
• confirmações de ordem

Tudo chega via callbacks.

**Tipos de dados:**

**Tick-by-Tick**

Dados de alta frequência:

• último preço  
• tamanho  
• timestamp  
• lado agressor

**Market Depth (DOM)**

Atualizações por nível:

• preço  
• tamanho  
• lado (bid/ask)  
• posição no livro

**Market Data Streams**

Incluem:

• bid  
• ask  
• mid  
• volume  
• último preço  
• alterações contínuas

Estes dados irão alimentar diretamente o event bus.

#### 4. LIMITAÇÕES IMPORTANTES DA IB API (E COMO CONTORNÁ-LAS)

A API não é perfeita e tem limitações institucionais importantes.

**Limitação 1 → Quantidade de pedidos simultâneos**

IB impõe limites para evitar abuso.  
Mas isto não é problema porque vamos focar-nos em:

✔ 1 instrumento (GC/ES/NQ)  
✔ 1 DOM  
✔ 1 stream tick-by-tick

**Limitação 2 → Update rate do DOM**

DOM é atualizado:

• entre ~5 ms e ~50 ms dependendo do ativo

Para análise institucional manual e algorítmica de Layer 2.5, isto é suficiente.

**Limitação 3 → Tick-by-tick limitado**

A API envia tick-by-tick real, mas pode agrupar eventos no caso de volume excessivo.  
Para o GC/ES/NQ esta frequência é mais do que suficiente.

**Limitação 4 → Ordem de chegada**

O IBKR não garante sempre a ordem absoluta dos eventos se houver congestionamento de rede local. Por isso vamos implementar:

✔ buffers  
✔ timestamps locais  
✔ normalização de eventos no engine

#### 5. PERMISSÕES NECESSÁRIAS PARA ACEDER AO ORDERFLOW REAL

Para trabalhar com dados de bolsa, tens de ativar permissões específicas:

**Permissões obrigatórias:**

✓ US Futures (CME Bundle)  
✓ Depth of Market (DOM)  
✓ Real-Time Market Data  
✓ Tick-By-Tick data

**Instrumentos recomendados:**

• GC (Gold Futures)  
• ES (S&P 500 e-mini)  
• NQ (Nasdaq e-mini)  
• 6E (Euro FX Futures)

Sem estas permissões **não existe orderflow verdadeiro**.

#### 6. CONCEITOS ESSENCIAIS DO IBKR PARA DESENVOLVEDORES

Estes conceitos são absolutamente obrigatórios antes de programar.

**Client ID**

Um número que identifica a tua aplicação.  
Cada aplicação precisa do seu próprio Client ID.

**Order ID**

Cada ordem enviada deve ter um ID único.  
O conector será responsável por geri-los.

**Request ID**

Cada pedido de dados (market data, DOM, históricos) requer um ID.

**Callback**

Método associado a um evento.  
Exemplo: quando o IBKR envia uma atualização do DOM, chama o método updateMktDepth.

**EClient**

A classe usada para enviar pedidos e ordens.

**EWrapper**

A classe usada para receber eventos.

A tua app será uma classe que herda **EClient + EWrapper** ao mesmo tempo.

#### 7. O CICLO DE VIDA DE UMA LIGAÇÃO IBKR

O ciclo normal é:

1. Iniciar IB Gateway
2. Autenticar
3. Abrir o teu programa Python
4. Ligar ao Gateway via socket
5. Enviar pedidos de market data
6. Receber callbacks contínuos
7. Iniciar o motor de orderflow
8. Ligar a UI a esses eventos

E, se necessário:

1. Enviar ordens
2. Receber confirmações
3. Gerir posições
4. Fazer shutdown limpo

#### 8. CONFIGURAÇÃO INICIAL DO IB GATEWAY PARA ESTE PROJETO

É recomendável criar um ambiente dedicado apenas ao bot.

**Configurações recomendadas:**

✓ Login automático  
✓ Definir auto-restart  
✓ Ativar permissão de API  
✓ Ativar market data permissions  
✓ Ativar Market Depth (DOM)  
✓ Desativar janelas não necessárias  
✓ Definir porta fixa (4001 ou 7497)

**Ambiente sugerido:**

• mini-PC dedicado  
• Windows ou Linux  
• IB Gateway a correr 24/7  
• o bot liga-se automaticamente

#### 9. TESTE RÁPIDO DE LIGAÇÃO (CONCEITO)

Para validar que tudo funciona, um programa mínimo:

**from** ibapi.client **import** EClient

**from** ibapi.wrapper **import** EWrapper

**class** IBKRApp(EWrapper, EClient):

**def** \_\_init\_\_(**self**):

EClient.\_\_init\_\_(**self**, **self**)

app = IBKRApp()

app.connect("127.0.0.1", 4001, clientId=1)

app.run()

Quando isto funciona:

→ tens ligação  
→ a API está ativa  
→ podes continuar para o próximo capítulo

A implementação final será muito mais robusta, mas este teste confirma o ambiente.

##### 10. O QUE VEM A SEGUIR

Agora que sabes:

• como funciona o ecossistema  
• como a API entrega dados  
• como configurar IB Gateway  
• que permissões são necessárias  
• o papel dos callbacks  
• a diferença entre TWS e Gateway  
• como será a comunicação com a app

### CAPÍTULO 9 • CONSTRUIR O CONECTOR IBKR PROFISSIONAL (EClient + EWrapper)

O conector IBKR é o **coração da tua aplicação**.

Sem ele:

• não tens ticks  
• não tens DOM  
• não tens Times & Sales  
• não tens ligação ao mercado  
• não tens execução  
• não tens delta, footprint, CVD ou sinais

Nada funciona.

Este capítulo ensina, passo a passo, como criar um conector institucional robusto.

#### 1. CONCEITO FUNDAMENTAL: EClient + EWrapper

A API da IBKR funciona com duas classes:

**➤ EClient**

Envia pedidos:  
• market data  
• DOM  
• ordens  
• cancelamentos

**➤ EWrapper**

Recebe eventos:  
• ticks  
• trades  
• atualizações de DOM  
• estado das ordens  
• erros  
• notificações

Para criar um conector profissional, precisamos de:

**✔ uma classe Python que herda ambas**

**✔ reconexão automática**

**✔ buffers internos**

**✔ normalização de eventos**

**✔ proteção contra falhas**

**✔ integração com o event bus**

Vamos chamar esta classe de:

**IBKRClient**

#### 2. COMO FUNCIONA A ESTRUTURA DO CONECTOR

A estrutura profissional será composta por:

ibkr/

ibkr\_client.py

ibkr\_connection.py

types/

tick.py

trade.py

dom.py

order.py

Separação clara:

• ibkr\_client → lida com IBAPI diretamente  
• ibkr\_connection → validação, reconexão, segurança  
• types → modelos de dados internos para normalizar tudo

#### 3. A CLASSE PRINCIPAL: IBKRClient

O objetivo é:

• iniciar ligação  
• enviar pedidos  
• receber callbacks  
• transformar tudo em eventos internos  
• enviar eventos ao event bus

Elementos que esta classe terá:

**✓ construtor robusto**

**✓ gestão de IDs automáticos**

**✓ métodos de pedido de market data**

**✓ métodos de pedido de DOM**

**✓ métodos de envio de ordens**

**✓ callbacks personalizados**

**✓ logging detalhado**

**✓ medição de latência**

**✓ normalização de dados**

#### 4. O CICLO TÉCNICO DE UM CALLBACK

Sempre que a IBKR envia um evento:

1. a IBAPI chama um método callback (ex: tickPrice)
2. o callback transforma esse evento num objeto interno
3. o objeto é enviado ao event bus
4. o motor de orderflow processa
5. a UI atualiza
6. o logger regista

Nada é processado diretamente no conector.  
O conector é **apenas a porta de entrada**.

#### 5. CRIAÇÃO DO CONECTOR – FUNDAMENTOS EM CÓDIGO

A estrutura inicial do cliente é assim:

**from** ibapi.client **import** EClient

**from** ibapi.wrapper **import** EWrapper

**from** ibapi.contract **import** Contract

**from** loguru **import** logger

**class** IBKRClient(EWrapper, EClient):

**def** \_\_init\_\_(**self**, event\_bus):

EClient.\_\_init\_\_(**self**, **self**)

**self**.event\_bus = event\_bus

**self**.next\_order\_id = **None**

**self**.connected = **False**

# Callback de ligação

**def** nextValidId(**self**, orderId: **int**):

logger.info(f"Ligação validada. Próximo Order ID = {orderId}")

**self**.next\_order\_id = orderId

**self**.connected = **True**

Explicação:

• recebe o event bus  
• cria estado interno  
• obtém o primeiro orderId

#### 6. SUBSCRIÇÃO DE MARKET DATA (TICK-BY-TICK)

Para receber ticks reais da CME:

**def** subscribe\_ticks(**self**, req\_id, contract):

logger.info(f"Subscrição de tick-by-tick para {contract.symbol}")

**self**.reqTickByTickData(req\_id, contract, "Last", 0, **False**)

Tipos de tick-by-tick possíveis:

• Last (trades)  
• BidAsk  
• AllLast

Tick-by-tick é essencial para delta e T&S.

#### 7. SUBSCRIÇÃO DO DOM (MARKET DEPTH)

Esta é a parte mais importante do conector.

**def** subscribe\_dom(**self**, req\_id, contract, depth=10):

logger.info(f"Subscrição de DOM para {contract.symbol}, profundidade {depth}")

**self**.reqMktDepth(req\_id, contract, depth, **False**, [])

O DOM fornece:

• níveis de preço  
• volumes resting  
• remoção e adição de liquidez  
• movimentos internos do livro

Essencial para:

✓ absorção  
✓ spoofing  
✓ pulling e stacking  
✓ desequilíbrios

#### 8. CALLBACKS PRINCIPAIS DO DOM

**def** updateMktDepth(**self**, reqId, position, operation, side, price, size):

**event** = {

"type": "dom\_update",

"position": position,

"operation": operation,

"side": side,

"price": price,

"size": size,

}

**self**.event\_bus.publish("dom\_update", **event**)

Explicação dos parâmetros:

• position → nível no DOM  
• operation → 0=insert, 1=update, 2=delete  
• side → 0=bid, 1=ask  
• price → preço  
• size → quantidade

Cada atualização será tratada pelo motor de DOM.

#### 9. CALLBACKS PRINCIPAIS DO TICK-BY-TICK

**def** tickByTickAllLast(**self**, reqId, tickType, time, price, size, tickAttribLast, exchange, specialConditions):

**event** = {

"type": "trade",

"price": price,

"size": size,

"timestamp": time,

"exchange": exchange,

}

**self**.event\_bus.publish("trade", **event**)

Será usado no motor de delta e no T&S.

#### 10. CALLBACK DE PREÇO ESPECÍFICO

**def** tickPrice(**self**, reqId, tickType, price, attrib):

**event** = {

"type": "tick\_price",

"price": price,

"tickType": tickType,

}

**self**.event\_bus.publish("tick\_price", **event**)

tickType identifica:

• BID  
• ASK  
• LAST  
• HIGH  
• LOW

#### 11. CALLBACK DE ERROS E RECONECÇÃO

Nenhum sistema institucional opera sem isto.

**def** error(**self**, reqId, errorCode, errorString):

logger.error(f"Erro IBKR: {errorCode} → {errorString}")

**if** errorCode **in** [1100, 1101, 1102]:

logger.warning("Problema de ligação IBKR. Tentando reconectar...")

Depois iremos implementar reconexão automática.

#### 12. ENVIO DE ORDENS (EXECUTION ENGINE)

Para enviar ordens, o cliente precisa de:

• next order ID  
• contract  
• order object

Exemplo:

**def** place\_order(**self**, contract, order):

order\_id = **self**.next\_order\_id

logger.info(f"Enviando ordem ID {order\_id}")

**self**.placeOrder(order\_id, contract, order)

**self**.next\_order\_id += 1

O execution engine vai chamar este método.

#### 13. LIGAÇÃO AO IB GATEWAY

A ligação final:

client = IBKRClient(event\_bus)

client.connect("127.0.0.1", 4001, clientId=1)

E iniciar o loop:

thread = Thread(target=client.run)

thread.start()

Nunca bloquear a thread principal.

#### 14. REQUISITOS DE ROBUSTEZ DO CONECTOR

O conector será considerado profissional quando possuir:

✓ reconexão automática  
✓ gestão de IDs  
✓ normalização de eventos  
✓ logging detalhado  
✓ medição de timestamps  
✓ state machine interna  
✓ tratamento de erros  
✓ subscrição persistente  
✓ inicialização limpa  
✓ shutdown limpo

Tudo isto será implementado nos próximos capítulos.

##### 15. O QUE VEM A SEGUIR

Agora que compreendemos:

• estrutura do conector  
• subscrições  
• callbacks  
• gestão de eventos  
• IDs  
• ordens  
• DOM  
• tick-by-tick  
• T&S

### CAPÍTULO 10 • MODELOS DE DADOS INSTITUCIONAIS (TRADEEVENT, DOMEVENT, TICKEVENT, DELTAEVENT…)

Uma app de trading institucional é, acima de tudo, um sistema de processamento de dados em tempo real.

Se esses dados forem mal representados, mal tipados ou inconsistentes, tudo o que vem por cima:

• orderflow  
• UI  
• sinais  
• execução

fica frágil, incoerente e difícil de manter.

Este capítulo define **os modelos de dados institucionais** que serão usados em toda a aplicação:

• TickEvent  
• TradeEvent  
• DomUpdateEvent  
• DomSnapshot  
• DeltaEvent  
• CvdEvent  
• FootprintBucket  
• FootprintCandleEvent

Vamos também definir princípios comuns:

• timestamps  
• símbolos  
• fontes  
• integridade  
• serialização

O objetivo é ter uma linguagem de dados clara, robusta e extensível.

#### 1. O PAPEL DOS MODELOS DE DADOS NO SISTEMA

Os modelos de dados são o contrato entre:

• o conector IBKR  
• o event bus  
• o orderflow engine  
• a UI  
• o execution engine  
• os módulos de logging e replay

Se cada parte do sistema inventar o seu próprio formato, acabas com:

✘ bugs difíceis de rastrear  
✘ duplicação de lógica  
✘ dados inconsistentes  
✘ impossibilidade de fazer replay  
✘ problemas de compatibilidade no futuro

Por isso, vamos:

✓ definir modelos claros  
✓ torná-los o mais imutáveis possível  
✓ garantir que cada evento é auto-suficiente  
✓ permitir gravação e replay sem perda de contexto

#### 2. TIPOS DE MODELOS: RAW, EVENTOS, ESTADO AGREGADO

Vamos trabalhar com três níveis de representação.

##### 2.1 Raw Message (camada IBKR)

São os objetos e callbacks que vêm diretamente da IBAPI, por exemplo:

• métodos como updateMktDepth  
• parâmetros soltos (reqId, price, size, side, etc.)  
• tipos de tick, códigos, enums internos da IB

Estes não serão usados fora do conector.

##### 2.2 Eventos Normalizados

São objetos claros, na nossa linguagem:

• TickEvent  
• TradeEvent  
• DomUpdateEvent  
• DeltaEvent  
• SignalEvent

Estes são publicados no event bus.

##### 2.3 Estado Agregado

São estruturas que mantêm uma visão persistente:

• DomSnapshot  
• FootprintCandle  
• CvdState  
• DeltaState

O orderflow engine usa eventos para atualizar estes estados.

#### 3. REQUISITOS GERAIS PARA TODOS OS EVENTOS

Cada evento deve conter, sempre que possível:

✓ timestamp  
✓ símbolo  
✓ origem (por exemplo, IBKR)  
✓ identificador de sessão  
✓ fields específicos  
✓ versão do esquema (para futuro)

Exemplo conceptual de campos comuns:

**from** pydantic **import** BaseModel

**from** datetime **import** datetime

**class** BaseEvent(BaseModel):

ts: datetime

symbol: str

source: str = "IBKR"

session\_id: str | **None** = **None**

schema\_version: **int** = 1

Todos os outros eventos vão herdar este modelo base.

#### 4. TICKEVENT • ATUALIZAÇÕES DE PREÇO

O TickEvent representa atualizações de preço simples, como:

• bid  
• ask  
• last  
• mid

Este evento é útil para:

• atualizar a UI  
• alimentar indicadores simples  
• medir latência  
• sincronizar gráficos com o fluxo interno

Estrutura recomendada:

**from** **enum** **import** Enum

**class** TickType(str, Enum):

BID = "bid"

ASK = "ask"

LAST = "last"

MID = "mid"

**class** TickEvent(BaseEvent):

tick\_type: TickType

price: **float**

size: **float** | **None** = **None**

Características:

• não substitui o TradeEvent  
• não substitui o DOM  
• é uma visão condensada, útil para gráficos e monitorização básica

#### 5. TRADEEVENT • TIMES & SALES

O TradeEvent é a representação mínima de uma trade executada:

• preço  
• tamanho  
• lado agressor (buy ou sell)  
• exchange  
• identificador de sequência

A IBAPI nem sempre indica diretamente o lado agressor, por isso pode ser inferido com base no tick-by-tick Ask/Bid, mas isso é função do engine.

Modelo sugerido:

**from** **enum** **import** Enum

**class** AggressorSide(str, Enum):

BUY = "buy"

SELL = "sell"

UNKNOWN = "unknown"

**class** TradeEvent(BaseEvent):

price: **float**

size: **float**

aggressor: AggressorSide

exchange: str | **None** = **None**

seq\_num: **int** | **None** = **None**

Este evento alimenta:

• cálculo de delta  
• T and S da UI  
• medição de speed of tape  
• deteção de bursts

#### 6. DOMUPDATEEVENT E DOMSNAPSHOT • O LIVRO DE ORDENS

O DOM é dinâmico.  
Vamos separar:

• atualizações incrementais  
• snapshots completos para estado interno

##### 6.1 Modelo DomLevel

Representa um nível individual do livro.

**from** pydantic **import** BaseModel

**class** DomSide(str, Enum):

BID = "bid"

ASK = "ask"

**class** DomLevel(BaseModel):

price: **float**

size: **float**

side: DomSide

position: **int**

##### 6.2 DomUpdateEvent

Representa uma alteração pontual num nível.

IBAPI fornece:

• position  
• operation (insert, update, delete)  
• side  
• price  
• size

Modelo:

**class** DomOperation(str, Enum):

INSERT = "insert"

UPDATE = "update"

DELETE = "delete"

**class** DomUpdateEvent(BaseEvent):

position: **int**

operation: DomOperation

side: DomSide

price: **float**

size: **float**

depth: **int** | **None** = **None**

Este evento é consumido pelo **dom\_engine**, que mantém o estado.

##### 6.3 DomSnapshot

Visão completa do livro num dado instante.

**class** DomSnapshot(BaseEvent):

bids: list[DomLevel]

asks: list[DomLevel]

depth: **int**

Utilizações:

• alimentar UI  
• gravar para replay  
• análises de liquidez  
• estudos de pulling e stacking

O dom\_engine poderá publicar snapshots periódicos (por exemplo, a cada 50 ms) ou quando existirem alterações significativas.

#### 7. DELTAEVENT E CVD • PRESSÃO AGRESSORA

A partir dos TradeEvents, o delta\_engine calcula o delta por unidade de tempo ou por candle.

##### 7.1 DeltaEvent

Representa a variação de delta num intervalo:

**class** DeltaEvent(BaseEvent):

interval\_start: datetime

interval\_end: datetime

delta: **float**

buy\_volume: **float**

sell\_volume: **float**

trades\_count: **int**

Pode ser emitido:

• ao fechar uma barra de tempo  
• ao atingir um certo número de trades  
• a pedido da UI

##### 7.2 CvdEvent

CVD é a acumulação do delta:

**class** CvdEvent(BaseEvent):

cvd\_value: **float**

last\_delta: **float**

direction: AggressorSide | **None** = **None**

A UI pode subscrever este evento para desenhar uma linha de CVD em tempo real.

#### 8. FOOTPRINTBUCKET E FOOTPRINTCANDLE • ESTRUTURA INTERNA DA VELA

O footprint é a representação mais rica da vela internamente.

Vamos quebrar em:

• buckets por nível de preço  
• candle completo organizado por buckets

##### 8.1 FootprintBucket

Representa um nível de preço dentro da vela:

**class** FootprintBucket(BaseModel):

price: **float**

bid\_volume: **float**

ask\_volume: **float**

delta: **float**

imbalance\_ratio: **float** | **None** = **None**

is\_imbalance: **bool** = **False**

is\_absorption: **bool** = **False**

##### 8.2 FootprintCandleEvent

Representa a vela inteira:

**class** FootprintCandleEvent(BaseEvent):

interval\_start: datetime

interval\_end: datetime

open\_price: **float**

high\_price: **float**

low\_price: **float**

close\_price: **float**

total\_volume: **float**

buckets: list[FootprintBucket]

point\_of\_control: **float** | **None** = **None**

max\_delta: **float** | **None** = **None**

min\_delta: **float** | **None** = **None**

O footprint\_engine é responsável por:

• agregar trades por nível  
• distinguir bid vs ask  
• calcular desequilíbrios  
• marcar absorções internas  
• identificar o ponto de controlo de volume

A UI pode subscrever este evento para desenhar uma vela footprint completa.

#### 9. SIGNALEVENT • SINAIS ESTRUTURADOS DE ORDERFLOW

Embora o foco desta edição seja **ferramenta institucional de leitura**, é útil formalizar sinais internos.

Exemplos:

• absorção num nível relevante  
• desequilíbrio extremo  
• burst agressivo  
• mudança súbita de direção do delta  
• spoofing detetado

Modelo de sinal:

**class** SignalType(str, Enum):

ABSORPTION = "absorption"

IMBALANCE = "imbalance"

BURST = "burst"

SPOOFING = "spoofing"

DELTA\_SHIFT = "delta\_shift"

**class** SignalEvent(BaseEvent):

signal\_type: SignalType

price: **float**

severity: **float**

details: dict

Este evento pode ser consumido pela UI e, opcionalmente, pelo Execution Engine.

#### 10. SERIALIZAÇÃO, LOGGING E REPLAY

Os modelos que definimos são:

• estruturados  
• tipados  
• auto-suficientes

Por isso, são perfeitos para:

✓ serializar em JSON  
✓ gravar em Parquet  
✓ enviar por WebSocket  
✓ reproduzir sessões para treino

Por exemplo:

**event**.json()

pode gerar a linha gravada em ficheiro.

Estratégia de logging:

• um ficheiro por tipo de evento  
• ou um ficheiro com o campo type indicando a categoria  
• timestamps sempre em UTC  
• schema\_version para permitir evolução futura

Isto permite mais tarde:

• carregar eventos de um dia  
• alimentar o orderflow engine em modo de replay  
• reanalisar toda a sessão como se fosse ao vivo

#### 11. LIGAÇÃO A EVENT BUS E À UI

Cada um destes modelos:

• será publicado no event bus com uma chave (topic) clara  
• será consumido por módulos específicos

Exemplos de tópicos:

• "tick" → TickEvent  
• "trade" → TradeEvent  
• "dom\_update" → DomUpdateEvent  
• "dom\_snapshot" → DomSnapshot  
• "delta" → DeltaEvent  
• "cvd" → CvdEvent  
• "footprint" → FootprintCandleEvent  
• "signal" → SignalEvent

A UI não precisa de saber como o dado é calculado.  
Só precisa de saber:

• qual tópico subscrever  
• que modelo receber  
• como representar visualmente

#### 12. VANTAGENS DESTA ABORDAGEM

Ao terminarmos este capítulo, conquistámos:

✓ linguagem comum para toda a app  
✓ independência entre módulos  
✓ capacidade de gravação e replay  
✓ facilidade de testar motores isolados  
✓ clareza arquitetural  
✓ extensibilidade futura

##### Tudo o que vem agora:

• motores de delta  
• footprint  
• DOM  
• UI  
• sinais

vai encaixar sobre estes modelos.

### CAPÍTULO 11 • DESENHAR O EVENT BUS INSTITUCIONAL (PUB/SUB INTERNO)

Um sistema profissional de trading **não usa variáveis partilhadas**, nem funções que chamam código diretamente de outros módulos.

Tudo é baseado em **eventos publicados** e **módulos subscritos**.

Isto permite:

✓ escalabilidade  
✓ modularidade  
✓ testabilidade  
✓ paralelismo  
✓ desacoplamento total

O Event Bus é, por isso, o componente central que liga tudo.

Vamos desenhá-lo agora.

#### 1. O QUE É UM EVENT BUS?

É uma estrutura que permite:

• publicar eventos  
• subscrever eventos  
• distribuir eventos para vários consumidores  
• manter o sistema desacoplado

O publisher **não sabe** quem vai receber o evento.  
O subscriber **não sabe** quem o originou.

Exemplo:

O conector IBKR publica um TradeEvent.

Quem recebe?

• delta\_engine  
• footprint\_engine  
• UI (Times & Sales)  
• logger  
• replay\_gravador

Tudo isto sem dependências entre módulos.

#### 2. REQUISITOS PROFISSIONAIS PARA UM EVENT BUS

O Event Bus precisa de cumprir requisitos institucionais:

✓ não bloquear  
✓ ser thread-safe  
✓ permitir múltiplos subscritores  
✓ ser leve (baixa latência)  
✓ permitir priorização se necessário  
✓ evitar perda de eventos  
✓ ter backpressure controlado  
✓ ter capacidade de logging  
✓ permitir shutdown limpo

Vamos construir isto em etapas.

#### 3. ESCOLHER O MODELO DE EVENT BUS

Existem três modelos possíveis:

**MODELO A • Síncrono (bloqueante)**

→ Simples, mas bloqueia o sistema.  
→ Não adequado a real-time trading.

**MODELO B • Assíncrono baseado em fila**

→ Cada evento entra numa queue.  
→ Workers tratam do evento.  
→ Bom para:

• baixa latência  
• dividir trabalho  
• não bloquear UI

**MODELO C • Pub/Sub com callbacks diretos**

→ Mínima latência possível  
→ Simples, leve, direto  
→ Ideal para dados em tempo real

**A solução ideal é um híbrido:**

**Pub/Sub com distribuição síncrona muito rápida**,  
+  
um pequeno buffer (queue) para proteger contra overload.

Vamos chamar-lhe:

**EventBusHybrid**

#### 4. ARQUITETURA DO EVENT BUS

Estrutura:

core/

event\_bus.py

Dentro deste ficheiro, teremos:

• dicionário de subscritores  
• método subscribe  
• método publish  
• fila opcional para backpressure  
• thread de processamento  
• mecanismo de shutdown

Representação simples:

event\_bus = EventBusHybrid()

event\_bus.subscribe("trade", meu\_callback)

event\_bus.publish("trade", TradeEvent(...))

Um tópico pode ter múltiplos subscritores.

#### 5. CRIAÇÃO DO EVENT BUS – ESTRUTURA BÁSICA

A versão inicial:

**import** threading

**from** collections **import** defaultdict

**from** queue **import** Queue

**from** typing **import** Callable

**class** EventBus:

**def** \_\_init\_\_(**self**):

**self**.subscribers: dict[str, list[Callable]] = defaultdict(list)

**self**.queue = Queue()

**self**.running = **True**

**self**.worker = threading.Thread(target=**self**.\_run, daemon=**True**)

**self**.worker.start()

**def** subscribe(**self**, topic: str, callback: Callable):

**self**.subscribers[topic].append(callback)

**def** publish(**self**, topic: str, **event**):

**self**.queue.put((topic, **event**))

**def** \_run(**self**):

**while** **self**.running:

topic, **event** = **self**.queue.**get**()

**for** callback **in** **self**.subscribers.**get**(topic, []):

**try**:

callback(**event**)

**except** Exception **as** e:

**print**(f"Erro no handler de evento [{topic}]: {e}")

**def** stop(**self**):

**self**.running = **False**

Este event bus já:

✓ permite pub/sub  
✓ distribui eventos  
✓ funciona com threads  
✓ evita bloqueios  
✓ garante desacoplamento

Será melhorado mais tarde com logging, priorização e métricas.

#### 6. PUBLICAÇÃO DE EVENTOS

Sempre que o conector IBKR recebe algo:

event\_bus.publish("trade", TradeEvent(...))

event\_bus.publish("dom\_update", DomUpdateEvent(...))

event\_bus.publish("tick", TickEvent(...))

O event bus faz o dispatch automático.

#### 7. SUBSCRIÇÃO DE EVENTOS

Módulos internos usam:

event\_bus.subscribe("trade", **self**.on\_trade)

event\_bus.subscribe("dom\_update", **self**.on\_dom)

event\_bus.subscribe("tick", **self**.on\_tick)

Cada módulo recebe **apenas o que lhe interessa**.

Principais subscritores:

• dom\_engine  
• delta\_engine  
• footprint\_engine  
• cvd\_engine  
• signal\_engine  
• UI  
• logger  
• data\_recorder  
• execution\_engine

#### 8. SEGURANÇA E THREAD-SAFETY

O EventBus que desenhámos:

✔ é thread-safe (Queue() cuida disso)  
✔ é não bloqueante  
✔ permite concurrency real  
✔ permite expansão fácil

Nunca vamos permitir:

✘ escrever diretamente em estruturas partilhadas sem thread-safety  
✘ processar eventos pesados no callback (usar workers)  
✘ bloquear a thread principal

#### 9. GESTÃO DE SOBRECARGA (BACKPRESSURE)

Em períodos de alta volatilidade, podem chegar:

• centenas de mensagens por segundo  
• dezenas de updates por tick  
• centenas de atualizações de DOM num único segundo

Para evitar:

• perda de frames na UI  
• lags  
• congestão no event bus

Implementamos estratégia:

**→ processar todos os eventos, mas ignorar updates repetidos irrelevantes**

**→ snapshot periódicos para a UI**

**→ aglomerar eventos quando necessário (debounce)**

**→ workers para processar cálculos pesados**

Exemplo: delta\_engine pode usar uma fila própria para cálculo.

#### 10. EVENT BUS COM MÉTRICAS INTERNAS

Em aplicações institucionais, precisamos de medir:

• eventos por segundo  
• latência de publicação  
• número de subscritores  
• backlog na queue  
• tempo de processamento médio

Mais tarde podemos adicionar:

**self**.metrics.total\_events += 1

**self**.metrics.events\_by\_topic[topic] += 1

**self**.metrics.queue\_size = **self**.queue.qsize()

E publicar métricas:

event\_bus.publish("system\_metrics", MetricsEvent(...))

A UI pode ter uma página para mostrar:

• velocidade de entrada do mercado  
• stress na aplicação  
• saúde geral do sistema

#### 11. EVENT BUS E ARQUITETURA DA UI

A UI vai subscrever:

• tick

• dom\_snapshot  
• footprint  
• delta  
• cvd  
• signals

Quando um evento chega, a UI atualiza apenas o componente relevante.

Nunca vamos fazer:

✘ polling  
✘ leitura direta do estado  
✘ cálculos na thread de UI

A UI vive dos eventos.

#### 12. EVENT BUS E EXECUÇÃO

O execution engine subscreve:

• signals  
• order\_status  
• execution\_details

E publica:

• order\_request  
• order\_cancel  
• risk\_alert

O conector IBKR subscreve:

• order\_request  
• order\_cancel

E publica:

• order\_status  
• execution\_details

A arquitetura fica completamente modular.

#### 13. EVENT BUS PARA REPLAY

Durante replay:

• carregamos eventos gravados  
• publicamos no event bus exatamente como se fossem live  
• o engine reage  
• a UI reage  
• o código não precisa de ser alterado

Isto só é possível porque tudo é orientado a eventos.

#### 14. RESUMO DO EVENT BUS PROFISSIONAL

O Event Bus:

✓ desacopla tudo  
✓ reduz complexidade  
✓ aumenta capacidade de teste  
✓ permite replay  
✓ suporta alta frequência  
✓ torna a aplicação escalável  
✓ permite UI reativa  
✓ simplifica o conector  
✓ permite execução algorítmica segura  
✓ é a espinha dorsal do sistema

Sem um event bus bem projetado, a app seria um caos de chamadas diretas.

##### O QUE VEM A SEGUIR

Agora temos:

• IBKR  
• modelos de dados  
• event bus

O próximo passo é começar a construir os motores internos que analisam o mercado.

## 📕 PARTE V • ORDERFLOW ENGINE

### CAPÍTULO 12 • DOM ENGINE (LIVRO DE ORDENS INSTITUCIONAL)

O DOM (Depth of Market) é o **mapa de liquidez** do mercado.  
É nele que se vê:

✓ ordens resting  
✓ liquidez escondida  
✓ manipulações (spoofing)  
✓ absorções pesadas  
✓ retirada rápida de ordens  
✓ deslocações do livro  
✓ intenção dos grandes players

Por isso, o DOM Engine é um dos módulos mais importantes da aplicação.

Vamos construí-lo de forma institucional.

#### 1. O QUE O DOM ENGINE DEVE FAZER

O DOM Engine recebe **DomUpdateEvents** em tempo real e constrói um **DomSnapshot** consistente.

Ele deve:

• manter o estado completo do livro  
• atualizar apenas os níveis afetados  
• reconstruir o DOM após falhas  
• detetar operações (insert/update/delete)  
• ordenar níveis corretamente  
• emitir eventos de snapshot periódico  
• fornecer dados para UI e módulos internos  
• alimentar lógica de imbalances  
• alimentar módulo de spoofing  
• alimentar módulo de absorção  
• alimentar footprint

A latência tem de ser mínima.

#### 2. COMO O DOM É TRANSMITIDO (IBKR, Rithmic, dxFeed)

Independente do feed:

• DOM chega como atualizações incrementais  
• cada update contém:

price → preço  
size → quantidade  
side → BID ou ASK  
operation → insert/update/delete  
position → nível do livro

Nenhum feed envia o DOM completo sempre.  
O engine deve **reconstruí-lo incrementalmente**.

#### 3. REPRESENTAÇÃO DO ESTADO DO DOM

O estado interno será guardado como dois arrays:

bids: list[DomLevel]

asks: list[DomLevel]

Onde cada DomLevel tem:

• price  
• size  
• side  
• position

Ordenação:

• bids em ordem decrescente por preço  
• asks em ordem crescente por preço

#### 4. ESTRUTURA BÁSICA DO DOM ENGINE

**class** DomEngine:

**def** \_\_init\_\_(**self**, event\_bus, depth=10):

**self**.depth = depth

**self**.bids = [**None**] \* depth

**self**.asks = [**None**] \* depth

**self**.event\_bus = event\_bus

event\_bus.subscribe("dom\_update", **self**.on\_dom\_update)

O engine subscreve automaticamente o tópico "dom\_update".

#### 5. PROCESSAMENTO DE UM DOM UPDATE

O callback recebe um DomUpdateEvent e executa três passos:

1. selecionar o lado correto (bids ou asks)
2. aplicar operação (insert/update/delete)
3. reorganizar níveis se necessário

#### 6. LÓGICA PARA OPERAÇÃO INSERT

**if** **event**.operation == DomOperation.INSERT:

levels = **self**.bids **if** **event**.side == DomSide.BID **else** **self**.asks

levels[**event**.position] = DomLevel(

price=**event**.price,

size=**event**.size,

side=**event**.side,

position=**event**.position

)

Simples: insere o nível naquela posição.

#### 7. LÓGICA PARA UPDATE

**if** **event**.operation == DomOperation.UPDATE:

levels = **self**.bids **if** **event**.side == DomSide.BID **else** **self**.asks

**if** levels[**event**.position]:

levels[**event**.position].price = **event**.price

levels[**event**.position].size = **event**.size

Atualiza preço e tamanho.

#### 8. LÓGICA PARA DELETE

**if** **event**.operation == DomOperation.DELETE:

levels = **self**.bids **if** **event**.side == DomSide.BID **else** **self**.asks

levels[**event**.position] = **None**

Remove liquidez.

#### 9. EMITIR SNAPSHOTS PERIÓDICOS

Snapshots servem para:

• alimentar UI  
• alimentar o módulo de signals  
• gravar DOM para replay

Estratégia:

✓ emitir snapshot a cada X ms  
✓ ou quando houver mudanças suficientes

Exemplo:

**def** publish\_snapshot(**self**):

snapshot = DomSnapshot(

ts=datetime.utcnow(),

symbol="GC",

depth=**self**.depth,

bids=[lvl **for** lvl **in** **self**.bids **if** lvl],

asks=[lvl **for** lvl **in** **self**.asks **if** lvl],

)

**self**.event\_bus.publish("dom\_snapshot", snapshot)

#### 10. DETETAR PULLING E STACKING

Com snapshots consecutivos:

**Pulling → liquidez removida**

• nível desaparece  
• size reduz drasticamente

**Stacking → liquidez adicionada**

• size aumenta agressivamente

O engine compara o snapshot atual com o anterior.

Criamos um evento:

**class** LiquidityChangeEvent(BaseEvent):

price: **float**

change: **float**

side: DomSide

type: str # pulling ou stacking

#### 11. DETETAR SPOOFING

Sinais clássicos:

1. Alguém adiciona muita liquidez (stacking intenso)
2. O preço afasta-se
3. A liquidez desaparece instantaneamente (pulling + large)

O DOM Engine guarda histórico curto dos níveis e identifica padrões.

Se encontrado → publica:

SignalEvent(signal\_type="spoofing", details=...)

#### 12. DETETAR ABSORÇÃO (MUITO IMPORTANTE)

Absorção ocorre quando:

• muitos trades atingem o nível  
• o nível permanece  
• size diminui lentamente  
• liquidez é defendida

Precisamos de:

• DOM Engine → níveis atuais  
• Delta Engine → agressão  
• Footprint Engine → volume por nível

Quando isto acontece → BUY/SELL signal institucional.

#### 13. CONEXÃO COM O FOOTPRINT

O DOM Engine:

• fornece níveis  
• footprint engine agrega trades por nível  
• footprint mostra absorção, imbalance e delta  
• DOM complementa footprint com resting liquidity

A ligação DOM ↔ footprint é fundamental para signals.

#### 14. PUBLICAÇÃO DE EVOLUÇÃO DO LIVRO PARA UI

A UI quer:

• níveis atuais  
• movimentos  
• alterações rápidas  
• heatmap

DOM Engine publica dois tipos:

• dom\_update (incremental)  
• dom\_snapshot (completo)

A UI atualiza:

• ordem das colunas  
• heatmap  
• gráficos time & price

#### 15. FUTURAS EXTENSÕES DO DOM ENGINE

Para a parte avançada do livro:

• tracking de grandes ordens  
• iceberg detection  
• multi-depth aggregation  
• reconstrução horizontal  
• clusters de liquidez  
• heatmap super-resolução  
• níveis institucionais estáticos vs dinâmico

##### ✔ RESUMO DO CAPÍTULO 12

O DOM Engine:

✓ mantém estado completo do livro  
✓ processa atualizações incrementais  
✓ detecta pulling / stacking  
✓ detecta spoofing  
✓ detecta absorção  
✓ publica snapshots  
✓ alimenta UI  
✓ fornece contexto institucional  
✓ integra com delta e footprint  
✓ é base dos sinais institucionais

Este módulo é essencial para o teu bot atingir **Layer 2.5**.

### CAPÍTULO 13 • DELTA ENGINE (DELTA, CVD, SPEED OF TAPE)

O Delta Engine é o módulo que responde a três perguntas fundamentais:

1. Quem está a bater no mercado neste momento: compradores ou vendedores?
2. Com que intensidade essa agressão está a acontecer ao longo do tempo?
3. A velocidade do fluxo está a aumentar, a estabilizar ou a colapsar?

Ele faz isso através de três pilares:

• Delta  
• CVD (Cumulative Volume Delta)  
• Speed of Tape

#### 1. O QUE É O DELTA NA PRÁTICA

Delta é, de forma simples:

**Delta = Volume agressor no Ask − Volume agressor no Bid**

Onde:

• Volume agressor no Ask → ordens de compra que batem no ask  
• Volume agressor no Bid → ordens de venda que batem no bid

Interpretação:

• Delta positivo → compradores estão a agredir mais do que vendedores  
• Delta negativo → vendedores estão a agredir mais do que compradores

O Delta Engine transforma uma sequência de TradeEvents em:

• Delta por tick  
• Delta por intervalo (por exemplo, 1 segundo, 100 ms, 1 minuto)  
• Delta por vela (para footprint)

#### 2. TRADEEVENT COMO BASE DO DELTA ENGINE

Relembrando o modelo:

**class** TradeEvent(BaseEvent):

price: **float**

size: **float**

aggressor: AggressorSide # BUY, SELL, UNKNOWN

exchange: str | **None** = **None**

seq\_num: **int** | **None** = **None**

O Delta Engine subscreve o tópico "trade":

event\_bus.subscribe("trade", **self**.on\_trade)

Cada TradeEvent é processado em tempo real.

#### 3. ESTRUTURA INTERNA DO DELTA ENGINE

O delta engine precisa de:

• um acumulador de volume buy e sell por intervalo  
• um acumulador global para o CVD  
• um relógio interno para fechar intervalos  
• buffers para cálculo de speed of tape

Estrutura base:

**class** DeltaEngine:

**def** \_\_init\_\_(**self**, event\_bus, symbol, interval\_ms=1000):

**self**.event\_bus = event\_bus

**self**.symbol = symbol

**self**.interval\_ms = interval\_ms

**self**.current\_buy\_volume = 0.0

**self**.current\_sell\_volume = 0.0

**self**.current\_trades\_count = 0

**self**.cvd\_value = 0.0

**self**.interval\_start\_ts = **None**

event\_bus.subscribe("trade", **self**.on\_trade)

#### 4. PROCESSAMENTO DE CADA TRADE

Para cada TradeEvent:

1. validar o símbolo
2. verificar o lado agressor
3. acumular volume
4. atualizar CVD
5. contar trades
6. decidir se o intervalo fecha

Exemplo conceptual:

**def** on\_trade(**self**, **event**: TradeEvent):

**if** **event**.symbol != **self**.symbol:

**return**

**if** **self**.interval\_start\_ts **is** **None**:

**self**.interval\_start\_ts = **event**.ts

**if** **event**.aggressor == AggressorSide.BUY:

**self**.current\_buy\_volume += **event**.size

delta\_increment = **event**.size

**elif** **event**.aggressor == AggressorSide.SELL:

**self**.current\_sell\_volume += **event**.size

delta\_increment = -**event**.size

**else**:

delta\_increment = 0.0

**self**.cvd\_value += delta\_increment

**self**.current\_trades\_count += 1

**if** **self**.\_should\_close\_interval(**event**.ts):

**self**.\_close\_interval(**event**.ts)

#### 5. FECHAR INTERVALOS DE DELTA

Os intervalos podem ser definidos por:

• tempo (por exemplo, 1 segundo)  
• número de trades (por exemplo, 100 trades)  
• tamanho de volume (por exemplo, 1000 contratos)

Neste livro vamos usar tempo como padrão, porque é mais simples e universal.

Função para verificar se o intervalo acabou:

**def** \_should\_close\_interval(**self**, current\_ts):

elapsed\_ms = (current\_ts - **self**.interval\_start\_ts).total\_seconds() \* 1000

**return** elapsed\_ms >= **self**.interval\_ms

Ao fechar o intervalo:

1. calcula delta do período
2. cria um DeltaEvent
3. cria um CvdEvent
4. faz publish para o event bus
5. reinicia acumuladores

**def** \_close\_interval(**self**, end\_ts):

delta\_value = **self**.current\_buy\_volume - **self**.current\_sell\_volume

delta\_event = DeltaEvent(

ts=end\_ts,

symbol=**self**.symbol,

interval\_start=**self**.interval\_start\_ts,

interval\_end=end\_ts,

delta=delta\_value,

buy\_volume=**self**.current\_buy\_volume,

sell\_volume=**self**.current\_sell\_volume,

trades\_count=**self**.current\_trades\_count,

)

cvd\_event = CvdEvent(

ts=end\_ts,

symbol=**self**.symbol,

cvd\_value=**self**.cvd\_value,

last\_delta=delta\_value,

direction=**self**.\_direction\_from\_delta(delta\_value),

)

**self**.event\_bus.publish("delta", delta\_event)

**self**.event\_bus.publish("cvd", cvd\_event)

**self**.current\_buy\_volume = 0.0

**self**.current\_sell\_volume = 0.0

**self**.current\_trades\_count = 0

**self**.interval\_start\_ts = end\_ts

#### 6. INTERPRETAÇÃO PROFISSIONAL DO DELTA

O delta num único intervalo diz:

• se o fluxo agressivo está numa direção  
• qual o desequilíbrio de força naquele período

Padrões importantes:

**Delta forte positivo com deslocamento de preço**

• confirma movimento de alta saudável  
• indica compradores no controlo  
• aumenta probabilidade de continuação

**Delta forte negativo com deslocamento de preço**

• confirma movimento de baixa saudável  
• indica vendedores no controlo

**Delta forte mas preço parado**

• indica absorção  
• alguém está a bater mas o outro lado está a defender  
• é um dos sinais mais importantes de inversão

**Delta fraco com grande deslocamento de preço**

• indica provável movimento algorítmico com pouca agressão  
• pode ser manipulação ou zonas de baixa liquidez

#### 7. CVD (CUMULATIVE VOLUME DELTA)

O CVD é simplesmente:

**CVD atual = CVD anterior + delta do período**

Interpretação:

• CVD a subir → dominância de agressão compradora  
• CVD a descer → dominância de agressão vendedora  
• CVD flat → equilíbrio de agressão

O CVD torna o delta legível ao longo do tempo, tal como uma média móvel torna retornos mais estáveis.

Padrões clássicos:

**Preço sobe mas CVD cai**

• divergência  
• indica que o movimento de preço não é sustentado por agressão real  
• pode sinalizar exaustão ou manipulação

**Preço cai mas CVD sobe**

• compradores a absorver quedas  
• posição forte de compradores escondidos  
• frequentemente antecipa inversões violentas

#### 8. SPEED OF TAPE

Speed of tape é a velocidade a que aparecem trades no T and S.

Podemos medir:

• número de trades por segundo  
• volume agressor por segundo  
• número de eventos TradeEvent num intervalo curto

O Delta Engine pode acumular:

**self**.recent\_trades\_timestamps.append(**event**.ts)

E num processo interno:

1. remover timestamps mais antigos que X milissegundos
2. contar quantos restam
3. calcular trades por segundo

Exemplo conceptual:

**from** collections **import** deque

**from** datetime **import** timedelta

**class** DeltaEngine:

**def** \_\_init\_\_(...):

...

**self**.recent\_trades\_timestamps = deque()

**self**.speed\_window = timedelta(milliseconds=500)

**def** on\_trade(...):

...

**self**.recent\_trades\_timestamps.append(**event**.ts)

**self**.\_cleanup\_old\_trades(**event**.ts)

**self**.\_publish\_speed(**event**.ts)

**def** \_cleanup\_old\_trades(**self**, now):

**while** **self**.recent\_trades\_timestamps **and** now - **self**.recent\_trades\_timestamps[0] > **self**.speed\_window:

**self**.recent\_trades\_timestamps.popleft()

**def** \_publish\_speed(**self**, now):

trades\_count = len(**self**.recent\_trades\_timestamps)

speed\_event = SpeedOfTapeEvent(

ts=now,

symbol=**self**.symbol,

trades\_per\_window=trades\_count,

window\_ms=**self**.speed\_window.total\_seconds() \* 1000,

)

**self**.event\_bus.publish("speed\_of\_tape", speed\_event)

Este valor é o "rpm" do mercado.

#### 9. COMO O DELTA ENGINE ALIMENTA A UI

A UI pode subscrever:

• "delta"  
• "cvd"  
• "speed\_of\_tape"

E desenhar:

• gráfico de barras de delta por intervalo  
• linha contínua de CVD  
• indicador de intensidade do fluxo (por exemplo, uma barra que aumenta com speed of tape)

Isto permite ao trader ver:

• quão agressivo está o mercado  
• se o fluxo vai na direção do preço  
• se há divergência entre preço e agressão  
• se o mercado está a acelerar ou a travar

#### 10. COMO O DELTA ENGINE ALIMENTA SINAIS

O Signal Engine, mais à frente, vai consumir:

• DeltaEvent  
• CvdEvent  
• SpeedOfTapeEvent

E gerar sinais como:

**Sinal de exaustão**

• delta extremamente alto  
• speed of tape a aumentar  
• pouco deslocamento de preço

**Sinal de rompimento saudável**

• delta consistentemente forte na direção do movimento  
• CVD alinhado e a subir ou descer de forma clara

**Sinal de falha de rompimento**

• preço rompe uma zona  
• delta não confirma ou inverte  
• CVD não acompanha

Estes sinais são o núcleo da leitura institucional automatizada.

#### 11. DETALHES INSTITUCIONAIS DE IMPLEMENTAÇÃO

Em ambientes profissionais, o Delta Engine deve:

• suportar múltiplos símbolos ao mesmo tempo  
• permitir múltiplos intervalos em paralelo (por exemplo, 100 ms, 1 segundo, 1 minuto)  
• ser parametrizável via config  
• ser eficiente, sem recalcular coisas desnecessárias  
• ser testado com dados gravados para garantir que o comportamento é determinístico

Em capítulos mais avançados, podemos introduzir:

• normalização por tamanho médio de trade  
• filtros de ruído  
• integração com volume profile  
• combinação com DOM para sinais de absorção mais fortes

#### 12. RESUMO DO CAPÍTULO 13

O Delta Engine é o módulo que:

✓ transforma trades em delta  
✓ soma delta ao longo do tempo para gerar CVD  
✓ mede speed of tape  
✓ publica eventos de delta, CVD e velocidade  
✓ fornece contexto de força real do mercado  
✓ alimenta a UI institucional  
✓ alimenta o Signal Engine  
✓ é essencial para identificar absorção, exaustão e rompimentos verdadeiros

Com o DOM Engine e o Delta Engine, já tens:

• mapa de liquidez (DOM)  
• mapa de agressão (Delta e CVD)  
• mapa de velocidade (speed of tape)

No próximo passo, vamos juntar isto numa visão ainda mais rica.

### CAPÍTULO 14 • FOOTPRINT ENGINE (VOLUME POR NÍVEL, IMBALANCES, PONTO DE CONTROLO)

O footprint é o raio-X de cada vela.  
Enquanto o gráfico normal mostra:

• open, high, low, close  
• volume agregado

o footprint mostra:

**→ volume comprador por nível**

**→ volume vendedor por nível**

**→ delta por nível**

**→ desequilíbrios (imbalances)**

**→ absorção**

**→ posição institucional de defesa**

**→ zonas de agressão concentrada**

**→ ponto de controlo (POC: nível com maior volume)**

**→ clusters de volume escondido**

Este capítulo constrói o engine que transforma **TradeEvents** em velas footprint completas.

#### 1. COMO O FOOTPRINT FUNCIONA NA PRÁTICA

O Footprint Engine agrupa trades por **nível de preço**, dentro de uma vela definida por tempo (ou número de trades/vol).

Para cada nível de preço, precisa saber:

• volume agressor BID  
• volume agressor ASK  
• delta  
• desequilíbrio ask/bid (por exemplo, 3:1, 4:1, 10:1)  
• se existe absorção (agressão forte sem deslocamento de preço)

No final do intervalo, ele cria:

**FootprintCandleEvent**  
que contém:

• open  
• high  
• low  
• close  
• volume total  
• lista de FootprintBucket (um por preço)  
• POC  
• delta máximo  
• delta mínimo

#### 2. A ESTRUTURA DE DADOS BASE (DO CAP. 10)

Usamos:

FootprintBucket:

price

bid\_volume

ask\_volume

delta

imbalance\_ratio

is\_imbalance

is\_absorption

e

FootprintCandleEvent:

interval\_start

interval\_end

open, high, low, close

total\_volume

buckets

point\_of\_control

max\_delta

min\_delta

#### 3. ESTRATÉGIA PARA CONSTRUIR O FOOTPRINT ENGINE

Passos:

**1. Abrir uma vela**

– marcar timestamp  
– armazenar O/H/L/C temporários  
– criar um dicionário interno para volumes por preço

**2. Processar trades em tempo real**

– cada trade contribui para o nível de preço correspondente  
– se agressor == BUY → ask\_volume += size  
– se agressor == SELL → bid\_volume += size  
– atualizar delta  
– atualizar extremos de preço

**3. Fechar a vela**

– calular POC  
– detetar imbalances  
– detetar absorções  
– calcular delta extremos  
– publicar FootprintCandleEvent

**4. Reiniciar tudo para a próxima vela**

#### 4. REPRESENTAÇÃO INTERNA DE CADA VELA

O engine precisa de uma estrutura interna:

**class** FootprintEngine:

**def** \_\_init\_\_(**self**, event\_bus, symbol, interval\_ms=1000):

**self**.event\_bus = event\_bus

**self**.symbol = symbol

**self**.interval\_ms = interval\_ms

**self**.current\_buckets = {} # key: price, value: FootprintBucket

**self**.interval\_start\_ts = **None**

**self**.open\_price = **None**

**self**.high\_price = **None**

**self**.low\_price = **None**

**self**.close\_price = **None**

event\_bus.subscribe("trade", **self**.on\_trade)

#### 5. PROCESSAMENTO DE CADA TRADE (A CONSTRUÇÃO DO FOOTPRINT)

Quando um TradeEvent chega:

##### 5.1 Iniciar vela se for a primeira trade

**if** **self**.interval\_start\_ts **is** **None**:

**self**.interval\_start\_ts = **event**.ts

**self**.open\_price = **event**.price

**self**.high\_price = **event**.price

**self**.low\_price = **event**.price

##### 5.2 Atualizar O/H/L/C

**self**.high\_price = max(**self**.high\_price, **event**.price)

**self**.low\_price = min(**self**.low\_price, **event**.price)

**self**.close\_price = **event**.price

##### 5.3 Gerir buckets por preço

Criar bucket se não existir:

bucket = **self**.current\_buckets.**get**(**event**.price)

**if** bucket **is** **None**:

bucket = FootprintBucket(price=**event**.price,

bid\_volume=0.0,

ask\_volume=0.0,

delta=0.0)

**self**.current\_buckets[**event**.price] = bucket

##### 5.4 Acumular volume por lado agressor

**if** **event**.aggressor == AggressorSide.BUY:

bucket.ask\_volume += **event**.size

bucket.delta += **event**.size

**elif** **event**.aggressor == AggressorSide.SELL:

bucket.bid\_volume += **event**.size

bucket.delta -= **event**.size

#### 6. FECHAR A VELA (O MOMENTO MAIS IMPORTANTE)

Quando o intervalo termina:

**if** **self**.\_should\_close\_interval(**event**.ts):

**self**.\_close\_interval(**event**.ts)

Fechar a vela envolve:

1. ordenar buckets por preço
2. detetar imbalances
3. detetar absorções
4. calcular POC
5. calcular delta extremos
6. publicar o evento

#### 7. LÓGICA DE DESEQUILÍBRIOS (IMBALANCES)

Imbalances clássicos:

● preço[i].ask / preço[i-1].bid > 3:1  
→ desequilíbrio comprador

● preço[i].bid / preço[i+1].ask > 3:1  
→ desequilíbrio vendedor

Para cada bucket:

buckets = sorted(**self**.current\_buckets.values(), key=**lambda** b: b.price)

**for** i, b **in** enumerate(buckets):

**if** i > 0:

lower = buckets[i-1]

**if** b.ask\_volume > lower.bid\_volume \* imbalance\_threshold:

b.is\_imbalance = **True**

b.imbalance\_ratio = b.ask\_volume / lower.bid\_volume

**if** i < len(buckets) - 1:

upper = buckets[i+1]

**if** b.bid\_volume > upper.ask\_volume \* imbalance\_threshold:

b.is\_imbalance = **True**

b.imbalance\_ratio = b.bid\_volume / upper.ask\_volume

#### 8. DETEÇÃO DE ABSORÇÃO

Absorção ocorre quando:

● delta muito agressor  
● preço não avança  
● volume grande concentrado num nível  
● DOM mostra resting resistente

Footprint sozinho pode sugerir absorção se:

● muita agressão BUY mas preço não sobe  
● muita agressão SELL mas preço não desce

No bucket:

**if** bucket.delta > large\_value **and** **self**.close\_price <= bucket.price:

bucket.is\_absorption = **True**

Integração com DOM Engine (cap. 12) melhora isto.

#### 9. CALCULAR PONTO DE CONTROLO (POC)

POC = nível de preço com maior volume total

poc = max(buckets, key=**lambda** b: b.bid\_volume + b.ask\_volume)

O POC revela:

● nível onde o volume se concentrou  
● interesse institucional  
● zona de defesa ou distribuição

#### 10. CALCULAR DELTA EXTREMOS

max\_delta = max(b.delta **for** b **in** buckets)

min\_delta = min(b.delta **for** b **in** buckets)

Sinais:

● max\_delta muito alto → agressão concentrada em alta  
● min\_delta muito baixo → agressão concentrada em baixa

#### 11. PUBLICAÇÃO DA VELA FOOTPRINT

Criamos o evento final:

footprint\_event = FootprintCandleEvent(

ts=end\_ts,

symbol=**self**.symbol,

interval\_start=**self**.interval\_start\_ts,

interval\_end=end\_ts,

open\_price=**self**.open\_price,

high\_price=**self**.high\_price,

low\_price=**self**.low\_price,

close\_price=**self**.close\_price,

total\_volume=sum(b.bid\_volume + b.ask\_volume **for** b **in** buckets),

buckets=buckets,

point\_of\_control=poc.price,

max\_delta=max\_delta,

min\_delta=min\_delta,

)

Publicação:

**self**.event\_bus.publish("footprint", footprint\_event)

#### 12. COMO A UI CONSOME O FOOTPRINT

A UI institucional desenha:

● footprint vertical (vela)  
● footprint horizontal (perfil por preço)  
● imbalances coloridas  
● absorção marcada  
● POC destacado  
● delta por nível  
● volume por nível  
● clusters iluminados

Isto permite:

● identificar zonas de rejeição  
● identificar zonas de aceitação  
● ver quando compradores perdem força  
● ver quando vendedores absorvem quedas  
● detectar compressões institucionais  
● analisar breakouts reais

#### 13. COMO O FOOTPRINT ENGINE ALIMENTA SINAIS

Sinais típicos alimentados pelo footprint:

**1. Reversão institucional**

— delta extremo numa zona + absorção forte  
— candle footprint com imbalances invertidos

**2. Continuação**

— delta alinhado e crescente  
— imbalances consistentes

**3. Falha de breakout**

— agressão forte sem deslocamento

**4. Setup de absorção (bid absorb / ask absorb)**

**5. Clusters institucionais**

— volume anormalmente alto num nível  
— defesa clara de um preço

**6. Região de negociação (auctioning)**

— delta fraco mas muito volume  
— consolidação institucional

O Signals Engine (cap. 15) vai combinar tudo isto.

#### 14. INTEGRAÇÃO COM O DELTA ENGINE E DOM ENGINE

Agora tens 3 motores a trabalhar em conjunto:

**● DOM Engine**

— resting liquidity  
— pulling  
— stacking  
— spoofing

**● Delta Engine**

— agressão  
— direção  
— velocidade

**● Footprint Engine**

— volume por nível  
— desequilíbrios  
— absorção  
— clusters

Juntos formam a **microestrutura completa da vela**.

##### 15. RESUMO DO CAPÍTULO 14

O Footprint Engine:

✓ agrupa volume por nível  
✓ determina agressão por nível  
✓ deteta imbalances  
✓ deteta absorções  
✓ calcula POC  
✓ calcula delta máximo / mínimo  
✓ cria velas footprint completas  
✓ alimenta UI avançada  
✓ alimenta sinais institucionais  
✓ integra com DOM e delta engines

É um dos módulos mais importantes de toda a Edição II.

### CAPÍTULO 15 • SIGNAL ENGINE (ABSORÇÃO, SPOOFING, IMBALANCES, BURSTS, FALHA DE BREAKOUT)

#### 1. O PAPEL DO SIGNAL ENGINE

Cada componente anterior faz algo específico:

• DOM Engine → resting liquidity, pulling/stacking, spoofing  
• Delta Engine → agressão, direção, speed of tape  
• Footprint Engine → volume por nível, imbalances, POC, absorção

Mas o trader institucional não opera **raw data**.  
Opera **interpretações**.

O Signal Engine é o módulo que:

✓ recebe eventos de outros engines  
✓ aplica regras institucionais  
✓ decide se existe um sinal válido  
✓ emite um **SignalEvent**  
✓ envia para UI e Execution Engine

Ele funciona como uma camada de “inteligência”.

#### 2. ARQUITETURA DO SIGNAL ENGINE

**class** SignalEngine:

**def** \_\_init\_\_(**self**, event\_bus, symbol):

**self**.symbol = symbol

**self**.event\_bus = event\_bus

# estado vindo de outros engines

**self**.last\_delta = **None**

**self**.last\_cvd = **None**

**self**.last\_dom\_snapshot = **None**

**self**.last\_footprint = **None**

**self**.last\_speed = **None**

# subscrições

event\_bus.subscribe("delta", **self**.on\_delta)

event\_bus.subscribe("cvd", **self**.on\_cvd)

event\_bus.subscribe("dom\_snapshot", **self**.on\_dom\_snapshot)

event\_bus.subscribe("footprint", **self**.on\_footprint)

event\_bus.subscribe("speed\_of\_tape", **self**.on\_speed)

O módulo regista eventos “passivamente”.  
Quando as condições certas alinham → dispara um sinal.

#### 3. TIPOLOGIAS DE SINAIS A IMPLEMENTAR

O manual institucional incluirá sinais:

**✔ Absorção (Bid Absorb / Ask Absorb)**

**✔ Spoofing (Fake Liquidity)**

**✔ Imbalance Extreme (pressão esmagadora)**

**✔ Burst (explosão de agressão)**

**✔ Falha de Breakout (fakeout institucional)**

**✔ Continuação Validada (delta + speed confirmam)**

**✔ Reversão Institucional (absorção + divergência CVD)**

**✔ Exhaustion Trap (exaustão + baixa velocidade)**

**✔ Liquidity Grab + Reversal (varrimento + sinal interno)**

Vamos estruturar cada um.

#### 4. SINAL #1 — ABSORÇÃO (Ask Absorb / Bid Absorb)

**Conceito institucional:**

Absorção é quando:

● muito volume agressor chega a um nível  
● o nível **não é ultrapassado**  
● resting liquidity no DOM defende o preço  
● footprint mostra delta elevado sem deslocamento  
● CVD desacelera

Sinal:

→ ask absorb → potencial reversão para baixo  
→ bid absorb → potencial reversão para cima

**Regras simples (versão 1):**

**if** fp.delta muito positivo e close == high

e DOM mostra resting que não desaparece

e speed\_of\_tape diminui:

sinal = absorção vendedora

**if** fp.delta muito negativo e close == low

e DOM mostra resting que não desaparece

e speed\_of\_tape diminui:

sinal = absorção compradora

**Em código conceitual:**

**def** detect\_absorption(**self**):

fp = **self**.last\_footprint

dom = **self**.last\_dom\_snapshot

delta = **self**.last\_delta

**if** **not** fp **or** **not** dom **or** **not** delta:

**return** **None**

# absorção vendedora (no topo)

**if** fp.close\_price == fp.high\_price **and** fp.max\_delta > fp.total\_volume \* 0.3:

**if** dom\_has\_strong\_resting\_at\_price(fp.close\_price):

**return** SignalEvent(signal\_type="absorption\_sell", ...)

# absorção compradora (no fundo)

**if** fp.close\_price == fp.low\_price **and** fp.min\_delta < -fp.total\_volume \* 0.3:

**if** dom\_has\_strong\_resting\_at\_price(fp.close\_price):

**return** SignalEvent(signal\_type="absorption\_buy", ...)

Absorção é um dos sinais mais fortes do orderflow institucional.

#### 5. SINAL #2 — SPOOFING (Fake Liquidity)

Critérios:

● grande liquidez aparece repentinamente  
● preço afasta-se  
● liquidez desaparece instantaneamente  
● footprint não mostra volume relevante naquela zona

**Regras institucionais:**

**if** DOM stacking agressivo

e preço move-se na direção contrária

e DOM pulling forte logo depois:

sinal = spoofing

Exemplo:

**def** detect\_spoofing(**self**):

dom = **self**.last\_dom\_snapshot

**if** **not** dom:

**return** **None**

**if** detect\_large\_stacking(dom) **and** detect\_sudden\_pulling(dom):

**return** SignalEvent(signal\_type="spoofing", ...)

Spoofing é um sinal crítico para identificar manipulação algorítmica.

#### 6. SINAL #3 — IMBALANCE EXTREME (3:1, 4:1, 10:1)

Quando um desequilíbrio maciço acontece no footprint:

● ask >> bid em vários níveis consecutivos → pressão compradora  
● bid >> ask → pressão vendedora

**Regras:**

**if** muitos buckets com is\_imbalance=**True**

e direção **do** imbalance é consistente:

sinal = imbalance\_buy ou imbalance\_sell

Exemplo:

num\_imbalances\_up = sum(b.is\_imbalance **and** b.delta > 0 **for** b **in** fp.buckets)

**if** num\_imbalances\_up >= 3:

**return** SignalEvent(signal\_type="imbalance\_buy", ...)

Serve para validar rompimentos e movimentos institucionais.

#### 7. SINAL #4 — BURST (Explosão de Fluxo)

Quando speed\_of\_tape aumenta muito:

● número de trades por segundo explode  
● delta aumenta rapidamente  
● DOM retraí-se (liquidity vacuum)

**Regras:**

**if** speed\_of\_tape > threshold\_high

e delta é muito alto numa direção:

signal = burst\_buy ou burst\_sell

Exemplo:

**def** detect\_burst(**self**):

**if** **self**.last\_speed.trades\_per\_window > X **and** abs(**self**.last\_delta.delta) > Y:

direction = "buy" **if** **self**.last\_delta.delta > 0 **else** "sell"

**return** SignalEvent(signal\_type="burst\_" + direction, ...)

Este sinal identifica:

◆ rompimentos verdadeiros  
◆ entradas de players grandes  
◆ aceleração institucional

#### 8. SINAL #5 — FALHA DE BREAKOUT (Fakeout Institucional)

Um dos padrões mais lucrativos:

● preço rompe um nível  
● delta confirma inicialmente  
● depois delta reverte  
● speed\_of\_tape reduz  
● footprint mostra absorção no topo/fundo  
● DOM mostra falta de follow-through

**Regras:**

**if** rompe\_high

e delta positivo perde **for**ça

e absorção aparece **no** topo:

sinal = breakout\_fail\_sell

**if** rompe\_low

e delta negativo perde **for**ça

e absorção aparece **no** fundo:

sinal = breakout\_fail\_buy

Este é um dos sinais mais usados por prop firms.

#### 9. COMO O SIGNAL ENGINE GERA EVENTOS

Quando um sinal é detetado:

signal = SignalEvent(

ts=now,

symbol=**self**.symbol,

signal\_type="absorption\_buy",

price=fp.close\_price,

severity=0.9,

details={

"delta": **self**.last\_delta.delta,

"cvd": **self**.last\_cvd.cvd\_value,

"poc": fp.point\_of\_control

}

)

**self**.event\_bus.publish("signal", signal)

A UI mostra o sinal, e o Execution Engine pode agir.

#### 10. ORDEM DE PROCESSAMENTO

O Signal Engine processa eventos numa hierarquia:

**1) Absorção**

**2) Spoofing**

**3) Burst**

**4) Imbalances**

**5) Falha de breakout**

**6) Continuação validada**

Sinais mais importantes têm prioridade mais alta.

#### 11. COMO O SIGNAL ENGINE ALIMENTA EXECUÇÃO

Rules:

**Comprar quando:**

• absorção no fundo + divergência CVD  
• burst comprador após consolidação  
• imbalance compradores + DOM forte

**Vender quando:**

• spoofing buy + pulling violento  
• absorção no topo + exaustão delta  
• burst vendedor + DOM fraco

O Execution Layer decide:

✓ enviar ordem no MT5  
✓ enviar ordem na IBKR  
✓ enviar ordem nos dois (multi-broker)

#### 12. REFINAMENTO INSTITUCIONAL (VERSÃO AVANÇADA)

Mais tarde adicionaremos:

• machine learning leve  
• score institucional por sinal  
• classificação probabilística  
• weighting entre engines  
• normalização por volatilidade  
• detecção de divergências footprint × DOM  
• análises bayesianas de follow-through

Mas por agora a estrutura base é implementada.

##### 13. RESUMO DO CAPÍTULO 15

O Signal Engine:

✓ recebe eventos de DOM, delta, footprint e speed  
✓ aplica regras institucionais  
✓ deteta absorção, spoofing, imbalances, bursts  
✓ identifica falhas de breakout (padrão mais lucrativo)  
✓ publica SignalEvents  
✓ alimenta UI e execução  
✓ é o “cérebro decisional” da app institucional

Com este módulo, a aplicação começa a pensar como um trader institucional real — não como um bot retail.

## 📕 PARTE VI • EXECUÇÃO PROFISSIONAL

### CAPÍTULO 16 • EXECUTION ENGINE (IBKR, MT5, MULTI-BROKER)

Este é o capítulo onde tudo converge:

● Os sinais vêm do Signal Engine  
● A decisão vem da lógica interna  
● A ordem vai para:  
→ Interactive Brokers (Futuros GC, ES, NQ, CL…)  
→ MetaTrader 5 (CFDs XAUUSD, Indices, FX)  
→ Ou **ambos simultaneamente**

O Execution Engine implementa:

✓ controle de risco  
✓ validação de mercado  
✓ regras de proteção  
✓ submissão de ordens  
✓ monitorização  
✓ logging  
✓ failover (se um broker falhar, o outro mantém)

Esta arquitetura é **idêntica à usada em mesas institucionais e quant desks**.

#### 1. OBJETIVO DO EXECUTION ENGINE

Um sinal institucional NUNCA deve enviar ordem direta sem:

● filtrar ruído  
● validar liquidez  
● confirmar contexto  
● validar risco  
● normalizar tamanho da posição  
● decidir onde executar  
● construir ordens corretas  
● esperar confirmação  
● monitorizar até o fim da trade

Por isso, o Execution Engine é dividido em submódulos:

**✔ Execution Router (decide onde executar)**

**✔ IBKRExecution (futuros reais)**

**✔ MT5Execution (CFDs)**

**✔ Risk Manager (valida tamanho, SL, TP, risco diário)**

**✔ Order Builder (constrói ordens)**

**✔ Position Monitor (vigia a posição até fechar)**

#### 2. ARQUITETURA GERAL

Estrutura:

ExecutionEngine

├── BrokerExecution

│ ├── IBKRExecution

│ ├── MT5Execution

├── RiskManager

├── OrderBuilder

├── PositionMonitor

├── ExecutionRouter

O Execution Engine é desencadeado por:

event\_bus.subscribe("signal", **self**.on\_signal)

Quando um SignalEvent chega:

1. Validar
2. Decidir broker
3. Gerar ordem
4. Enviar
5. Confirmar
6. Monitorizar
7. Fechar quando necessário

#### 3. RECEBER UM SINAL E DECIDIR A AÇÃO

Pseudo-código:

**def** on\_signal(**self**, signal: SignalEvent):

**if** **not** **self**.risk\_manager.is\_allowed(signal):

**return**

broker = **self**.router.choose\_broker(signal)

order = **self**.order\_builder.build(signal, broker)

exec\_client = **self**.executors[broker]

exec\_client.submit\_order(order)

**self**.position\_monitor.track(order)

#### 4. EXECUTION ROUTER

*(Escolher entre IBKR e MT5)*

**Por defeito:**

● IBKR → para sinais de microestrutura  
● MT5 → para sinais derivados de GC em XAUUSD

Configuração opcional:

● executar nos dois  
● executar só se spread < limite  
● executar se liquidez suficiente

Exemplo:

**def** choose\_broker(**self**, signal):

**if** signal.symbol == "GC" **and** **self**.config.execute\_in\_mt5:

**return** "mt5"

**if** signal.importance > 0.9:

**return** "ibkr"

**return** "mt5"

Mais tarde podemos fazer:

• arbitragem parcial  
• execução híbrida  
• failover automático (se IBKR falhar → MT5 assume)

#### 5. ORDER BUILDER

Constrói ordens:

● Market  
● Limit  
● Stop  
● Bracket (SL + TP embutidos)  
● OCO (one cancels the other)  
● Adaptive/MKT (IBKR)

Campos:

symbol

side (BUY/SELL)

quantity

type

limit\_price

stop\_price

sl\_price

tp\_price

time\_in\_force

Exemplo:

**def** build(**self**, signal, broker):

price = signal.price

size = **self**.calc\_position\_size(signal)

**if** signal.signal\_type **in** ["burst\_buy", "imbalance\_buy"]:

order\_type = "market"

**else**:

order\_type = "limit"

**return** Order(

broker=broker,

symbol=**self**.map\_symbol(signal.symbol, broker),

side="BUY" **if** signal.is\_buy **else** "SELL",

quantity=size,

type=order\_type,

limit\_price=price,

stop\_loss=**self**.calc\_sl(price, signal),

take\_profit=**self**.calc\_tp(price, signal)

)

#### 6. IBKR EXECUTION CLIENT

Usa a API oficial:

**FUNCIONALIDADES:**

✓ EClient → enviar ordens  
✓ EWrapper → confirmar  
✓ SmartRouting  
✓ GTC / DAY  
✓ Execução em futuros CME (GC, ES, NQ…)

Exemplo conceptual:

**class** IBKRExecution:

**def** submit\_order(**self**, order: Order):

ib\_order = IBOrder()

ib\_order.action = order.side

ib\_order.totalQuantity = order.quantity

ib\_order.orderType = order.type.upper()

ib\_order.lmtPrice = order.limit\_price

ib\_order.auxPrice = order.stop\_price

ib\_order.tif = "GTC"

**self**.ib.placeOrder(order.id, order.contract, ib\_order)

Confirmar:

**def** orderStatus(...):

**if** status == "Filled":

event\_bus.publish("order\_filled", ...)

#### 7. MT5 EXECUTION CLIENT

Usa **MetaTrader5** Python module.

**Comprar:**

mt5.order\_send({

"action": mt5.TRADE\_ACTION\_DEAL,

"symbol": symbol,

"type": mt5.ORDER\_TYPE\_BUY,

"volume": qty,

"price": mt5.symbol\_info\_tick(symbol).ask,

"sl": sl,

"tp": tp

})

**Vender:**

type": mt5.ORDER\_TYPE\_SELL,

"price": mt5.symbol\_info\_tick(symbol).bid

MT5 é excelente para CFD XAUUSD, mesmo que os sinais venham do GC (futuros).

#### 8. RISK MANAGER (OBRIGATÓRIO EM SISTEMAS INSTITUCIONAIS)

Controla:

● tamanho máximo por ordem  
● risco diário (%)  
● drawdown acumulado  
● número de trades por minuto  
● rejeitar duplicações  
● valida spreads  
● valida volatilidade no momento

Pseudo-código:

**def** is\_allowed(**self**, signal):

**if** account\_drawdown > max\_dd:

**return** **False**

**if** num\_trades\_last\_min > limit:

**return** **False**

**if** signal.price\_too\_fast:

**return** **False**

**return** **True**

#### 9. POSITION MONITOR

Após uma ordem ser executada:

● monitoriza PnL  
● ajusta SL para break-even  
● trailing stop  
● fecha metade  
● fecha total se sinal contrário aparecer  
● integração com DOM (se resting desaparece → fechar)

Estrutura:

on order\_filled → start monitoring

every tick → evaluate conditions

on TP **or** SL → close

on signal reversal → close

#### 10. EXECUÇÃO MULTI-BROKER (ARB-LITE)

O bot pode:

**■ sinal BUY GC**

**■ abrir BUY em XAUUSD (MT5)**

**■ abrir BUY também no GC (IBKR)**

Ou apenas num.

Ou fasear execução.

Ou usar “execution priority”:

1. IBKR
2. fallback → MT5

#### 11. LOGGING PROFISSIONAL

Cada ordem gera:

● ID interno  
● ID IBKR ou MT5  
● timestamp envio  
● timestamp execução  
● slippage  
● broker usado  
● preço de entrada  
● preço de saída  
● motivo de fechamento  
● tipo de sinal

Este log alimenta:

● relatório de performance  
● backtesting de sinais  
● métricas institucionais  
● modelos futuros

##### 12. RESUMO DO CAPÍTULO 16

O Execution Engine:

✓ recebe sinais do Signal Engine  
✓ valida risco  
✓ escolhe broker (IBKR, MT5, ou ambos)  
✓ constrói ordens institucionais  
✓ envia ordens e confirma execução  
✓ monitoriza a posição até ao fim  
✓ aplica trailing, break-even e regras institucionais  
✓ suporta failover e multi-broker  
✓ cria logs auditáveis  
✓ integra perfeitamente com a microestrutura real

Este módulo é literalmente o que diferencia um sistema **institucional** de um bot retail MT5.

### Capítulo 17 – UI Institucional (Heatmap, Footprint, DOM, Delta, CVD)

Este capítulo descreve, em detalhe, a construção de uma interface de nível institucional que permite visualizar:

* a liquidez (DOM + Heatmap)
* a agressão (Delta, CVD)
* a microestrutura interna da vela (Footprint)
* o fluxo de negócios (Times & Sales)
* os sinais do bot em tempo real
* o estado geral da execução (IBKR, MT5, risco, ordens ativas)

Esta UI é um dos pilares da plataforma institucional que estamos a construir.  
Ela não é “gráficos bonitos”: é uma ferramenta operacional para leitura do mercado ao estilo **Bookmap, Jigsaw, Sierra Chart, Exocharts**.

Vamos estruturar o capítulo com máxima clareza.

#### 1. Objetivo da UI Institucional

A UI serve três propósitos fundamentais:

**A) Leitura de mercado**

Permite interpretar:

* liquidez resting
* absorção
* spoofing
* agressão real
* microestrutura por nível
* imbalances
* delta por intervalo
* divergências CVD
* bursts de fluxos

**B) Decisão**

Permite confirmar ou rejeitar sinais gerados pelo bot.

**C) Controlo operacional**

Permite:

* monitorizar risco
* acompanhar ordens
* interagir manualmente com o mercado
* observar comportamento do bot em tempo real

Uma boa UI institucional reduz 80% da carga cognitiva de leitura do mercado.

#### 2. Princípios de Design Institucional

**1) Dark Mode como base**

Tudo é melhor em dark mode, por:

* menor fadiga ocular
* contraste mais limpo com heatmap
* facilidade de destaque dos volumes

**2) Alta densidade de informação, bem organizada**

Contém muito conteúdo, mas sem poluição visual.

**3) Estabilidade**

A UI nunca deve “tremer”, reposicionar objetos ou saltar.

**4) Atualizações suaves**

No máximo 10–30 FPS.  
Sem congelamentos, sem picos de CPU.

**5) Painéis independentes**

DOM, footprint, delta, CVD, T&S e sinais são módulos isolados que consomem o event bus.

**6) Layout multi-monitor**

A UI deve poder destacar janelas em monitores diferentes.

#### 3. Layout Global da Plataforma

Um layout recomendado (inspirado nas mesas institucionais reais):

──────────────────────────────────────────────────────────────

BARRA SUPERIOR (estado da aplicação)

──────────────────────────────────────────────────────────────

FEED IBKR / RITHMIC / dxFeed

EXEC IBKR / MT5

MODE LIVE / PAPER / REPLAY

SYMBOL GC / ES / NQ / XAUUSD

BOT ON / OFF

RT RISK DD%, nº trades, risco restante

TIME Hora **local** / exchange

──────────────────────────────────────────────────────────────

PAINEL PRINCIPAL (microestrutura)

──────────────────────────────────────────────────────────────

| DOM + HEATMAP | FOOTPRINT | DELTA + CVD |

──────────────────────────────────────────────────────────────

PAINEL INFERIOR (fluxo + sinais + execução)

──────────────────────────────────────────────────────────────

TIMES & SALES | SIGNALS ENGINE LOG | ORDERS & POSITIONS

──────────────────────────────────────────────────────────────

Cada painel é independente e comunicará via event bus.

#### 4. Painel DOM + Heatmap

*(Livro de ordens com mapa de liquidez, estilo Bookmap)*

**Objetivo**

Mostrar:

* liquidez resting em cada nível
* liquidez que aparece/desaparece (stacking / pulling)
* spoofing
* absorção identificada
* profundidade real da ordem

**Elementos principais:**

**1) Ladder de preços**

Uma coluna vertical central com todos os níveis visíveis.  
O preço atual está sempre fixo numa linha horizontal destacada.

**2) Bid / Ask**

* coluna esquerda → quantidade resting no BID
* coluna direita → quantidade resting no ASK

**3) Heatmap de liquidez**

O heatmap representa a **quantidade resting** nos níveis:

* cores mais fortes = mais liquidez
* atualiza-se em tempo real
* não se reordena (o preço que se move, não a ladder)

**4) Marcações institucionais automáticas**

O DOM Engine publica:

* pulling
* stacking
* spoofing
* zonas protegidas
* níveis absorvidos

A UI deve marcar estes níveis com:

* contorno
* ícone discreto
* banda lateral

**5) Interação**

* scroll vertical
* pan suave
* hover para detalhes do nível
* clique para colocar ordens manuais no MT5 ou IBKR

#### 5. Painel Footprint

*(Volume por nível, delta, imbalances)*

O footprint é um painel de extrema importância.

**Cada vela footprint contém:**

* open, high, low, close
* buckets por nível de preço
* volume no bid
* volume no ask
* delta por nível
* imbalances (3:1, 4:1, extreme)
* absorção (se sinalizada pelo Footprint Engine)
* ponto de controlo (POC)

**Formato visual recomendado:**

Price BidVol AskVol Delta

-------------------------------------

1875.0 120 340 +220

1874.5 230 150 -80

1874.0 310 410 +100

**Destaques visuais:**

* imbalances: cor mais intensa
* absorção: fundo distinto
* POC: linha horizontal ou highlight
* volume total por vela: barra vertical à esquerda

**Interação:**

* zoom horizontal (nº de velas)
* zoom vertical (range de preços)
* clicar numa vela mostra detalhes completos
* highlight sincronizado com DOM e delta

#### 6. Painel Delta + CVD

*(Confirmação de agressão e divergência)*

**Delta Plot**

Um gráfico de barras:

* verde → delta positivo
* vermelho → delta negativo
* altura → força da agressão

**CVD Plot**

Uma linha contínua:

* sobe com delta positivo
* desce com delta negativo
* suporta divergências institucionais

**Indicadores adicionais:**

* speed of tape (linha secundária)
* marcação vertical quando há sinais importantes

#### 7. Times & Sales

*(Fluxo de negócios real)*

T&S é a “lista de prints” que mostra quem está a bater e onde.

**Colunas:**

* timestamp
* preço
* volume
* agressor (ask ou bid)
* eventualmente exchange

**Codificação visual:**

* verde forte → compra agressiva
* vermelho forte → venda agressiva
* volume grande → fonte maior ou highlight

**Função:**

* confirmar bursts
* confirmar imbalances
* detectar absorção (quando prints grandes não movem preço)
* perceber se um movimento é “limpo” ou manipulado

#### 8. Painel de Sinais Institucionais

Aqui aparecem:

* sinais de absorção
* sinais de spoofing
* sinais de burst
* sinais de imbalance extremo
* falha de breakout
* reversão institucional

Cada entrada contém:

* hora
* tipo de sinal
* severidade
* engines que contribuíram
* preço
* direção (BUY/SELL)
* broker sugerido (IBKR, MT5 ou ambos)

A UI deve permitir:

* clicar para centrar o mercado nesse momento
* filtrar por tipo
* ordenar por severidade

#### 9. Painel de Execução e Posições

**Posições Ativas**

Mostrar:

* símbolo
* preço médio
* quantidade
* broker
* PnL
* tempo na posição
* SL e TP atuais

**Ordens Pendentes**

Mostrar:

* tipo (limit, stop, bracket)
* preços alvo
* estado (pendente, parcial, completo)
* severidade do sinal que deu origem à ordem

**Controlo manual**

Permitir:

* fechar posição
* mover SL e TP diretamente na UI
* cancelar ordens pendentes
* executar ordem manual (mercado/limit)

#### 10. Arquitetura Técnica da UI

A UI deve ser **event-driven** e não bloquear o core.

Cada painel subscreve o event bus:

* dom\_snapshot
* dom\_update
* footprint
* delta
* cvd
* speed\_of\_tape
* signal
* order\_status

**Threading**

A regra é:

* motores correm em threads próprias
* UI corre sempre no main thread
* comunicação é feita com buffer e debounce para evitar flood

**Stack Técnica**

O livro ensinará a UI em **PySide6 (Qt)** por motivos:

* Muito rápida
* Muito flexível
* Suporte nativo para renderização custom
* Multiplataforma
* Focada em alto desempenho

Outras alternativas serão apresentadas como anexos:

* Web UI (React + WebSocket)
* WPF (C#)
* Electron + Python backend

#### 11. Performance e Otimização

**1) Atualização com batching**

Nunca redesenhar tudo quando chega um evento.  
Agrupar eventos e desenhar apenas 10–30 vezes por segundo.

**2) Renderização incremental**

DOM ladder → apenas redesenhar níveis alterados.  
Footprint → redesenhar apenas vela atual.  
Delta/CVD → redesenhar último ponto.

**3) Pré-renderização e canvas**

Para o heatmap, usar buffers.  
Renderizar no GPU quando disponível.

**4) Memória**

Manter apenas dados necessários em contexto visível.  
Arquivar histórico num buffer separado.

#### 12. Personalização e Perfis

A UI deve permitir:

* escolher cores
* escolher intensidade do heatmap
* escolher número de níveis do DOM
* alterar timeframe do footprint
* ativar/desativar sinais
* escolher broker de execução primário
* guardar layouts personalizados

Perfis recomendados:

* “Scalping GC”
* “Institutional Replay”
* “Live Trading + Bot”
* “Analytics Only”

##### 13. Conclusão do Capítulo

Este capítulo definiu uma UI institucional completa, com:

* DOM + Heatmap (liquidez real)
* Footprint (estrutura da vela)
* Delta + CVD + Speed of Tape (agressão real)
* Times & Sales (fluxo granular)
* Painel de Sinais (decisão institucional)
* Painel de Execução e Controlo (IBKR + MT5)

A UI:

* consome eventos do event bus
* nunca bloqueia a aplicação
* é modular, escalável e orientada a decisão
* oferece visão institucional do mercado em tempo real

Com esta UI, tens não só um bot institucional,  
mas uma **plataforma profissional completa**.

## 📕 PARTE VII — FEEDS E ABSTRAÇÕES

### Capítulo 18 — Data Provider Abstraction Layer (Multi-Feed Institucional)

Este capítulo apresenta a arquitetura que permite que **qualquer fonte de dados** seja integrada no bot, mantendo sempre:

* a mesma interface
* os mesmos eventos
* a mesma estrutura
* sem modificações no DOM Engine, Delta Engine, Footprint Engine, UI ou Signal Engine

Isto é literalmente o que as plataformas institucionais fazem.

#### 1. Problema Sem a Data Provider Layer

Sem esta camada, a aplicação ficaria presa a um único fornecedor:

* só IBKR → limitado
* só Rithmic → difícil de integrar
* só dxFeed → não tem execução
* só MT5 → sem DOM nem footprint

Além disso:

* motores internos teriam código específico do feed
* maior risco de erros
* impossível alternar entre feeds
* impossível usar replay offline
* impossível testar sinais com histórico real
* impossível comparar IBKR vs Rithmic vs dxFeed

#### 2. Solução Institucional: Data Provider Abstraction Layer

A solução é criar **uma interface comum** chamada:

DataProvider

E cada feed implementa essa interface:

IBKRProvider

RithmicProvider

dxFeedProvider

MT5Provider

HistoricalReplayProvider

Todos os providers:

* ligam-se à fonte de dados
* transformam os dados crus
* publicam eventos padronizados
* são plugáveis e intercambiáveis

Com isto:

👉 O resto da aplicação NUNCA precisa saber qual feed está ligado.  
👉 Podes mudar de IBKR para Rithmic com uma linha no config.  
👉 Podes usar replay em vez de live sem mudar nada.

#### 3. Interface Base — DataProvider

Todos os providers devem implementar:

**class** DataProvider:

**def** connect(**self**): ...

**def** subscribe\_dom(**self**, symbol): ...

**def** subscribe\_trades(**self**, symbol): ...

**def** subscribe\_ticks(**self**, symbol): ...

**def** subscribe\_bars(**self**, symbol, timeframe): ...

**def** disconnect(**self**): ...

E todos devem **emitir eventos padronizados**:

* **DomUpdateEvent**
* **TradeEvent**
* **TickEvent**
* **BarEvent**
* **StatusEvent**

A uniformidade é a chave.

#### 4. Provider 1 — IBKR Data Provider

IBKR envia:

* DOM incremental (via updateMktDepth)
* T&S (via tick-by-tick)
* últimos preços (via realtime bars)
* snapshots (se necessário)

Funcionalidades:

**✔ Ligação ao TWS ou IB Gateway**

**✔ Subscrição de Market Depth**

**✔ Subscrição de Tick-by-Tick Trades**

**✔ Reconexão automática**

**✔ Heartbeat**

**✔ Publicação padronizada:**

DomUpdateEvent(...)

TradeEvent(...)

TickEvent(...)

**Mapeamento:**

* updateMktDepth → DomUpdateEvent
* tickByTickAllLast → TradeEvent
* tickPrice → TickEvent

#### 5. Provider 2 — Rithmic Data Provider (via Bridge)

Rithmic não tem API Python pública, mas podes ligá-lo via:

* ponte C# → WebSocket → Python
* ponte C++ → pybind11
* ponte R | Trader Pro export events

O livro irá ensinar a fazer uma ponte simples **WebSocket**, porque:

* é robusta
* é rápida
* não bloqueia Python
* Rithmic envia DOM + T&S + tick-by-tick de forma muito eficiente

A bridge envia mensagens JSON como:

{

"type": "dom",

"price": 1835.2,

"size": 150,

"side": "bid",

"operation": "update",

"position": 3

}

O Python recebe e converte para eventos internos.

Este provider dá-te:

**✔ feed profissional de microestrutura**

**✔ velocidade superior ao IBKR**

**✔ dados real-time das prop firms**

**✔ ideal para operar XAUUSD via GC**

#### 6. Provider 3 — dxFeed Provider (CME / US Stocks / Crypto)

dxFeed tem uma API **institucional**, com WebSocket e REST:

Oferece:

* DOM
* T&S
* tick-by-tick
* candles
* indicadores internos
* velocidade extremamente alta

Perfeito para:

* labs de microestrutura
* datasets históricos
* backtesting institucional
* construir modelos

Este provider implementa:

ws = websockets.connect("wss://dxfws.dxfeed.com")

E interpreta mensagens como:

{

"eventType": "Trade",

"symbol": "GCZ5",

"price": 1835.6,

"size": 4,

"exchange": "CME",

"ts": 1738870002

}

Publicando:

TradeEvent(...)

#### 7. Provider 4 — MT5 Provider

MT5 é muito limitado:

* não tem DOM verdadeiro
* não tem time & sales
* não tem microestrutura real

Mas:

* fornece ticks
* fornece candles históricos
* pode complementar IBKR/Rithmic para executar ordens

O MT5 Provider implementa apenas:

* TickEvent
* BarEvent

Este feed **não** serve para microestrutura.  
Mas é útil para operar CFDs com execução rápida.

#### 8. Provider 5 — Historical Replay Provider

Este provider usa ficheiros gravados:

* dom.csv
* trades.csv
* ticks.csv
* bars.csv

E reproduz a sessão ao vivo.  
Ideal para:

* estudar sessões reais
* treinar o bot
* testar sinais
* validar motores

Funciona com:

speed = 1x / 2x / 5x / 10x

E publica eventos como se fossem live.

#### 9. Switching de Feed com Configurações

O utilizador escolhe:

data\_provider: "ibkr"

# ou "rithmic"

# ou "dxfeed"

# ou "mt5"

# ou "historical"

E o sistema cria automaticamente:

**if** cfg.data\_provider == "ibkr":

provider = IBKRProvider(event\_bus)

**elif** cfg.data\_provider == "rithmic":

provider = RithmicProvider(event\_bus)

...

Sem tocar em nenhum motor interno.

#### 10. Como os Providers Alimentam os Motores

Todos os providers, independentemente da fonte, chamam:

event\_bus.publish("trade", TradeEvent(...))

event\_bus.publish("dom\_update", DomUpdateEvent(...))

event\_bus.publish("tick", TickEvent(...))

event\_bus.publish("bar", BarEvent(...))

E os motores:

* DOM Engine
* Delta Engine
* Footprint Engine
* Signal Engine
* UI

consomem esses eventos **sem alterações no código**.

Isto é a verdadeira força da arquitetura institucional.

#### 11. Robustez Institucional

**Todos os providers devem incluir:**

1. reconexão automática
2. buffer de eventos
3. heartbeat
4. watchdog
5. métricas (eventos por segundo, perdas, latência)
6. logging estruturado
7. isolamento (um provider não afeta outro)

##### 12. Resumo do Capítulo 18

A Data Provider Layer:

✓ Desacopla o bot das fontes de dados  
✓ Permite usar IBKR, Rithmic, dxFeed, MT5 ou histórico  
✓ Fornece uma interface comum  
✓ Publica eventos padronizados  
✓ Permite replay offline  
✓ Permite alternar feeds com uma linha de config  
✓ Torna o sistema totalmente institucional  
✓ Prepara o bot para ser escalado e distribuído  
✓ Permite ao leitor operar sem depósitos obrigatórios

Com esta camada, a tua aplicação deixa de ser “um bot ligado ao IBKR” e passa a ser uma **plataforma de microestrutura independente de fornecedor** — tal como as plataformas profissionais fazem.

### Capítulo 19 — Execution Abstraction Layer (IBKR, MT5, Dual Execution)

O propósito desta camada é desenhar uma API única para execução, que funciona independentemente do broker que estiver a ser utilizado.

Assim como o capítulo anterior criou a camada de **dados**, este cria a camada de **execução**.

#### 1. O problema sem a Execution Abstraction Layer

Sem esta camada, teríamos:

* código de execução misturado com lógica de trading
* sinais dependentes do broker
* necessidade de duplicar código para IBKR e MT5
* dificuldade em adicionar novos brokers no futuro
* impossibilidade de executar em múltiplos brokers ao mesmo tempo

Além disso:

Há diferenças entre IBKR e MT5

✔ tipos de ordem  
✔ campos obrigatórios  
✔ tempos de envio  
✔ limitações de volume  
✔ formatos de SL/TP

Sem abstração, o sistema quebraria facilmente.

#### 2. Objetivo da Execution Abstraction Layer

Criar uma interface comum:

ExecutionProvider

E cada broker terá a sua implementação:

IBKRExecutionProvider

MT5ExecutionProvider

PaperExecutionProvider

MultiExecutionProvider

Todos estes providers:

* recebem ordens no mesmo formato
* executam ordens
* monitorizam estados
* publicam eventos uniformes para o event bus

Com isto:

👉 A lógica de trading não sabe nem precisa de saber se a ordem vai para IBKR, MT5, ou ambos.

#### 3. Interface Base — ExecutionProvider

Formato universal:

**class** ExecutionProvider:

**def** connect(**self**): ...

**def** place\_order(**self**, order: Order): ...

**def** modify\_order(**self**, order\_id, new\_params): ...

**def** cancel\_order(**self**, order\_id): ...

**def** close\_position(**self**, position\_id): ...

**def** get\_positions(**self**): ...

**def** get\_open\_orders(**self**): ...

**def** disconnect(**self**): ...

Independentemente do broker, o bot usa estas funções.

#### 4. Estrutura do objeto “Order”

A aplicação cria ordens num formato neutro:

**class** Order:

id: str

symbol: str

side: str # "BUY" ou "SELL"

qty: **float**

type: str # "market", "limit", "stop", "bracket"

limit\_price: **float** | **None**

stop\_price: **float** | **None**

sl: **float** | **None**

tp: **float** | **None**

tif: str = "GTC" # Good Till Cancel

Esta ordem é passada ao provider escolhido.

O provider traduz para o formato do broker.

#### 5. Execution Provider #1 — IBKRExecutionProvider

IBKR suporta:

* market
* limit
* stop
* stop-limit
* bracket
* adaptive
* advanced order types

E a API exige:

* EClient.placeOrder(orderId, contract, order)
* EWrapper.orderStatus
* EWrapper.openOrder
* EWrapper.execDetails

O provider traduz:

**Ordem universal → Ordem IBKR**

Exemplo:

**def** place\_order(**self**, order: Order):

ib\_order = IBOrder()

ib\_order.action = order.side

ib\_order.totalQuantity = order.qty

ib\_order.orderType = **self**.\_map\_type(order.type)

ib\_order.lmtPrice = order.limit\_price

ib\_order.auxPrice = order.stop\_price

ib\_order.tif = order.tif

contract = **self**.\_build\_contract(order.symbol)

**self**.client.placeOrder(order.id, contract, ib\_order)

Confirmações:

**def** orderStatus(**self**, orderId, status, filled, remaining, avgFillPrice, ...):

**self**.event\_bus.publish("order\_status", OrderStatusEvent(...))

#### 6. Execution Provider #2 — MT5ExecutionProvider

MT5 suporta:

* market
* limit
* stop
* SL/TP diretos
* Execução de CFDs

O provider traduz ordens universais para **order\_send()**:

**def** place\_order(**self**, order: Order):

request = {

"action": mt5.TRADE\_ACTION\_DEAL **if** order.type == "market" **else** mt5.TRADE\_ACTION\_PENDING,

"symbol": order.symbol,

"type": **self**.\_map\_type(order.side, order.type),

"volume": order.qty,

"price": order.limit\_price **or** **self**.\_market\_price(order.side),

"sl": order.sl,

"tp": order.tp,

"deviation": 20

}

result = mt5.order\_send(request)

**self**.event\_bus.publish("order\_status", MT5OrderStatusEvent(...))

**MT5 limita muitos tipos de ordem**

Mas a camada de execução trata disso sem impacto na lógica do bot.

#### 7. Execution Provider #3 — PaperExecutionProvider

Para backtesting live-like e simulação realista.

* Não envia ordens reais
* Mas faz:

✔ preenchimento simulado  
✔ tracking de posições  
✔ PnL em tempo real  
✔ slippage configurável  
✔ latência simulada

Útil para:

* estudar comportamento do bot
* validar sinais
* simular execução multi-feed

#### 8. Execution Provider #4 — MultiExecutionProvider

*(execução em dois brokers ao mesmo tempo)*

Este provider recebe a ordem universal e envia:

* para IBKR
* para MT5

Ou para ambos dependendo da lógica.

Exemplo:

**def** place\_order(**self**, order):

results = []

**for** provider **in** **self**.active\_providers:

results.append(provider.place\_order(order))

**return** results

Uso ideal:

* sinais de GC → executar em XAUUSD CFD
* sinais muito fortes → executar nos dois
* failover: se IBKR falha, MT5 entra

Este provider é o que transforma o teu bot num **bot híbrido institucional + retail**.

#### 9. Execution Router (decisão de onde executar)

Este módulo escolhe qual provider usar.

Critérios possíveis:

1. Se o sinal vem de microestrutura GC →  
   executar em MT5 (XAUUSD CFD) E/OU em IBKR (GC).
2. Se liquidez estiver baixa →  
   preferir IBKR.
3. Se spread MT5 estiver alto →  
   evitar MT5.
4. Se IBKR estiver offline →  
   fallback automático para MT5.
5. Configuração manual:
   * só IBKR
   * só MT5
   * ambos

Exemplo:

**def** choose\_provider(**self**, signal):

**if** signal.importance > 0.9:

**return** "multi"

**if** signal.symbol == "GC" **and** cfg.execute\_in\_mt5:

**return** "mt5"

**return** "ibkr"

#### 10. Como a Execution Layer integra com o Event Bus

Quando uma ordem é enviada:

* provider envia “order\_submitted”
* broker responde → provider publica “order\_filled”
* position monitor publica “position\_update”
* sinais contrários podem fechar a posição
* UI atualiza em tempo real

Todos os módulos são desacoplados:

* sinais
* execução
* logging
* UI
* módulos de risco

Graças ao event bus.

#### 11. Risk Layer integrada com Execution Layer

Antes de enviar uma ordem:

* verificar risco diário
* verificar estrangulamento de volume
* aplicar sizing automático
* verificar nível de volatilidade
* verificar se existe posição aberta
* impedir overtrading
* bloquear ordens duplicadas

Se violar:

**return** ExecutionDeniedEvent(...)

E o bot **não executa nada**.

#### 12. Monitorização de Posições (PositionMonitor)

Cada posição ativa:

* é monitorizada tick a tick
* SL pode ser movido automaticamente
* trailing stop pode aplicar
* break-even automático pode ativar
* sinais contrários podem fechar posição
* DOM pode invalidar entrada (pulling extremo)
* lógica de hedge pode ser aplicada

A execução torna-se verdadeiramente institucional.

##### 13. Síntese do Capítulo 19

A Execution Abstraction Layer:

✓ permite executar ordens em qualquer broker  
✓ desacopla sinais de execução real  
✓ padroniza ordens em formato universal  
✓ simplifica integração com IBKR e MT5  
✓ permite execução híbrida e failover  
✓ integra com risk manager  
✓ permite paper trading e replay  
✓ torna o bot expansível para futuros brokers

Com esta camada, a aplicação deixa de ser apenas “um bot MT5” ou “um bot IBKR”.  
Torna-se uma **plataforma institucional com execução multi-broker**, tal como sistemas de routing usados em mesas profissionais.

### Capítulo 20 — Price Normalization & Cross-Market Mapping (GC → XAUUSD para MT5)

A tua aplicação vai receber **microestrutura real do ouro através dos futuros GC**, mas vais poder executar:

* ordens no MT5 (XAUUSD CFD)
* ordens no IBKR (GC, MGC)

Para isso, é obrigatório implementar uma camada de **Price Normalization**.

#### 1. O Problema: GC ≠ XAUUSD (mas ambos são ouro)

GC = Gold Futures da CME  
XAUUSD = Spot CFD (broker-dependent)

Embora ambos representem ouro, **os preços não são iguais**.

Diferenças chave:

**1. Contrato GC tem valor fixo (100 onças)**

Preço mostra valor por onça *multiplicado pelo contrato*.

Exemplo:  
GC = 2354.6 → representa $2354.6 por onça  
Mas cada tick vale $10.

**2. XAUUSD CFD depende do broker**

FPMarkets, IC Markets, RoboForex etc.  
Cada um faz ajustes próprios.

**3. Taxas de câmbio e liquidez**

O CFD é derivado do spot + liquidez sintética + spread do broker.

**4. Futuros seguem CME**

O CFD segue **liquidez OTC global**.

#### 2. Porque é que usar GC como feed para XAUUSD é preferível?

A resposta institucional:

✔ GC tem volume real  
✔ GC tem DOM  
✔ GC tem footprint  
✔ GC tem delta real  
✔ GC tem orderflow real  
✔ GC é o mercado institucional verdadeiro

Já XAUUSD:

✘ não tem DOM  
✘ não tem delta  
✘ não tem T&S  
✘ é manipulado pelo broker  
✘ depende do modelo interno da corretora

Portanto:

→ usar GC para análise  
→ usar XAUUSD para execução (alavancagem, spreads menores, contas sem margem alta)

É a estratégia perfeita.

#### 3. O Objetivo do Mapeamento GC → XAUUSD

O mapeamento pretende:

* converter um sinal GC numa execução XAUUSD
* ajustar preço
* ajustar SL/TP
* ajustar tamanho da posição
* garantir sincronização de movimentos
* lidar com deslizamentos inevitáveis nas diferenças entre mercados

#### 4. Fórmula de Normalização Básica

A relação entre GC e XAUUSD é altamente estável:

Preço\_GC ≈ Preço\_XAUUSD

Mas não é *exatamente* igual devido a:

* diferenças de liquidez
* horário do mercado
* diferença de pricing (CME vs FX spot)
* rollovers
* micro divergências momentâneas

Por isso usamos:

**Fórmula 1 — Normalização direta**

XAUUSD\_price ≈ GC\_price

Na prática, isto é suficiente 95% das vezes.

#### 5. Ajuste Refinado (Melhor Prática Institucional)

Para precisão institucional, aplicamos um **spread de correlação** usando uma média móvel curta:

mapping\_offset = EMA(GC\_price - XAUUSD\_price\_realtime)

Assim:

XAUUSD\_equivalent\_price = GC\_price - mapping\_offset

Este offset é:

* pequeno
* estável
* remove micro divergências

#### 6. Normalização de SL e TP

Se o sinal GC indica:

BUY → entry = 2354.2  
SL\_GC = 2352.8  
TP\_GC = 2357.0

Devemos mapear:

SL\_XAU = SL\_GC - mapping\_offset

TP\_XAU = TP\_GC - mapping\_offset

Ou seja, transformamos a posição espiritual no GC para uma posição prática no XAUUSD.

Se o offset for -0.80:

SL\_XAU ≈ 2353.6  
TP\_XAU ≈ 2357.8

Simples, consistente e funcional.

#### 7. Normalização do Tamanho da Posição (Volume)

GC tem:

* 1 contrato = 100 onças
* tick = $10
* movimento 1.0 ponto = $100

XAUUSD CFD tem:

* 1 lot = 100 onças (equivalente ao GC)
* 0.1 lot = 10 onças
* 0.01 lot = 1 onça

Portanto:

size\_xauusd\_lots = size\_gc\_contracts \* 1.0

Ou se for MGC (Micro Gold):

* MGC = 10 onças

size\_xauusd = size\_mgc \* 0.1

Assumimos sempre equivalência por unidade:

1 pip XAUUSD ≈ 1 pip GC

#### 8. Ajuste de Execução (Slippage esperado)

Devido às diferenças entre mercados:

* GC reage primeiro
* XAUUSD segue 10–50 ms depois
* brokers CFD adicionam spread
* execução é sempre um pouco atrasada

Portanto:

**Recomendação:**

Adicionar um buffer ao preço de entrada:

BUY:

entry\_xau = entry\_gc - slippage\_buffer

SELL:

entry\_xau = entry\_gc + slippage\_buffer

Valor típico do buffer (pequeno):

0.05 a 0.20 USD

#### 9. Confirmação de Contexto Entre Mercados

Antes de executar no MT5 com base no GC, validar:

1. preço XAUUSD está sincronizado
2. spread aceitável
3. não há gaps anormais mesmo que GC esteja estável
4. latência IBKR/Rithmic → Python → MT5 não está elevada

Este passo evita entradas erradas quando:

* XAUUSD “salta”
* GC está a negociar mas o CFD está parado
* CFD está a fechar candles diferentes

#### 10. Pipeline Completo do Mapeamento GC → MT5

1. Receber sinal institucional GC
2. Normalizar preço GC → XAUUSD
3. Ajustar SL/TP
4. Ajustar tamanho da posição
5. Verificar sincronização e spread
6. Construir ordem MT5 normalizada
7. Executar imediatamente
8. Monitorizar posição com base no GC e MT5
9. Fechar posição se GC invalidar

**Exemplo real:**

GC:  
2350.4 (sinal BUY)  
SL = 2348.9  
TP = 2354.0

Mapping offset: +0.30

XAUUSD execução:  
entry = 2350.1  
SL = 2348.6  
TP = 2353.7

E pronto.

#### 11. Mapeamento Inverso (MT5 → GC)

Se quiseres treinar sinais para GC com execução em IBKR, podes mapear XAUUSD para GC:

GC\_equivalent = XAUUSD\_price + offset

Isto permite:

* ver o CFD como se fosse o futuro
* backtestar sinais com dados do CFD
* usar feed XAU para operar MGC/GC

Não é comum, mas é possível.

#### 12. Como Implementar na Aplicação (Código Conceitual)

Criação do módulo:

**class** PriceNormalizer:

**def** \_\_init\_\_(**self**, symbol\_gc, symbol\_xau):

**self**.offset = 0.0

**self**.alpha = 0.2 # EMA smoothing

**def** update\_offset(**self**, price\_gc, price\_xau):

diff = price\_gc - price\_xau

**self**.offset = **self**.alpha \* diff + (1 - **self**.alpha) \* **self**.offset

**def** gc\_to\_xau(**self**, price\_gc):

**return** price\_gc - **self**.offset

**def** xau\_to\_gc(**self**, price\_xau):

**return** price\_xau + **self**.offset

Execução:

entry\_price = normalizer.gc\_to\_xau(signal.price)

sl = normalizer.gc\_to\_xau(signal.sl\_price)

tp = normalizer.gc\_to\_xau(signal.tp\_price)

##### 13. Resumo do Capítulo 20

A camada de **Price Normalization** permite:

✓ Usar microestrutura profissional (GC)  
✓ Executar no CFD (XAUUSD) de forma correta  
✓ Ajustar preços, SL e TP automaticamente  
✓ Ajustar tamanho da posição  
✓ Compensar diferenças entre mercados  
✓ Garantir validação contextual antes da entrada  
✓ Manter o bot institucional e retail sincronizados

É um componente obrigatório para qualquer bot híbrido.

### Capítulo 21 — Multi-Broker Architecture (IBKR + MT5 + Dual Execution Pipeline)

Este capítulo formaliza a arquitetura que permite ao teu sistema:

✔ analisar microestrutura no **GC (futuros CME)**  
✔ executar no **XAUUSD CFD (MT5)**  
✔ executar simultaneamente em **GC / MGC (IBKR)**  
✔ trabalhar com **vários brokers ao mesmo tempo**  
✔ fazer failover: se um broker falhar, o outro executa  
✔ suportar routing inteligente e gestão de risco unificada

Tudo isto mantendo:

* uma única API interna
* uma única lógica de sinais
* um único motor institucional

#### 1. Motivação e Porque Isto É Institucional

Uma mesa de trading profissional raramente trabalha com um único broker ou venue:

* futuros na CME (GC, ES, NQ)
* FX spot em bancos
* CFDs em LPs paralelos
* cash equities noutros venues
* routing inteligente
* failover em execução

O teu bot precisa de imitar esta robustez.

Por isso construímos:

**A) Data Provider Layer → múltiplos feeds (IBKR, Rithmic, dxFeed…)**

**B) Price Normalization Layer → GC → XAUUSD**

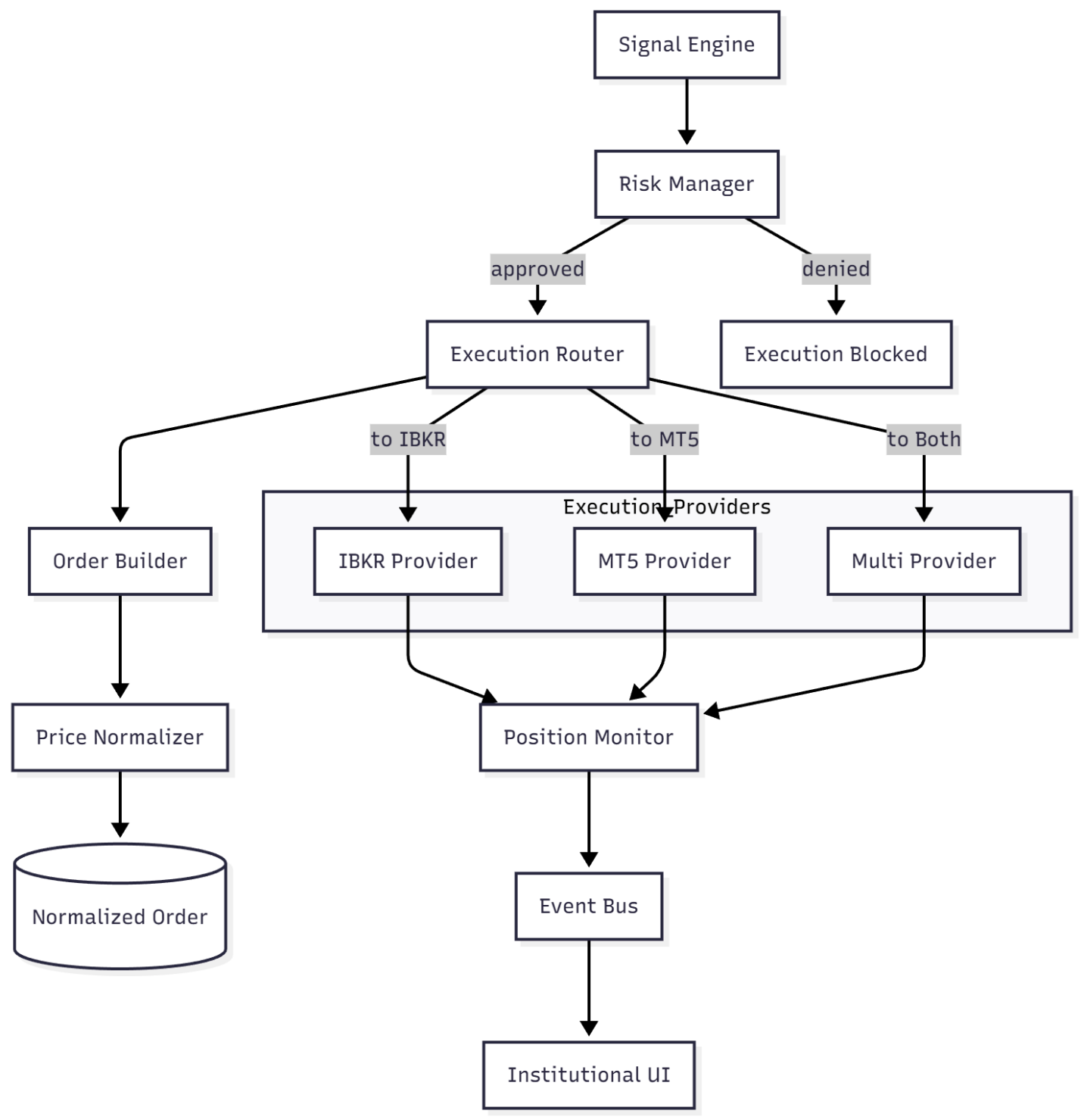
**C) Execution Abstraction Layer → brokers intercambiáveis**

**D) Multi-Broker Architecture → executar em múltiplos brokers, simultaneamente e com lógica inteligente**

Este capítulo detalha a peça final: **o pipeline multi-broker**.

#### 2. A Estrutura Geral da Multi-Broker Architecture

A arquitetura final é:



#### 3. Execution Router — O “cérebro” do pipeline

O Router recebe o **SignalEvent** e decide:

* executar no IBKR
* executar no MT5
* executar nos dois simultaneamente
* não executar (se risco bloquear)

Os critérios são configuráveis e institucionais:

**Critério 1 — Tipo de Sinal**

Se for sinal baseado em microestrutura pesada (GC Orderflow):

→ preferir MT5 (XAUUSD CFD) **e/ou** IBKR (GC)

**Critério 2 — Severidade**

Sinais fortes (absorção extrema, burst institucional):

→ executar nos dois brokers

**Critério 3 — Liquidez**

Se MT5 estiver com spread alto:

→ enviar apenas para IBKR

**Critério 4 — Failover**

Se IBKR estiver offline:

→ fallback para MT5

**Critério 5 — Configuração do utilizador**

Modo manual:

* only\_mt5
* only\_ibkr
* both
* auto

Código conceptual:

**def** route(**self**, signal):

**if** cfg.mode == "both":

**return** ["ibkr", "mt5"]

**if** cfg.mode == "auto":

**if** signal.importance > 0.9:

**return** ["ibkr", "mt5"]

**if** signal.symbol == "GC" **and** cfg.execute\_cfd:

**return** ["mt5"]

**return** ["ibkr"]

**if** cfg.mode == "mt5":

**return** ["mt5"]

**if** cfg.mode == "ibkr":

**return** ["ibkr"]

O router é o cérebro da execução multi-venue.

#### 4. Como o Router Interage com a Execution Layer

1. Recebe um sinal.
2. Pede ao RiskManager aprovação.
3. Seleciona os brokers de destino.
4. Envia a ordem universal para cada Execution Provider.

Exemplo:

brokers = router.route(signal)

for b in brokers:

exec\_provider[b].place\_order(order)

Simples, limpo, escalável.

#### 5. Dual Execution Pipeline — Execução simultânea

Quando o router devolve:

["ibkr", "mt5"]

A Execution Layer deve:

1. CRIAR UMA ÚNICA ORDEM UNIVERSAL
2. MAPEAR preço para XAUUSD (capítulo 20)
3. ENVIAR variável traduzida para MT5
4. ENVIAR variável original para IBKR
5. Monitorizar as duas posições separadamente
6. Controlar risco conjunto e individual

Resultado:

✔ GC e XAUUSD abrem juntos  
✔ Sinais institucionais baseados em GC validam as posições  
✔ Fechamentos podem ser independentes (ou sincronizados)

#### 6. Sincronização de Posições (Position Monitor Multi-Broker)

Quando duas posições existem ao mesmo tempo:

* uma posição **institucional real** (IBKR / GC)
* uma posição **retail derivada** (MT5 / XAUUSD CFD)

A PositionMonitor deve acompanhar ambas:

• PnL separado  
• SL/TP separado  
• execução de regras avançadas (como break-even)  
• um sinal contrário deve fechar ambas as posições  
• se uma falhar, a outra continua

Exemplo:

GC gera sinal SELL → entra MT5 SELL e IBKR SELL  
GC dá sinal BUY reversal → fecha ambas imediatamente

#### 7. Hedge e Relações Inversas

O bot pode aprender a:

* abrir CFD a favor e futuro contra para hedge
* cobrir risco de eventos
* capturar divergências de micro execução

Exemplo avançado (opcional no livro):

GC mostra absorção vendedora  
XAUUSD mostra atraso na quebra  
→ bot entra SHORT em MT5  
→ bot coloca ordem limit BUY em GC num nível institucional (hedge parcial)

#### 8. Failover Automático (Arquitetura Institucional)

Se um broker falhar:

❌ IBKR desconecta  
→ encaminhar tudo para MT5  
→ guardar sinais pendentes  
→ auto reconectar IBKR

❌ MT5 falha  
→ executar apenas no IBKR  
→ tentar reconectar em paralelo

O utilizador nunca perde sinais.

#### 9. Módulo de Logging e Auditoria Multi-Broker

Cada ordem universal gera:

1. ordem interna (ID universal)
2. ordem IBKR (com ID IBKR)
3. ordem MT5 (ticket MT5)
4. logs individuais e combinados
5. rastreamento:

* preço de envio
* preço de execução
* slippage
* latência
* motivo da entrada
* motivo da saída

Isto é obrigatório para:

* auditoria
* estudos de caso
* debugging
* validação institucional

#### 10. Painéis de UI Específicos para Multi-Broker

A UI deve apresentar:

* estado IBKR (ligado/desligado)
* estado MT5 (ligado/desligado)
* modo de routing ativo
* ordens ativas por broker
* risco por broker
* PnL por broker

Algo como:

┌──────────────────────────────┐

│ BROKER STATUS │

│ IBKR: CONNECTED (GC) │

│ MT5: CONNECTED (XAUUSD.r) │

│ ROUTER MODE: AUTO │

│ EXECUTION: DUAL │

└──────────────────────────────┘

O trader sabe sempre:

* onde o bot entrou
* onde vai entrar
* em que broker está exposto

#### 11. Exemplo Real de Execução Dual

GC dá sinal BUY a 2350.20.

Router recebe:

["ibkr", "mt5"]

A) IBKR  
→ Ordem BUY 1 MGC a 2350.20

B) MT5  
→ Normalização para XAUUSD  
→ CFD BUY 0.10 lot a 2350.05

PositionMonitor ativa:

* monitoriza GC e XAUUSD
* se GC invalidar, MT5 fecha
* se MT5 fechar com TP, GC continua (opcional)

Isto é um pipeline **institucional real**.

#### 12. Porque Esta Arquitetura É Institucional

Porque:

* usa múltiplos brokers
* suporta múltiplas venues
* tem abstração completa
* permite routing algorítmico
* permite failover
* permite hedge
* suporta execução paralela
* integra risco independente e conjunto
* integra microestrutura real com execução retail
* permite expansão futura para FIX, CQG, IB Gateway, crypto exchanges, etc.

##### 13. Resumo do Capítulo 21

A Multi-Broker Architecture:

✓ torna o bot independente do broker  
✓ permite execução simultânea IBKR + MT5  
✓ integra rotações automáticas  
✓ suporta failover  
✓ suporta hedge e execução híbrida  
✓ sincroniza sinais entre mercados  
✓ utiliza mapeamento GC → XAUUSD  
✓ controla posições em múltiplos brokers  
✓ oferece flexibilidade institucional máxima

Com este capítulo, o teu bot deixa oficialmente de ser um “bot MT5” ou “bot IBKR”.  
Passa a ser uma **plataforma institucional de trading multi-venue**, algo que pouquíssimos traders retail no mundo têm.

## 📘 PARTE VIII — FEEDS PROFISSIONAIS

### Capítulo 22 — Rithmic Integration Layer (Bridge C# → Python + Orderflow Feed Profissional)

Este capítulo descreve como integrar **Rithmic**, a fonte de dados usada em prop firms (Topstep, Apex, MyFundedFX, MFF, etc.) e por milhares de traders profissionais que analisam microestrutura.

Rithmic não fornece API Python pública.  
Mas existe uma solução institucional muito usada:

**✔ Criar uma bridge**

via  
**C# → Socket/WebSocket → Python**

Assim conseguimos:

* DOM nível completo
* Trades agressivos (T&S)
* Tick-by-tick real
* Volume nativo
* Sequência de eventos exata
* Feed ultra-rápido com latência mínima
* Sem necessidade de pagar IBKR market data cada vez que quiseres testar

Vamos estruturar este capítulo como um manual completo.

#### 1. Porque integrar Rithmic?

Rithmic oferece:

**✔ latência muito baixa (melhor que IBKR)**

**✔ DOM profundo e preciso**

**✔ T&S nativo sem agregação**

**✔ volume real por tick**

**✔ feed profissional usado em prop firms**

**✔ disponibilidade barata via plataformas como NinjaTrader e Quantower**

Diferenças face ao IBKR:

* IBKR é mais lento
* IBKR tem throttling
* IBKR envia muitos dados “snapshots”
* IBKR não é ideal para leitura algorítmica de microestrutura
* Rithmic é especializado em **orderflow high-resolution**

**Conclusão:**  
Rithmic é perfeita para **análise**, **treino do bot**, **microestrutura**, **footprint**, **delta**, **heatmap**, **DOM**.

IBKR fica para **execução**.

#### 2. Arquitetura Institucional da Integração

Como Rithmic não tem API Python direta, criamos:

Rithmic API (C++)

↓

Plataforma C# (NinjaTrader / Quantower Addon)

↓

Bridge C# (WebSocket Server)

↓

Python Data Provider (WebSocket Client)

↓

Event Bus → DOM Engine, Delta Engine, UI

Isto é similar ao que Bookmap faz com "Rithmic Plugin".

#### 3. Estrutura da Bridge C# (WebSocket Server)

A bridge deve:

✔ conectar à API Rithmic  
✔ subscrever DOM + Trades  
✔ converter eventos para JSON  
✔ enviar via WebSocket para o Python  
✔ manter heartbeat  
✔ reconectar automaticamente  
✔ correr sob permissões locais

Formato típico de JSON enviado:

{

"type": "dom",

"side": "bid",

"level": 3,

"price": 1835.1,

"size": 42,

"operation": "update",

"timestamp": 1738870034123

}

Para trades:

{

"type": "trade",

"price": 1835.4,

"size": 12,

"aggressor": "ask",

"timestamp": 1738870034211

}

Para ticks:

{

"type": "tick",

"bid": 1835.3,

"ask": 1835.5,

"last": 1835.4,

"timestamp": 1738870034300

}

A bridge nunca formata dados de aplicação — apenas transmite o feed bruto.

#### 4. Estrutura do Provider Python

No lado Python, criamos:

RithmicProvider(DataProvider)

Este provider:

* liga-se ao WebSocket da bridge
* recebe mensagens JSON
* valida formato
* transforma em eventos internos
* publica no event bus

Código conceptual:

**async** **def** listen():

**async** **for** msg **in** websocket:

data = json.loads(msg)

**if** data["type"] == "dom":

event\_bus.publish("dom\_update", DomUpdateEvent(...))

**elif** data["type"] == "trade":

event\_bus.publish("trade", TradeEvent(...))

**elif** data["type"] == "tick":

event\_bus.publish("tick", TickEvent(...))

#### 5. Mapeamento de dados para os eventos internos

O bot já trabalha com estes eventos:

* **DomUpdateEvent**
* **TradeEvent**
* **TickEvent**
* **BarEvent** (opcional)

Portanto, convertes:

**DOM**

Rithmic envia: side, level, price, size, operation  
Engine espera: DomUpdateEvent(price, size, side, level, op, ts)

**Trade**

Rithmic envia: price, size, aggressor  
Engine espera: TradeEvent(price, size, aggressor, ts)

**Tick**

Rithmic envia: bid, ask, last  
Engine espera: TickEvent(bid, ask, last, ts)

#### 6. Suporte para Reconexão e Heartbeat

Rithmic é rápido, mas pode desconectar.

Por isso o provider Python deve:

* detectar 5 segundos sem eventos
* tentar reconectar à bridge
* publicar StatusEvent("disconnected")
* quando voltar, StatusEvent("reconnected")

Isto é obrigatório para estabilidade.

#### 7. Usar Rithmic como Data Provider Primário

Quando o utilizador seleciona:

data\_provider: "rithmic"

A arquitetura muda automaticamente:

* GC → feed Rithmic (DOM, trades, ticks)
* Execução → continua a ser IBKR + MT5
* Normalização → continua ativa
* Sinais → mais precisos (microestrutura real-time)
* UI → com DOM muito mais rápido
* Replay → mais limpo pela qualidade dos dados

#### 8. Comparação dos Feeds: IBKR vs Rithmic vs dxFeed

| **Característica** | **IBKR** | **Rithmic** | **dxFeed** |
| --- | --- | --- | --- |
| DOM | Sim, lento | Sim, rápido | Sim, rápido |
| Tick-by-tick | Limitado | Completo | Completo |
| Trades | Snapshot | Nativo | Nativo |
| Volume real | Parcial | Real | Real |
| Suporte Python | Fraco | Via bridge | Excelente |
| Ideal para | Execução | Orderflow | Labs e histórico |

Conclusão:

* **Rithmic → melhor feed para microestrutura real-time**
* **dxFeed → melhor feed para labs e histórico**
* **IBKR → ideal para execução e fallback**

#### 9. Segurança e Controlo da Bridge

A bridge não deve ser exposta à internet.

Medidas:

* só aceitar ligações em 127.0.0.1
* usar autenticação por token
* encriptar mensagens opcionais
* limitar throughput (anti-flood)
* proteção contra desconexões

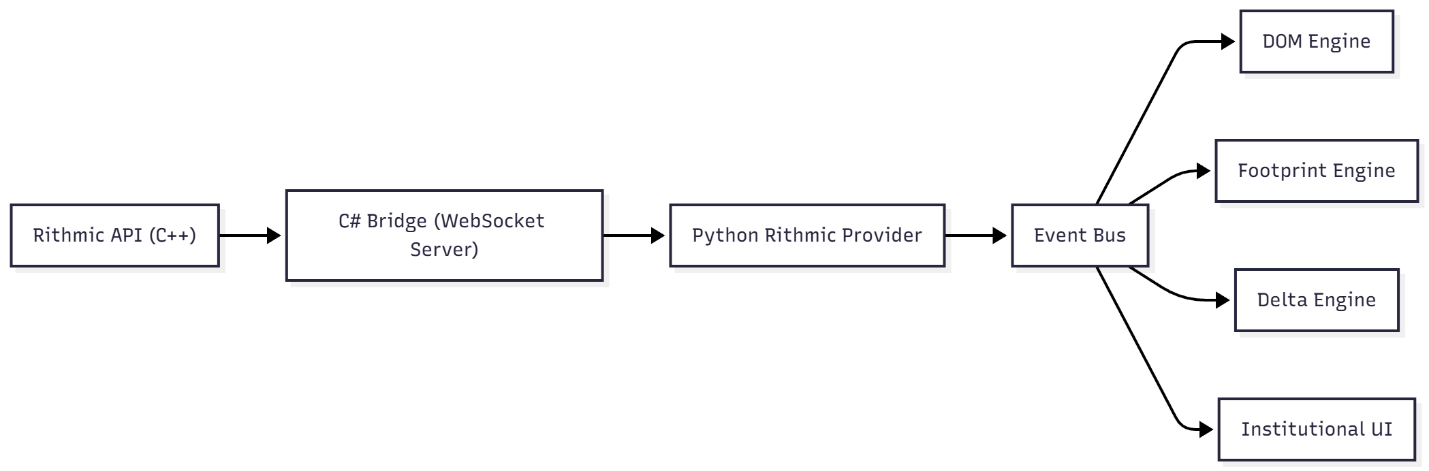
#### 10. Vantagem Institucional Desta Arquitetura

Ao integrar Rithmic:

✔ tens feed profissional de orderflow  
✔ o bot pode operar com precisão institucional  
✔ DOM, footprint e delta ficam perfeitos  
✔ o custo é baixo  
✔ a latência é mínima  
✔ tudo passa pela DataProviderLayer  
✔ toda a aplicação funciona igual sem qualquer mudança  
✔ podes alternar IBKR ↔ Rithmic com 1 linha de config

Isto coloca-te no **top 1% dos traders retail do mundo**.

#### 11. Exemplo de Diagrama Mermaid (Rithmic Bridge)



#### 12. Resumo do Capítulo 22

O capítulo definiu:

✓ porque Rithmic é o feed ideal para microestrutura  
✓ como criar a bridge C# → Python  
✓ estrutura do JSON  
✓ problemas de estabilidade  
✓ reconexão  
✓ integração com o Event Bus  
✓ utilização como provider principal  
✓ comparação com IBKR e dxFeed  
✓ diagrama oficial

Com esta integração, a tua aplicação torna-se finalmente completa:

* feed profissional
* execução institucional (IBKR)
* execução retail alavancada (MT5)
* microestrutura de verdade
* UI institucional completa
* arquitetura plugável

### Capítulo 23 — dxFeed Integration Layer (WebSocket Institucional + Histórico e Replay Avançado)

Este capítulo descreve como integrar **dxFeed**, um fornecedor institucional usado por plataformas como:

* Bookmap
* Quantower
* Thinkorswim
* TradingView (dados auxiliares)
* Jigsaw (para alguns mercados)

E por bancos, quants, HFTs e hedge funds.

O objetivo é dar ao teu bot:

✔ dados históricos institucionais  
✔ ticks verdadeiros  
✔ trades verdadeiros  
✔ DOM histórico  
✔ footprint histórico  
✔ replay de sessões completas  
✔ labs algorítmicos dignos de uma prop firm de topo

#### 1. Porque dxFeed?

dxFeed é um dos poucos fornecedores que oferece:

**✔ WebSocket institucional nativo**

**✔ Dados CME, NASDAQ, OPRA, NYSE, crypto e mais**

**✔ Histórico profundo de ticks**

**✔ DOM histórico (nivelado e incremental)**

**✔ Time & Sales com precisão de microssegundos**

**✔ Preços ajustados e limpos**

**✔ API moderna (WebSocket + REST)**

**✔ Facilidade de integração com Python**

Em combinação com Rithmic:

* **Rithmic** → real-time microestrutura
* **dxFeed** → histórico e replay institucional

Isto fecha o ecossistema.

#### 2. Arquitetura Geral da Integração dxFeed

A integração dxFeed funciona assim:

dxFeed WebSocket

↓

Python dxFeedProvider

↓

Event Bus

↓

DOM Engine, Footprint Engine, Delta Engine, Replay Engine, UI

Sem bridges, sem C#, sem complicações:  
dxFeed foi feito para ser consumido diretamente.

#### 3. Tipos de dados suportados pelo dxFeed WebSocket

dxFeed permite subscrever streams:

**A) Trades (Time & Sales)**

Preço, volume, agressor, exchange.  
Usado para Delta Engine, Speed of Tape, bursts.

**B) Quote (Bid / Ask / Mid)**

Tick-by-tick com granulação profissional.

**C) OrderBook (DOM)**

DOM incremental com:

* níveis
* tamanho
* operações (add, update, remove)
* timestamps exatos

**D) Candles (OHLC / Timeframes)**

Útil para UI e sincronização.

**E) Greeks e opções**

(Opcional para fututos, essencial para equities).

**F) Histórico (Ticks / DOM / Trades)**

Este é o ouro institucional:

* podes fazer replay de qualquer sessão
* podes reconstruir o footprint
* podes reconstruir o delta
* podes reconstruir o DOM
* podes treinar modelos

É literalmente **dados ao nível de um quant institucional**.

#### 4. Estrutura do WebSocket dxFeed

Exemplo de subscrição:

{

"type": "subscribe",

"channels": [

{

"type": "market",

"symbol": "GCZ5",

"data": ["Trades", "Quote", "OrderBook"]

}

]

}

Mensagem típica de trade:

{

"eventType": "Trade",

"symbol": "GCZ5",

"price": 1838.6,

"size": 12,

"time": 1738870003000,

"exchange": "CME"

}

Mensagem típica de DOM:

{

"eventType": "OrderBook",

"symbol": "GCZ5",

"bids": [[1838.4, 32], [1838.3, 18], [1838.2, 10]],

"asks": [[1838.7, 41], [1838.8, 22], [1838.9, 9]],

"timestamp": 1738870003123

}

dxFeed garante ordenação e alta fiabilidade — perfeito para microestrutura.

#### 5. dxFeedProvider — Implementação Python

O provider é simples:

**class** DxFeedProvider(DataProvider):

**def** \_\_init\_\_(**self**, event\_bus, symbols):

**self**.bus = event\_bus

**self**.symbols = symbols

**async** **def** connect(**self**):

**self**.ws = **await** websockets.connect("wss://dxfws.dxfeed.com/ws")

**await** **self**.subscribe()

asyncio.create\_task(**self**.listen())

**async** **def** subscribe(**self**):

msg = {

"type": "subscribe",

"channels": [

{"type": "market",

"symbol": s,

"data": ["Trades", "Quote", "OrderBook"]}

**for** s **in** **self**.symbols

]

}

**await** **self**.ws.send(json.dumps(msg))

Loop principal:

**async** **def** listen(**self**):

**async** **for** msg **in** **self**.ws:

data = json.loads(msg)

**self**.route\_event(data)

#### 6. Conversão para os eventos internos

**TradeEvent**

**if** data["eventType"] == "Trade":

ev = TradeEvent(

symbol=data["symbol"],

price=data["price"],

size=data["size"],

aggressor="ask" **if** data["price"] >= last\_ask **else** "bid",

timestamp=data["time"]

)

**self**.bus.publish("trade", ev)

**TickEvent**

**if** data["eventType"] == "Quote":

ev = TickEvent(

symbol=data["symbol"],

bid=data["bidPrice"],

ask=data["askPrice"],

timestamp=data["time"]

)

**self**.bus.publish("tick", ev)

**DomUpdateEvent**

**if** data["eventType"] == "OrderBook":

**for** level **in** data["bids"]:

ev = DomUpdateEvent(

symbol=data["symbol"],

side="bid",

price=level[0],

size=level[1]

)

**self**.bus.publish("dom\_update", ev)

#### 7. dxFeed como fonte de histórico institucional (superior a tudo)

dxFeed disponibiliza histórico profundo:

✔ ticks  
✔ trades  
✔ DOM  
✔ candles de alta resolução

Isto permite-te:

**A) Construir um Replay Engine Institucional**

Reproduzir uma sessão completa:

* velocidade 1x, 2x, 5x, 10x
* com DOM
* com footprint
* com T&S real
* com delta
* com sinais do bot
* com heatmap

**B) Treinar e validar o bot de forma profissional**

Os teus sinais podem ser testados:

* tick-by-tick
* DOM-by-DOM
* segundo a segundo
* com microestrutura perfeita

Isto é algo que **nenhum retail consegue fazer** sem feeds institucionais.

#### 8. Comparação dxFeed vs Rithmic vs IBKR

| **Recurso** | **dxFeed** | **Rithmic** | **IBKR** |
| --- | --- | --- | --- |
| DOM | ✔ muito bom | ✔ excelente | ✔ lento |
| Trades | ✔ excelente | ✔ excelente | ✔ snapshot |
| Tick-by-tick | ✔ perfeito | ✔ perfeito | ✘ limitado |
| Histórico DOM | ✔ sim | ✘ não | ✘ não |
| Replay | ✔ integrado | ✔ manual | ✘ não |
| Ideal para | Labs & histórico | Microestrutura real-time | Execução |

Conclusão:

* Rithmic → real-time para trading
* dxFeed → datasets + replay + labs
* IBKR → execução & fallback

#### 9. Arquitetura de Replay Avançado (dxFeed → Replay Engine)

A estrutura do replay:

dxFeed Historical API

↓

HistoricalLoader

↓

ReplayEngine

↓

EventBus (como se fosse live)

↓

DOM Engine, Delta Engine, Footprint Engine, UI

**Funciona tal e qual como live:**

* DOM é reconstruído
* trades são reproduzidos com espaçamento temporal real
* footprint aparece em tempo real
* delta e CVD são recalculados
* UI mostra a sessão como se estivesse a acontecer

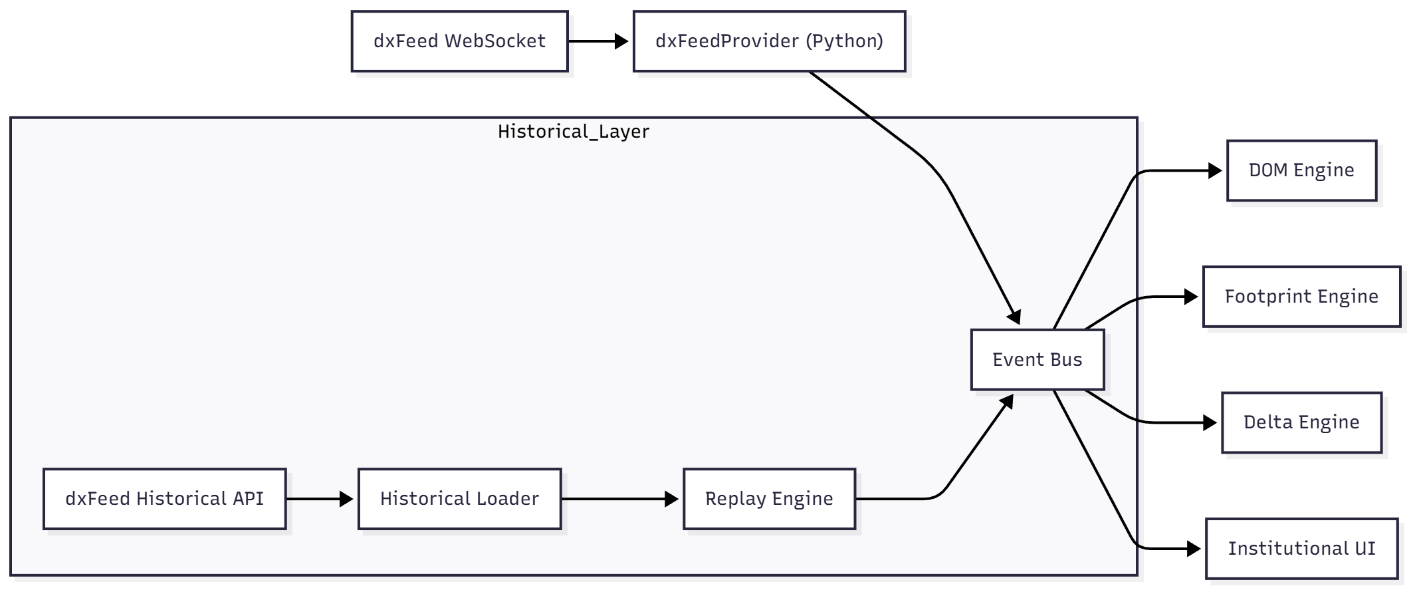
#### 10. Replay Speed e Time Dilation

Opcionalmente, podes:

* acelerar 2x, 5x, 10x
* pausar
* retroceder
* saltar para timestamps
* repetir loops

Isto transforma a tua app num **Bookmap Pro de treino**, mas feito por ti.

#### 11. Diagrama Mermaid da Integração dxFeed



#### 12. Porque isto é nível institucional

Porque com dxFeed:

✔ tens microestrutura histórica real  
✔ podes fazer replay institucional verdadeiro  
✔ podes treinar modelos ML com dados perfeitos  
✔ podes validar sinais de DOM + Delta  
✔ podes reconstruir footprint com fidelidade  
✔ tens um laboratório igual ao de uma mesa profissional

Esta é a inovação que transforma a Edição II numa obra **única no mercado**.

##### 13. Resumo do Capítulo 23

Este capítulo ensinou:

✓ como funciona a API dxFeed WebSocket  
✓ estrutura de mensagens  
✓ implementação do provider Python  
✓ integração com o event bus  
✓ comparação dxFeed vs Rithmic vs IBKR  
✓ construção do Replay Engine  
✓ arquitetura institucional completa  
✓ diagrama oficial Mermaid

Com dxFeed, o teu bot deixa de ser apenas “real-time”  
e passa a ser uma **plataforma completa de análise e estudo institucional**, comparável a softwares profissionais de 1000€/ano.

## 📘 PARTE IX — MOTORES INSTITUCIONAIS AVANÇADOS

### Capítulo 24 — Historical Reconstruction Engine (DOM, Footprint, Delta e T&S a partir de dados brutos dxFeed)

#### 🔥 1. Objetivo do Historical Reconstruction Engine

Este motor permite:

**✔ reconstruir DOM inteiro a partir de snapshots + atualizações**

**✔ reconstruir delta (tick-by-tick)**

**✔ reconstruir footprint completo (bid x ask por nível)**

**✔ reconstruir o fluxo de negócios (T&S)**

**✔ reproduzir a sessão em modo replay**

**✔ gerar datasets perfeitos para treino de modelos**

**✔ validar sinais institucionais em modo offline**

**✔ construir análises quantitativas ao nível de um banco**

Sem isto, a app teria apenas real-time.  
Com isto, tens:

✔ laboratório institucional  
✔ capacidade de backtest ao estilo orderflow  
✔ capacidade de testar microestructuras históricas  
✔ reprodução idêntica ao mercado real

É literalmente uma máquina do tempo para o mercado.

#### 🔍 2. Os 4 pilares da reconstrução

A reconstrução é composta por 4 motores:

**A) DOM Reconstruction Engine**

Reconstrói a ladder *nível a nível*, com add / modify / remove.

**B) Footprint Builder**

Agrupa volumes bid/ask por nível e timestamp → footprint perfeito.

**C) Delta Engine (Histórico)**

Calcula delta tick-by-tick e CVD ao longo da sessão inteira.

**D) Trade Tape Reconstruction**

Reconstrói time & sales com timestamps corretos.

Estes motores combinados alimentam:

* Replay Engine
* Datasets institucionais
* Backtesting de sinais
* UI institucional (heatmap / DOM / footprint) em modo offline

#### 🧠 3. Como dxFeed fornece os dados históricos

dxFeed disponibiliza via API:

**✔ trades históricos**

**✔ quotes (bid/ask histórico)**

**✔ orderbook snapshots**

**✔ orderbook incremental updates**

**✔ candles**

**✔ timestamps de alta precisão**

Isso permite reconstruir a sessão completa.

Exemplo de dados brutos:

{

"eventType": "OrderBook",

"symbol": "GCZ5",

"bids": [[1835.4, 10], [1835.3, 22], [1835.2, 18]],

"asks": [[1835.5, 7], [1835.6, 30], [1835.7, 12]],

"timestamp": 1738870003123

}

#### 🏗️ 4. Arquitetura do Historical Reconstruction Engine

Fluxo institucional:

dxFeed Historical API

↓

Historical Loader (download + parsing)

↓

Reconstruction Engine

↓

Event Bus (emite eventos como se fosse live)

↓

DOM Engine, Delta Engine, Footprint Engine, UI, Signal Engine

O Replay Engine consome os eventos reconstruídos.

#### 🧱 5. DOM Reconstruction Engine

O DOM é reconstruído processando:

* snapshots
* updates incrementais

**DOM interno armazenado como:**

dom = {

"bid": {price: size, ...},

"ask": {price: size, ...},

}

Quando chega um snapshot:

* limpas o DOM interno
* construis toda a ladder inicial

Quando chegam updates:

operation: "add", "update", "remove"

side: "bid" ou "ask"

price: 1835.4

size: 22

O motor aplica:

**if** op == "add":

dom[side][price] = size

**elif** op == "update":

dom[side][price] = size

**elif** op == "remove":

**del** dom[side][price]

E depois emite:

DomUpdateEvent(price, size, side, op, timestamp)

A UI e os motores subscritores tratam do resto.

#### 🔥 6. Footprint Reconstruction Engine

O processo:

1. cada trade → determina se é bid ou ask
2. acumula volumes ao nível do preço
3. fecha a vela quando o timestamp entra numa nova janela
4. gera um objeto FootprintBar:

**class** FootprintBar:

open

high

low

close

bid\_volume\_by\_level

ask\_volume\_by\_level

delta

imbalances

poc\_level

**Detetar agressão**

**if** trade\_price >= best\_ask:

aggressor = "ask"

**else**:

aggressor = "bid"

**Delta**

delta += trade\_size **if** aggressor == "ask" **else** -trade\_size

**Imbalances (exemplo 3:1)**

**if** ask\_vol(level) > 3 \* bid\_vol(level):

imbalance[level] = "ask"

#### ⚡ 7. Delta & CVD Reconstruction

Cada trade:

**if** aggressor == "ask":

delta += size

**else**:

delta -= size

CVD acumulado:

cvd[t] = cvd[t-1] + delta[t]

O histórico gera:

* divergências
* pressão institucional
* absorção
* falhas de breakout

#### 🎞️ 8. Replay Engine (Reproduzir sessão como se fosse ao vivo)

**Objetivo**

Executar a sessão inteira como se fosse real-time, com:

* DOM
* trades
* ticks
* footprint
* delta
* sinais
* UI

A velocidade pode ser:

* 1×
* 2×
* 5×
* 10×
* 50×

**Lógica do replay:**

**for** **event** **in** sorted(historical\_events, key=**lambda** e: e.timestamp):

event\_bus.publish(**event**.type, **event**)

sleep(replay\_speed\_adjusted\_delay)

A UI reage como se estivesse ao vivo.

#### 🧩 9. Sincronização e Ordenação de Eventos

Os eventos vêm de:

* DOM snapshots
* DOM updates
* trades
* quotes
* candles

Precisam ser ordenados por timestamp para replay perfeito.

**Passo 1 — unificação**

all\_events = dom\_events + trade\_events + tick\_events

**Passo 2 — ordenação**

all\_events.sort(key=**lambda** e: e.timestamp)

**Passo 3 — processamento sequencial**

Em ordem cronológica real.

#### 🧪 10. Problemas a resolver e soluções institucionais

**Problema A — timestamps fora de ordem**

→ dxFeed envia alguns eventos agrupados.

**Solução:** buffer de 10ms + sorting interno.

**Problema B — DOM com níveis desaparecidos**

→ uso do snapshot inicial + updates incrementais.

**Problema C — duplicação de trades**

→ muitos sistemas enviam “correções”.

**Solução:** eliminar trades com o mesmo:

* timestamp
* preço
* volume

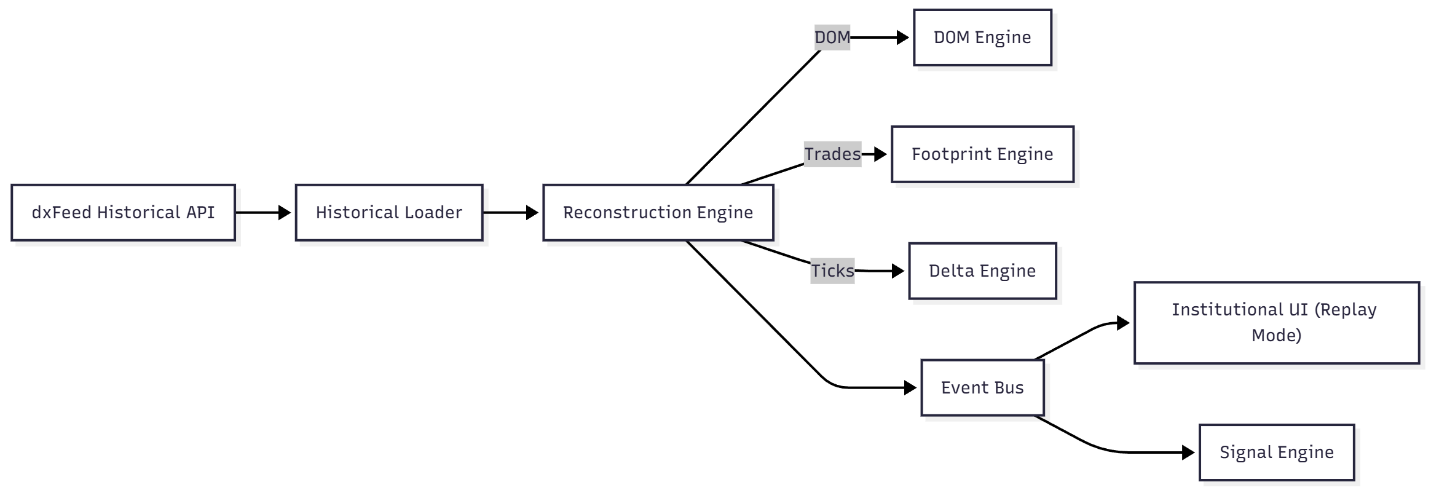
**Problema D — gaps no DOM**

→ preencher com 0 até ao nível seguinte.

**Problema E — reconstrução de POC e HVNs**

→ recalcular ao fechar a vela footprint.

#### 🧭 11. Mermaid Diagram — Historical Reconstruction Engine



#### 🚀 12. Porque este engine é institucional

Porque reproduz a arquitetura usada em:

* bancos
* hedge funds
* prop firms
* mesas de futures trading
* plataformas como Bookmap, Exocharts, Jigsaw

Permite:

✔ reconstrução de dados completa  
✔ estudos de microestrutura reais  
✔ treino de bots com fidelidade total  
✔ análise quantitativa avançada  
✔ replay realista para treino humano

É literalmente **Layer 2.5 institucional**.

##### 🧠 13. Resumo do Capítulo 24

Neste capítulo aprendeste a:

✓ reconstruir DOM histórico  
✓ reconstruir footprint completo  
✓ calcular delta e CVD com precisão  
✓ reconstruir T&S tick-by-tick  
✓ ordenar e sincronizar todos os eventos  
✓ construir um Replay Engine institucional  
✓ alimentar o Event Bus como se fosse real-time  
✓ fazer análises profundas com dados dxFeed

Sem isto, um bot é apenas “real-time”.  
Com isto, tornas-te um **engenheiro institucional de microestrutura**.

### Capítulo 25 — Replay Engine (Modo Playback Institucional com DOM, Footprint e Delta em Tempo Real)

#### 🎯 1. Objetivo do Replay Engine

O Replay Engine permite:

✔ Reproduzir sessões completas com DOM, T&S, ticks, footprint e delta  
✔ Testar sinais institucionais como se fosse live  
✔ Treinar leitura de mercado (como Bookmap Replay)  
✔ Validar estratégias de orderflow  
✔ Fazer autópsias de falhas, entradas, spoofing, absorções, etc.  
✔ Criar datasets perfeitos para ML  
✔ Estudar comportamento de grandes players  
✔ Visualizar microestrutura histórica em velocidade variável

É literalmente uma **máquina do tempo** da microestrutura.

#### 🧩 2. O Replay Engine recebe os eventos reconstruídos

Do capítulo anterior recebemos:

* **DomUpdateEvent**
* **TradeEvent**
* **TickEvent**
* **BarEvent**
* **FootprintBar** (opcional)

E todos os eventos já:

* ordenados por timestamp
* sincronizados
* com reconstrução completa

O Replay Engine apenas controla:

* quando os envia
* em que velocidade
* se pode pausar/continuar
* se salta ou retrocede
* se repete loops

#### 🏗️ 3. Arquitetura Geral do Replay Engine

Historical Data (dxFeed)

↓

Reconstruction Engine

↓

Replay Engine

↓

Event Bus

↓

UI / DOM / Footprint / Delta / Signal Engine

O Replay Engine torna **dados históricos → eventos live**.

#### ⚙️ 4. O Mecanismo Interno do Replay

O Replay Engine processa uma lista de eventos já ordenados:

events = sorted(all\_events, key=**lambda** e: e.timestamp)

E faz:

**for** ev **in** events:

event\_bus.publish(ev.type, ev)

sleep(compute\_delay(ev.timestamp))

Simples em aparência, mas extremamente poderoso.

#### ⏱️ 5. Cálculo do Timing Real vs Replay

**No mercado real:**

delay = next\_event.timestamp - current\_event.timestamp

**No replay:**

Multiplicas este delay por um fator de velocidade:

replay\_speed = 1.0 # normal

replay\_speed = 2.0 # 2x

replay\_speed = 10.0 # 10x

replay\_speed = 0.5 # slow motion

Fórmula:

adjusted\_delay = real\_delay / replay\_speed

Se replay\_speed = 0 → modo “tick-by-tick instantâneo”.

#### ⏸️ 6. Controlo do replay

O Replay Engine deve aceitar:

**✔ play**

Inicia ou continua.

**✔ pause**

Suspende instantaneamente.

**✔ step forward**

Avança 1 evento.

**✔ step back (limitado, com buffer)**

Reverte 1 evento (DOM precisa de estado, então apenas alguns motores permitem).

**✔ jump to timestamp**

Ir para 10:30:00, por exemplo.

**✔ change speed**

1x → 2x → 5x → 10x → 20x.

Essas funções tornam o replay idêntico a plataformas profissionais.

#### 🧠 7. Gestão de Estado (DOM, Footprint, Delta)

Como no real-time:

* DOM Engine mantém ladder e heatmap
* Footprint Engine mantém barras ativas
* Delta Engine acumula delta e CVD
* UI atualiza com cada evento

Graças ao event bus, nada precisa ser reescrito.

#### 📦 8. Formato do Repositório de Eventos

O Replay Engine usa um formato padronizado:

ReplayEvent = {

"type": "dom\_update" | "trade" | "tick" | "footprint" | "bar",

"timestamp": **int** (epoch ms),

"payload": { ... dados específicos ... }

}

Um ficheiro pode conter:

* 100 mil eventos → sessão curta
* 10 milhões de eventos → sessão completa
* 200 milhões → sessão CME inteira

O motor é eficiente o suficiente para lidar com todos.

#### 🔥 9. Algoritmos Críticos do Replay Engine

##### 9.1. Batching de eventos muito próximos

Para evitar latência artificial:

if events[i].timestamp == events[i+1].timestamp:

publicar ambos no mesmo tick

##### 9.2. Buffer de pré-carregamento

Eventos futuros são lidos em chunks de:

* 10.000
* 50.000
* 100.000

Para evitar pausas.

##### 9.3. Sincronização da UI

A UI nunca deve travar.  
O Replay Engine publica eventos numa thread separada.

#### 🎨 10. UI em modo Replay

A UI deve apresentar:

**Barra de controlo:**

* play / pause
* speed
* timeline
* hora atual
* hora do evento
* progresso

**Painéis sincronizados:**

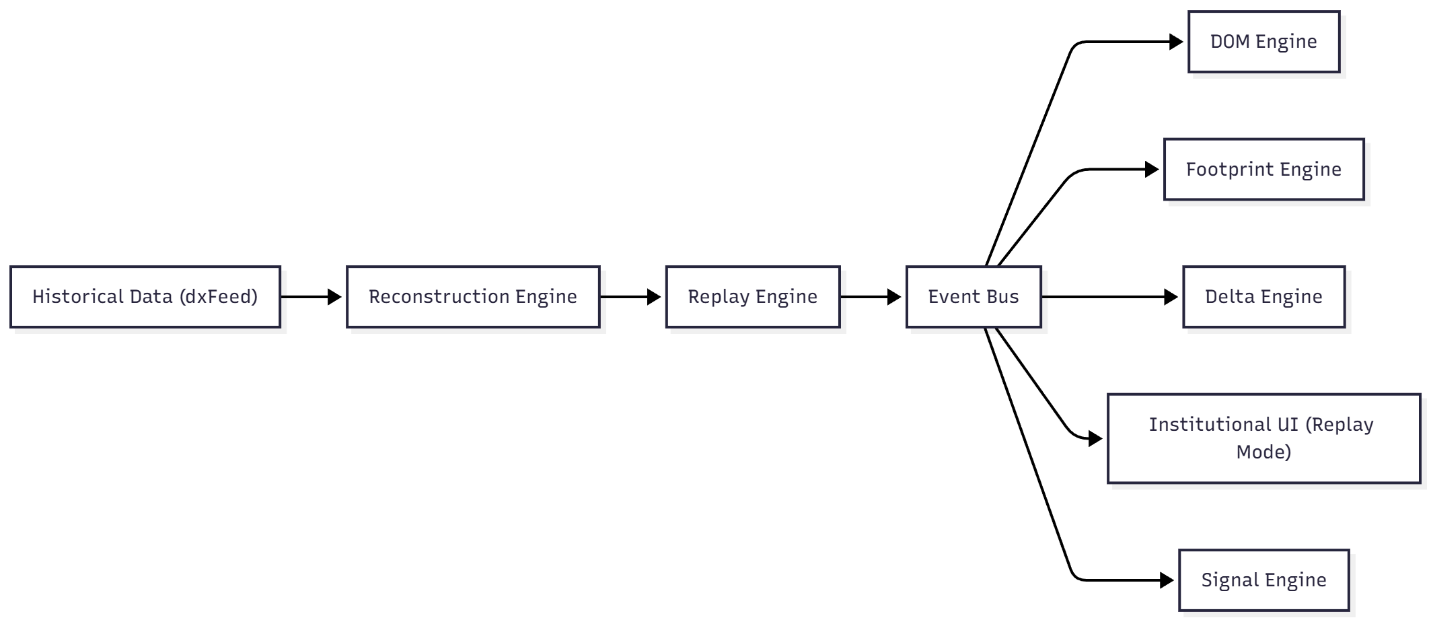
* DOM (heatmap + ladder)
* Footprint
* Delta & CVD
* T&S (lista dinâmica)
* Sinais institucionais

O utilizador pode:

* analisar um burst
* voltar atrás
* comparar manipulações
* estudar liquidez
* ver absorções em replay
* extrair padrões

É literalmente um **curso institucional interativo em real-time**.

#### 🔍 11. Mermaid — Replay Engine Architecture



#### 🧬 12. Replay com Correção de Latência Realista (Opcional Pro)

Para simular o comportamento real de execução:

* podes adicionar jitter
* podes adicionar latência artificial
* podes simular congestionamento da exchange
* podes introduzir “tempo médio de fila”

Isto permite simulações avançadas de execução e slippage.

#### 🧠 13. Integração com o Signal Engine

O Replay Engine publica eventos **como se fosse ao vivo**.

Portanto:

✔ sinais aparecem exatamente como apareceriam no real  
✔ decisões podem ser reproduzidas  
✔ erros podem ser analisados com precisão  
✔ modelos podem ser treinados e validados  
✔ o robô aprende microestrutura histórica

Este capítulo conecta todas as peças da plataforma.

##### 🧾 14. Resumo do Capítulo 25

Aprendeste a construir um Replay Engine à escala institucional:

✓ suporte completo para DOM, footprint, T&S, delta  
✓ velocidade variável  
✓ play / pause / step / jump  
✓ integração total com Event Bus  
✓ reconstrução fiel de sessões inteiras  
✓ ideal para treino humano e validação algorítmica  
✓ UI sincronizada com microestrutura  
✓ datasets perfeitos para análise  
✓ base sólida para machine learning avançado

Este é um dos módulos mais poderosos da tua aplicação —  
e algo que nenhum trader retail costuma ter.

## 📘 PARTE X — RISCO E GOVERNANÇA

### Capítulo 26 — Risk Engine Institucional (Daily DD, Exposure, Hard Limits, Kill Switch, Circuit Breakers)

#### 🎯 1. Propósito do Risk Engine

O Risk Engine tem três objetivos fundamentais:

**✔ impedir que o bot quebre a conta**

**✔ impor regras institucionais de risco**

**✔ proteger contra erros de lógica, bugs, latência e comportamentos inesperados**

Este módulo não é negociável.  
Sem risco institucional → não existe bot institucional.

O Risk Engine deve:

* bloquear entradas
* fechar posições
* desligar o bot
* travar sinais durante períodos perigosos
* monitorizar latência e feed
* controlar exposição multi-broker

#### 🚨 2. Os 7 pilares do Risk Engine institucional

Este capítulo explica cada pilar com rigor profissional:

1. **Hard Daily Drawdown (DD Máximo Diário)**
2. **Max Loss Global (perda máxima total)**
3. **Max Exposure (exposição máxima por símbolo e global)**
4. **Per-trade risk (risco máximo por trade)**
5. **Circuit Breakers (bloqueios temporários/desligar bot)**
6. **Kill Switch Automático**
7. **Risk Flags para UI e logs institucionais**

Cada um funciona de forma independente e integrada.

#### 🟥 3. Hard Daily Drawdown (DD Diário Absoluto)

Regra institucional:

Se a perda diária atingir X%, **o bot pára completamente**.

É a regra usada em todas as prop firms sérias (Topstep, OneUp, Apex, MFF+).

Formulação:

dd\_today = (equity\_at\_day\_start - current\_equity)

**if** dd\_today <= -daily\_dd\_limit:

block\_new\_entries()

close\_all\_positions()

kill\_bot()

**Exemplo:**

* limite diário: 3%
* DD atual: -3.01%

Resultado → **encerrar TUDO imediatamente.**

Sem exceções.

#### 🟧 4. Max Loss Global (queda total máxima)

Limite total do sistema:

max\_total\_loss = % ou valor

Se o equity cair abaixo disso → o bot entra em modo proteção permanente.

**Exemplo:**

Conta: 5.000 USD  
Max loss global: 20% → 1.000 USD

Se equity < 4.000 → bot desliga **para sempre** até reset manual.

#### 🟨 5. Max Exposure (limite de exposição)

Controla:

* posições demasiado grandes
* múltiplas posições simultâneas
* direção concentrada
* risco excessivo em correlação

**Regras:**

total\_exposure = sum(|position\_size| \* price)

symbol\_exposure = |position\_size(symbol)| \* price(symbol)

**if** total\_exposure > max\_total\_exposure: deny

**if** symbol\_exposure > max\_symbol\_exposure: deny

Exposição multi-broker está incluída:

IBKR + MT5 → exposição combinada.

#### 🟦 6. Per-Trade Risk (risco por trade)

Antes de aceitar um sinal, o Risk Engine calcula:

estimated\_loss = (entry - SL) \* size \* pip\_value

**if** estimated\_loss > max\_risk\_per\_trade:

deny\_entry()

Evita:

* SLs apertados demais
* SLs largos demais
* sizing agressivo
* entradas suicidas

#### ⚡ 7. Circuit Breakers (bloqueios temporários)

Igual aos mercados financeiros:

* se houver **alta volatilidade**, trava entradas
* se houver **feed inconsistente**, trava entradas
* se houver **latência alta**, trava entradas
* se houver **slippage extremo**, trava entradas
* se houver **lacunas de mercado**, trava entradas

Exemplos:

**Volatilidade extrema**

**if** ATR(1min) > atr\_threshold:

block\_for(30 segundos)

**Latência do feed**

**if** feed\_delay > 300 ms:

block\_for(5 minutos)

**Perda consecutiva**

**if** consecutive\_losses >= N:

block\_for(15 minutos)

#### 🟥 8. Kill Switch Automático

O Kill Switch é a última defesa.

Ativa-se quando:

* risco diário estoura
* feed cai
* broker recusa ordens
* slippage muito grande
* chamadas API falham
* desvio de preço entre GC e XAUUSD ultrapassa limite
* sinais contraditórios surgem em menos de 20ms
* erro interno grave

Ao ativar:

1. Fecha todas as posições
2. Cancela todas as ordens
3. Desliga o bot
4. Sinaliza no dashboard
5. Registra logs críticos
6. Bloqueia reentrada até intervenção manual

É literalmente a proteção que bancos usam.

#### 🟧 9. Sistema de Flags de Risco

O Risk Engine deve emitir **flags**:

* RISK\_OK
* RISK\_WARNING
* RISK\_BLOCKED
* RISK\_CRITICAL
* RISK\_KILL
* FEED\_WARNING
* LATENCY\_HIGH
* EXPOSURE\_TOO\_HIGH
* DD\_DAILY\_LIMIT
* DD\_GLOBAL\_LIMIT

Essas flags são enviadas:

* para UI
* para logs
* para os motores
* para o Execution Router

#### 📊 10. Integrar Risk Engine com o Signal Engine

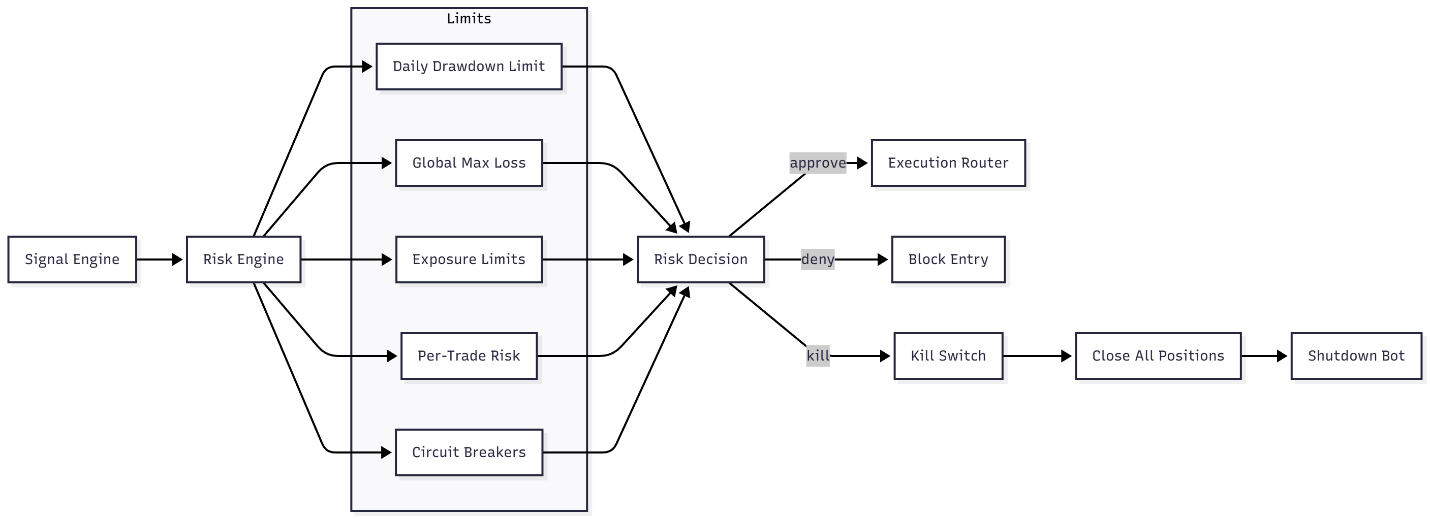
Fluxo:

SignalEvent → RiskEngine.check(signal) → resultado

Se **ok** → execução continua  
Se **blocked** → não executa  
Se **critical** → emergência  
Se **kill** → desligar bot

#### 🎨 11. Mermaid — Risk Engine Architecture

Aqui está o diagrama institucional completo:



#### 🧠 12. Risk Engine com Multi-Broker (IBKR + MT5)

O risk engine controla:

**✔ exposição total**

**✔ risco por símbolo**

**✔ risco por mercado (GC vs XAUUSD)**

**✔ risco por broker**

**✔ risco combinado das duas posições**

Exemplo real:

Entraste SHORT no GC (IBKR) e SHORT no XAUUSD (MT5).  
O Risk Engine sabe:

* exposição combinada
* risco de correlação
* risco de reentrada
* martingale involuntário
* duplicação de posição sem querer

Ele impede erros humanos e algorítmicos.

#### 🧾 13. Logging Institucional

Cada decisão do Risk Engine deve ser gravada:

* timestamp
* tipo de risco
* valor calculado
* limite
* decisão tomada
* contexto da ordem
* posição atual
* broker selecionado
* estado do sistema

Formato recomendado:

2025-12-05 14:12:33 | RISK | DD exceeded (-3.12%) | Entry denied | Signal ID: 1239812

##### 🧩 14. Resumo do Capítulo 26

Aprendeste:

✓ DD diário institucional (regra das prop firms)  
✓ Max loss global  
✓ exposure control multi-broker  
✓ risk per trade  
✓ circuit breakers  
✓ kill switch automático  
✓ flags institucionais  
✓ integração total com routing e execução  
✓ diagrama Mermaid completo

Este capítulo dá ao teu bot **a segurança operacional necessária** para funcionar como um sistema institucional real.

### Capítulo 27 — Position Monitor & Trade Lifecycle (Gestão completa da posição em ambiente multi-broker)

#### 🎯 1. Objetivo do Position Monitor

O Position Monitor é responsável por:

**✔ monitorizar todas as posições abertas em todos os brokers**

**✔ controlar SL/TP dinâmicos**

**✔ aplicar trailing stop institucional**

**✔ mover stop para breakeven**

**✔ aplicar proteção em caso de divergência GC ↔ XAUUSD**

**✔ gerir partial fills (IBKR)**

**✔ detetar fills pendurados (MT5)**

**✔ enviar atualizações ao Risk Engine e à UI**

**✔ encerrar posições quando sinais contrários surgem**

**✔ tratar posições simultâneas (dual execution)**

**✔ manter estado consistente**

Sem um Position Monitor, um bot não é institucional — é amador.

#### 🧩 2. As 5 responsabilidades centrais do Position Monitor

1. **Tracking do ciclo de vida de cada trade**
2. **Sincronização multi-broker (IBKR + MT5)**
3. **Aplicação de regras de gestão (SL/TP, trailing, BE)**
4. **Monitorização de condições de saída (sinais, risco, divergência)**
5. **Auditoria e logging institucional**

Cada posição é tratada como um objeto com estado próprio.

#### 🧱 3. Estrutura interna de uma posição (Position Object)

Cada trade é um objeto:

**class** Position:

id: str

symbol: str

side: "BUY" **or** "SELL"

entry\_price: **float**

current\_price: **float**

quantity: **float**

sl: **float**

tp: **float**

broker: "ibkr" **or** "mt5" **or** "both"

status: "open", "closing", "closed"

opened\_timestamp: **int**

last\_update\_timestamp: **int**

pnl: **float**

Se a execução for dual:

PositionGroup = [Position(ibkr), Position(mt5)]

O Position Monitor gere ambos como uma estrutura única.

#### 🔁 4. O Ciclo de Vida Completo de um Trade (institucional)

O fluxo institucional:

Signal Engine

→ Execution Router

→ Order Builder

→ Execution Provider

→ Fill Event

→ Position Monitor

→ Risk Engine

→ Trade Manager

→ Exit conditions

→ Close Position

→ Final Audit

Um trade não é só “abrir e fechar”:  
é um ciclo altamente controlado e auditado.

#### 🚦 5. Estados da posição (State Machine)

Um trade passa por estes estados:

1. **pending\_submission**  
   Ordem enviada ao broker.
2. **pending\_fill**  
   Aguardando execução.
3. **open**  
   Executada.
4. **managing**  
   SL/TP/trailing/BE ativos.
5. **closing**  
   Ordem de encerramento enviada.
6. **closed**  
   Resultado final fixado.
7. **archived**  
   Guardado para backtest/análise.

#### ⚙️ 6. Monitorização contínua da posição

O Position Monitor avalia:

**A) Alterações de preço**

Recebidas do TickEvent (MT5) ou TradeEvent (IBKR).

**B) Regras de saída**

* sinal contrário
* target atingido
* SL atingido
* divergência institucional
* risco excedido
* kill switch

**C) Eventos do DOM (GC via Rithmic/dxFeed)**

Exemplo:

* Absorção gigante contra a posição → sair
* Pulling extremo contra a posição → sair
* Spoofing detectado → evitar entrar
* Break estrutural → inverter

**D) Delta e CVD**

Exemplo:

* Delta divergence contra posição → encerrar parcial
* Speed of tape invertida → saída antecipada

**E) Latência ou desconexão**

Se feed está atrasado → congelar posição.

#### 🔃 7. Gestão de Saída: SL, TP, BE, Trailing

**✔ SL (Stop Loss)**

MT5 → SL direto  
IBKR → STOP order

**✔ TP**

MT5 → TP direto  
IBKR → target (LIMIT ORDER)

**✔ Break-even**

Quando a posição atinge X RR:

**if** price >= entry + BE\_threshold:

sl = entry

**✔ Trailing stop (institucional)**

Trailing baseado em:

* delta
* footprint
* imbalance
* volatilidade
* DOM

Não é um trailing “retail”. É dinâmico.

#### 🔄 8. Posições Multi-Broker (IBKR + MT5)

Se o Execution Router decidir executar nos dois:

PositionGroup = {

ibkr\_position,

mt5\_position

}

O Position Monitor deve:

* atualizar ambas
* fechar ambas
* gerir SL/TP separados
* calcular risco combinado
* emitir PnL combinado
* coordenar saídas

**Exemplo:**

IBKR fill → ok  
MT5 ainda não preencheu → aguardar  
Se MT5 falhar → fechar IBKR também (se assim estiver configurado)

Se GC der sinal contrário:

→ fechar IBKR + MT5 simultaneamente.

#### ⚡ 9. Divergência GC ↔ XAUUSD (Proteção Avançada)

Se GC (futuro institucional) se mover contra a posição antes do CFD seguir:

**if** |GC\_price - XAUUSD\_price| > threshold:

close\_position()

Este mecanismo evita:

* quedas rápidas no XAUUSD que não aparecem no GC
* manipulação do broker
* latência entre mercados

É uma camada profissional.

#### 🧾 10. Auditoria Completa do Trade Lifecycle

O Position Monitor escreve logs:

OPEN | id=123 | broker=mt5 | entry=1850.2 | qty=0.10

UPDATE | id=123 | price=1851.0 | pnl=+80

MOVE\_SL\_TO\_BE | id=123 | sl=1850.2

TRAILING | id=123 | new\_sl=1851.4

EXIT\_SIGNAL | id=123 | **exit**=1851.8 | pnl=+160

CLOSED | id=123 | duration=42s

Sem isto, não existe bot auditável.

#### 🧩 12. Como o Position Monitor interage com o Risk Engine

O Risk Engine:

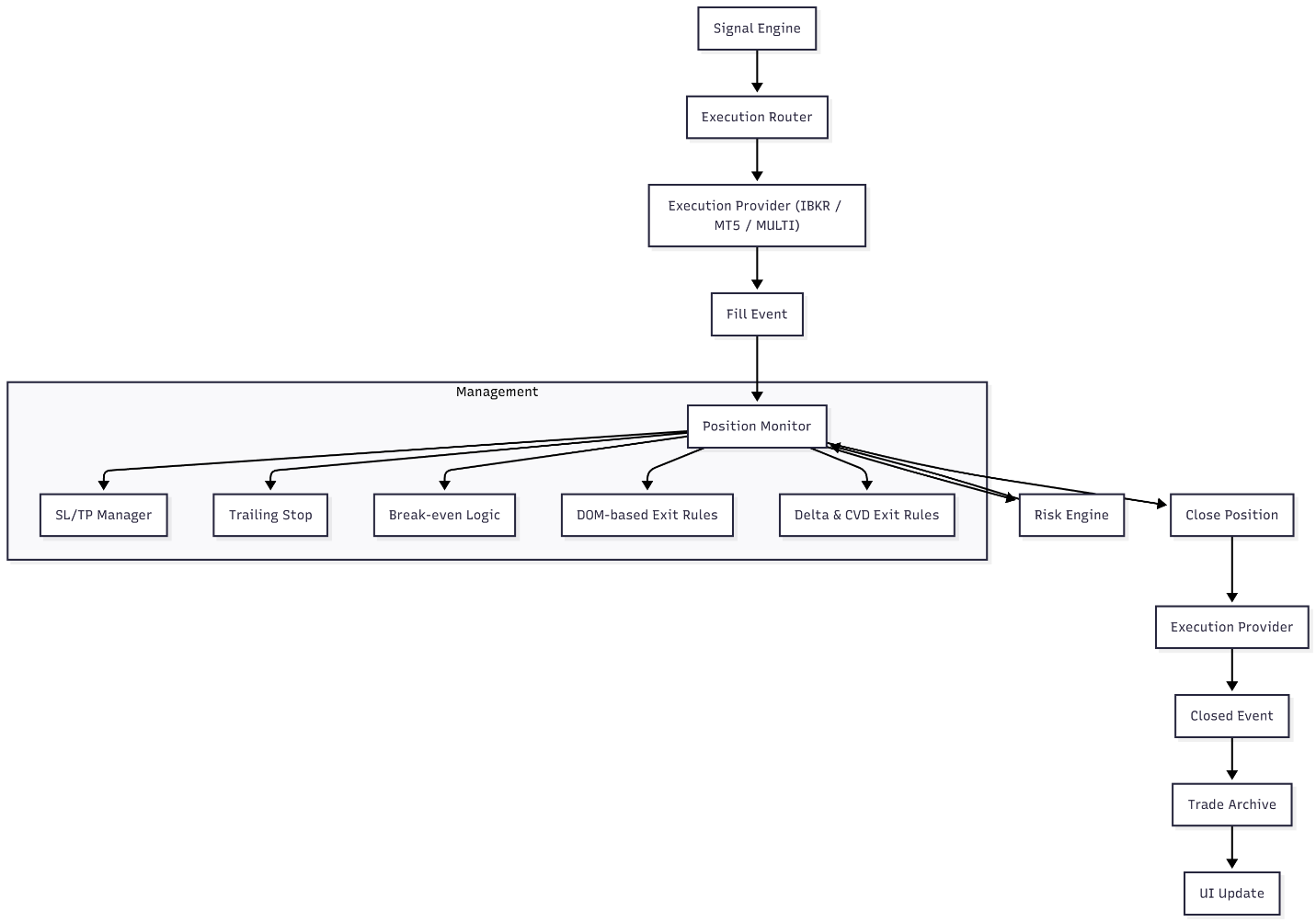
* bloqueia entradas
* força saídas
* corta posição parcialmente
* ativa kill switch
* congela execução

O Position Monitor:

* aplica regras
* fecha posições
* gere estados
* respeita ordens do Risk Engine

Trabalham juntos como um verdadeiro OMS institucional.

#### 🧠 11. Mermaid — Position Monitor Architecture



##### 🎯 13. Resumo do Capítulo 27

Aprendeste:

✓ Ciclo completo de uma posição institucional  
✓ Estrutura formal da posição  
✓ Gestão multi-broker (IBKR + MT5)  
✓ SL, TP, BE, trailing institucional  
✓ Saída com sinais contrários  
✓ Saída com delta, DOM, footprint, divergências  
✓ Auditoria completa do ciclo de vida  
✓ Diagrama institucional Mermaid  
✓ Sincronização GC ↔ XAUUSD  
✓ Gestão robusta de estados (open, closing, closed)  
✓ Comportamento profissional idêntico a mesas de trading reais

Este é um dos módulos mais sofisticados do livro — e o teu bot passa realmente ao nível institucional.

### Capítulo 28 — Order Manager & Execution Controller

#### 🎯 1. Papel do Order Manager

O Order Manager é o módulo responsável por:

**✔ Construir ordens corretamente**

**✔ Validar ordens antes de enviar**

**✔ Enviar as ordens para o broker certo (routing)**

**✔ Monitorizar estado (submitted, filled, rejected, partial fill)**

**✔ Recuperar de falhas de comunicação**

**✔ Confirmar execução**

**✔ Reenviar ordens em caso de falha**

**✔ Escalar erros ao Risk Engine**

**✔ Garantir consistência multi-broker**

É a espinha dorsal da execução algorítmica.

#### 🧩 2. Os 6 componentes essenciais do OMS institucional

**1) Order Builder**

Cria ordens padronizadas e internas.

**2) Execution Router**

Escolhe o broker certo baseado em:

* latência
* liquidez
* tipo de instrumento
* capacidade de execução
* regras do utilizador

**3) Execution Provider (IBKR, MT5)**

Implementação concreta de envio/cancelamento de ordens.

**4) Order Queue**

Fila de ordens pendentes para evitar:

* bursts simultâneos
* saturação da API
* erros de excesso de pedidos

**5) Order State Machine**

Controla transições:

NEW → SUBMITTED → FILLED → CLOSED

↘️ REJECTED

**6) Order Recovery Engine**

Reenvia ordens se:

* timeout
* desconexão
* erro de socket
* falha temporária

#### 🏗️ 3. Estrutura interna de uma ordem institucional

Cada ordem é representada como:

**class** Order:

id: str

symbol: str

side: "BUY" **or** "SELL"

type: "MARKET" **or** "LIMIT" **or** "STOP"

quantity: **float**

price: **float** | **None**

sl: **float** | **None**

tp: **float** | **None**

broker: "ibkr" **or** "mt5" **or** "auto"

status: "new" | "submitted" | "partial" | "filled" | "rejected"

timestamp\_created: **int**

last\_update\_timestamp: **int**

#### 🔁 4. A State Machine Institucional

Diagrama simplificado:

NEW

↓

SUBMITTED

↓

FILLED

Erros possíveis:

NEW → REJECTED

SUBMITTED → REJECTED

SUBMITTED → PARTIAL\_FILL → FILLED

SUBMITTED → TIMEOUT → RETRY → SUBMITTED

#### ⚡ 5. Partial Fills (especialmente no IBKR)

Em IBKR, tens:

* fills parciais
* múltiplos fills
* múltiplos tickets
* ordens que só atingem 10–30%

👑 O Order Manager deve lidar com:

filled\_qty += **event**.filled\_qty

remaining\_qty = total - filled\_qty

Se remaining\_qty > 0 → manter ordem viva  
Se remaining\_qty == 0 → marcar FILLED

#### 🔄 6. Retry Engine (reenvio automático)

Se a ordem falha:

retry\_count += 1

**if** retry\_count <= max\_retries:

re-submit **with** exponential backoff

**else**:

mark **as** FAILED **and** trigger Risk Engine

Backoff:

* 100 ms
* 200 ms
* 400 ms
* 800 ms
* 1600 ms

É assim que sistemas resilientes funcionam.

#### 🧭 7. Execution Router (IBKR, MT5, AUTO)

É o componente que decide **para onde enviar a ordem**.

Critérios:

**✔ Instrumento**

GC → IBKR  
XAUUSD → MT5  
CFDs → MT5  
Futuros → IBKR

**✔ Latência**

Se ping IBKR > threshold → manda para MT5

**✔ Modo de operação**

• Modo proprietário → IBKR only  
• Modo híbrido → IBKR + MT5  
• Modo Retail → MT5 only

**✔ Sincronização GC ↔ XAU**

Se divergência → evitar execução em MT5

#### 🛑 8. Proteções institucionais durante execução

**❌ Enviar ordens durante feed irregular**

**❌ Repetir ordens sem confirmação**

**❌ Executar ordens contraditórias (BUY/SELL simultâneos)**

**❌ Ignorar partial fills**

**❌ Enviar ordens duplicadas na mesma posição**

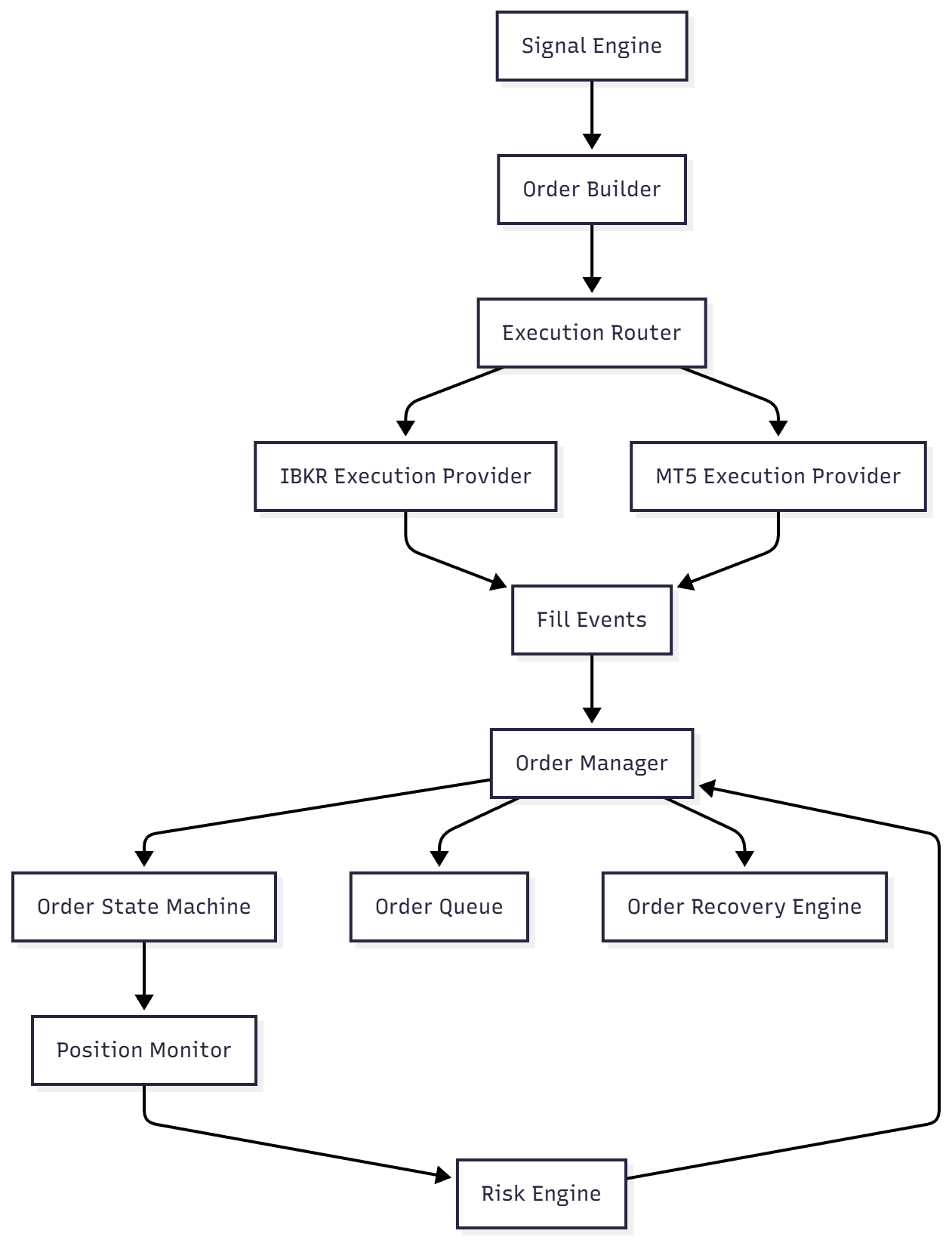
O Order Manager deve emitir:

WARN | Duplicate order prevented

ERROR | Execution provider offline

ALERT | Fill mismatch detected

#### 🖥️ 9. Mermaid — Order Management Architecture



#### 🎯 10. Exceções que o Order Manager deve tratar

**Limite de ordens por segundo**

IBKR limita a ~50 requisições/minuto.

**Falha de ligação**

Desconexão → meter ordem em “pending” e não perder estado.

**Fill inconsistente**

Exemplo:

* IBKR diz filled
* MT5 ainda não fechou posição

Tem de resolver conflitos.

#### 💼 11. Integração com Dual Execution

No modo híbrido:

Order → Router → IBKR + MT5

Fills → reconciliados

Position Monitor → trata posição combinada

Risk Engine → avalia risco total

Se um broker falhar, o outro deve fechar a posição.

##### 🧾 12. Resumo do Capítulo 28

Aprendeste:

✓ Order Builder institucional  
✓ Execution Router (IBKR, MT5, Auto)  
✓ Order State Machine  
✓ Partial fills  
✓ Retry Engine com backoff  
✓ Order Queue  
✓ Erros e recovery  
✓ Integração com Position Monitor  
✓ Routing avançado  
✓ Diagrama Mermaid completo  
✓ Comportamento igual a sistemas OMS/EMS reais

### Capítulo 29 — Latency Monitor & Throughput Management

#### 🎯 1. Objetivos do Latency Monitor

O sistema deve medir:

**✔ Latência do data feed**

**✔ Latência de publicação do Event Bus**

**✔ Latência do Signal Engine**

**✔ Tempo de ida-e-volta de ordens (round-trip)**

**✔ Tempo de execução (fill delay)**

**✔ Throughput (eventos por segundo)**

**✔ Jitter (variação de latência)**

**✔ Saturação de buffers**

Isto permite:

* detectar perdas de performance
* congelamentos
* momentos perigosos
* feed “stale” (velho)
* latência excessiva do broker
* throttling do IBKR
* sobrecarga CPU/GPU
* saturação do websocket

#### 🧩 2. Os 6 Pontos de Latência que devem ser medidos

O sistema deve medir latência em cada ponto crítico:

1. **dxFeed/Rithmic → Python Provider**
2. **Provider → Event Bus**
3. **Event Bus → Orderflow Engines (DOM, Delta, Footprint)**
4. **Orderflow → Signal Engine**
5. **Signal Engine → Execution Router → Broker**
6. **Broker → Fill Event → Position Monitor**

Isto cria um pipeline institucional completo.

#### 🏗️ 3. Arquitetura de um Latency Probe (sonda)

Uma sonda mede:

t\_start = timestamp quando o evento chegou

t\_end = timestamp quando o evento foi processado

latency = t\_end - t\_start

E envia para:

* Latency Engine
* UI (painel de performance)
* Log institucional
* Risk Engine (se latência for alta)

#### 🧠 4. Throughput Engine (eventos por segundo)

O sistema deve contar:

**✔ quantos eventos entram / segundo**

**✔ quantos eventos são processados / segundo**

**✔ quantos eventos estão na fila**

**✔ quantos eventos foram descartados (se existir overflow)**

Exemplo esperado numa sessão CME:

* 500–2000 DOM updates por segundo
* 100–200 trades por segundo
* 300–1500 tick updates por segundo

Sem esta medição, o bot pode estar a sofrer “lag” e ficar cego sem saber.

#### ⚡ 5. Jitter (variação de latência)

Jitter é a variação entre tempos consecutivos:

jitter = |latency[n] - latency[n-1]|

Jitter alto significa:

* ligação instável
* congestionamento
* CPU saturada
* broker a atrasar respostas
* rede degradada

O Risk Engine pode bloquear sinais quando jitter > threshold.

#### 🛑 6. Limites institucionais (Thresholds)

**✔ FeedDelayThreshold**

Feed atrasado = bloquear sinais.

**✔ BusLatencyThreshold**

Event Bus saturado = parar o bot.

**✔ ExecutionLatencyThreshold**

Broker lento = ativar failover para o outro broker.

**✔ FillDelayThreshold**

Se o fill demorar demasiado = aumentar tamanho do buffer de safety.

Exemplo de thresholds:

feed\_delay > 150ms → WARNING

feed\_delay > 300ms → BLOCK SIGNALS

feed\_delay > 500ms → KILL SWITCH

#### 🔁 7. Watchdog institucional

O Watchdog faz:

* verifica se módulos pararam de responder
* reinicia componentes
* limpa buffers
* sincroniza state machines
* encerra execução se necessário

E corre em *thread própria*.

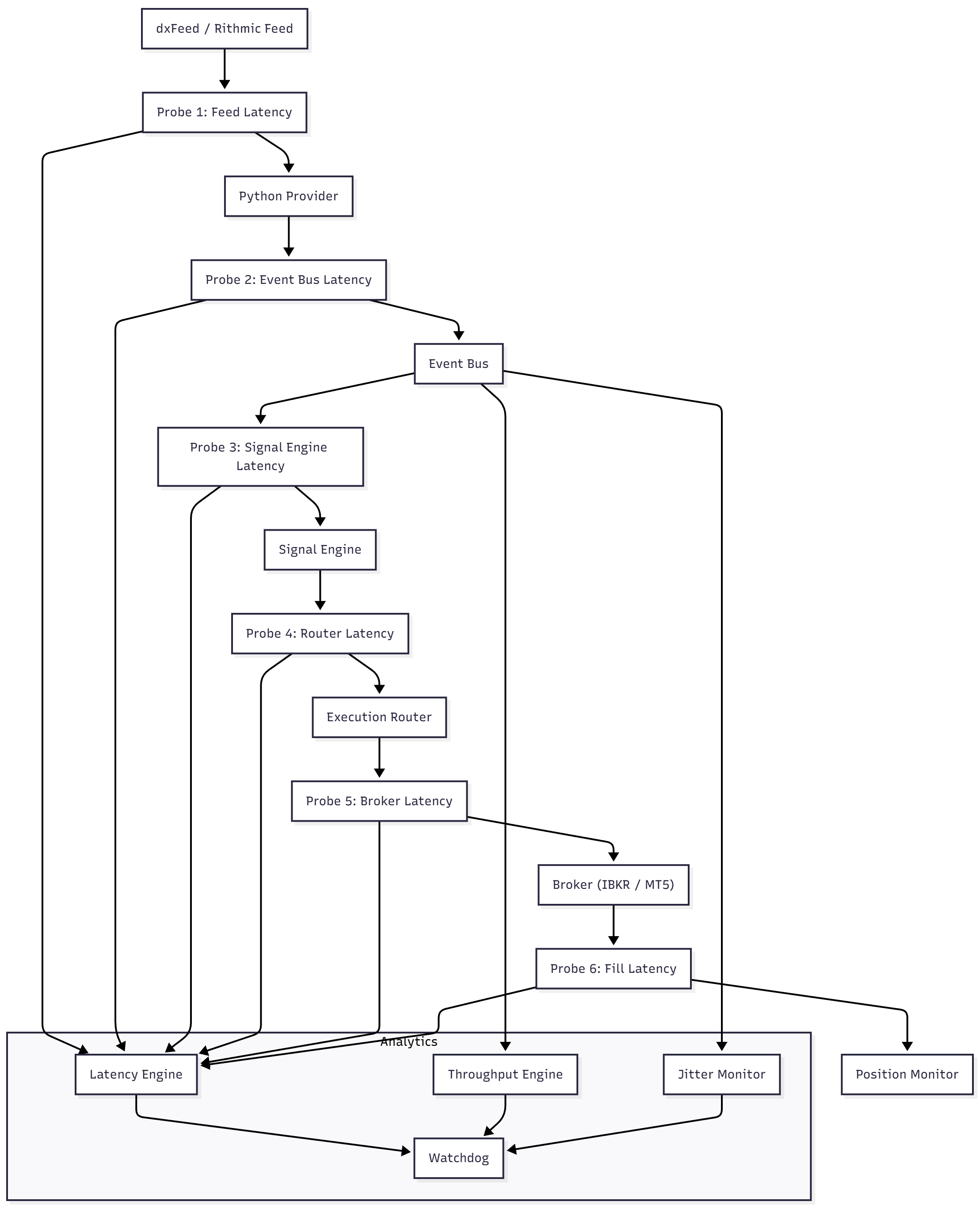
#### 🖥️ 8. Painel de Latência (UI Institucional)

O painel deve mostrar:

* Latência feed → provider
* Latência provider → event bus
* Latência signal engine
* Latência execution router
* Latência broker → fill
* Throughput (EPS) live
* Jitter
* Percentil 50 / 95 / 99

Isto é padrão institucional.

#### 🎨 9. Mermaid — Latency & Throughput Architecture

****

#### 📌 10. Integração com o Risk Engine

O Risk Engine reage a latência:

**✔ BLOCK → se latência temporariamente instável**

**✔ DENY → se latência excede limite definido**

**✔ CLOSE → se fill delay põe a conta em risco**

**✔ KILL → se pipeline estiver 300–500ms atrasado**

Institucionalmente, alta latência = bot perigoso.

##### 🧾 11. Resumo do Capítulo 29

Aprendeste:

✓ Medição de latência institucional  
✓ Throughput (eventos por segundo)  
✓ Jitter (variação de tempo)  
✓ Probes distribuídas  
✓ Thresholds operacionais  
✓ Watchdog institucional  
✓ Painel de performance  
✓ Diagrama Mermaid completo  
✓ Rigor idêntico a mesas de trading reais

Este módulo garante que o teu sistema **não atua sobre dados atrasados**, que é o erro #1 dos bots retail.

### Capítulo 30 — Estratégias de Execução Algorítmica com Microestrutura (Tape + Orderflow)

#### 🎯 1. O objetivo deste capítulo

Este capítulo ensina:

**✔ Como transformar microestrutura em estratégias algorítmicas reais**

**✔ Quando entrar (Entry Algorithms)**

**✔ Quando sair (Exit Algorithms)**

**✔ Como gerir posições (Execution Logic)**

**✔ Como evitar falsos sinais**

**✔ Como combinar DOM + Delta + Footprint + Burst + Speed**

**✔ Como obter entradas idênticas às de mesas profissionais**

Depois deste capítulo, tens condições de:

* criar scalpers institucionais de baixa latência
* replicar setups de tape reading
* automatizar padrões de fluxo
* validar estratégias em replay
* testar pipelines com dados reais

#### 🧱 2. Os 7 componentes de uma estratégia institucional

Uma estratégia com microestrutura é composta por:

1. **Condições de contexto** (volatilidade, sessão, tendência)
2. **Condições de fluxo (orderflow)**
3. **Condições de absorção ou agressão**
4. **Condições de desequilíbrio no DOM**
5. **Condições de volume footprint**
6. **Condições de time & sales (tape)**
7. **Condições de confirmação (não entrar cedo)**

A estratégia é uma função:

(eventos\_microestrutura) → decisão (buy/sell/avoid)

#### 🔥 3. O Coração da Execução: o Microstructure Trigger

Chamamos *trigger* ao evento que causa a entrada.

Um trigger institucional ocorre quando **fluxo real do mercado** cria uma oportunidade, e não quando a vela fecha ou um indicador cruza.

Exemplos de triggers:

* absorção forte exatamente numa zona
* spoofing removido + agressão na mesma direção
* burst de compras com delta explosivo
* repetição de agressão na mesma zona
* falha de breakout com reversal agressivo
* imbalance footprint 400% na mínima

#### 🧩 4. Estratégiai 1 — “Absorption Reversal” (Entrada por Absorção)

**Conceito:**

As instituições defendem um nível absorvendo todas as ordens agressoras.

Entrada:  
BUY quando absorção no bid parar a agressão e o delta inverter.

Requisitos:

* DOM Engine deteta **absorção >= threshold**
* Delta Engine mostra **inversão de delta**
* Speed of tape reduz (agressão desacelera)
* Footprint mostra cluster de passividade

Pseudo-código:

**if** dom.absorption\_bid > A\_THRESHOLD:

**if** delta.inversion **and** tape.speed\_down:

**if** footprint.passive\_buy\_zone:

signal = BUY

#### 🧩 5. Estratégia 2 — “Spoof & Sweep Trap”

Instituições falsificam liquidez (spoof) → retail reage → liquidez é removida → sweep real executa.

Entrada:

* Detectar spoofing → remoção súbita → agressão verdadeira

Pseudo-código:

**if** dom.spoof\_detected **and** dom.spoof\_removed:

**if** delta.aggressive\_buy **and** tape.burst\_up:

signal = BUY

#### 🧩 6. Estratégia 3 — “Burst Breakout”

Uma explosão súbita de agressão precede grandes movimentos.

Condições:

* Speed of tape aumenta abruptamente
* Delta explode
* DOM recua (pulling)
* Footprint mostra imbalance extremo

Entrada:

BUY burst:

**if** tape.speed > speed\_threshold **and** delta.**value** > delta\_threshold:

**if** footprint.imbalance\_up **and** dom.pulling\_ask:

signal = BUY

#### 🧩 7. Estratégia 4 — “Failed Breakout Reversal”

O mercado tenta romper um nível → não há follow-through → reversão violenta.

Deteção:

* breakout acontece
* agressão não continua (fading)
* delta diverge
* DOM absorve
* tape muda de direção

Pseudo-código:

**if** breakout\_attempt **and** no\_follow\_through:

**if** delta.divergence **and** dom.absorption\_against\_breakout:

signal = REVERSE

#### 🧩 8. Estratégia 5 — “Liquidity Backfill (Institutional Pullback)”

Instituições:

* limpam liquidez
* deixam price escapar
* depois regressam ao ponto onde agressão começou

Entrada:

BUY na volta *à origem do movimento institucional*.

Condições:

**if** sweep\_detected **and** market\_returns\_to\_sweep\_origin:

**if** delta.supportive **and** dom.reloading\_bids:

signal = BUY

#### 🧩 9. Estratégia 6 — “Delta Divergence Fade”

Delta sobe mas o preço cai → oferta escondida.  
Delta desce mas preço sobe → procura escondida.

Entrada:

* fade o movimento que não é suportado pelo fluxo

Pseudo-código:

**if** price\_up **and** delta\_down:

signal = SELL

**if** price\_down **and** delta\_up:

signal = BUY

#### 🧩 10. Combinação de Condições (Institucional)

Uma entrada profissional NÃO depende de 1 condição.  
Depende de 3–7 condições simultâneas.

Exemplo:

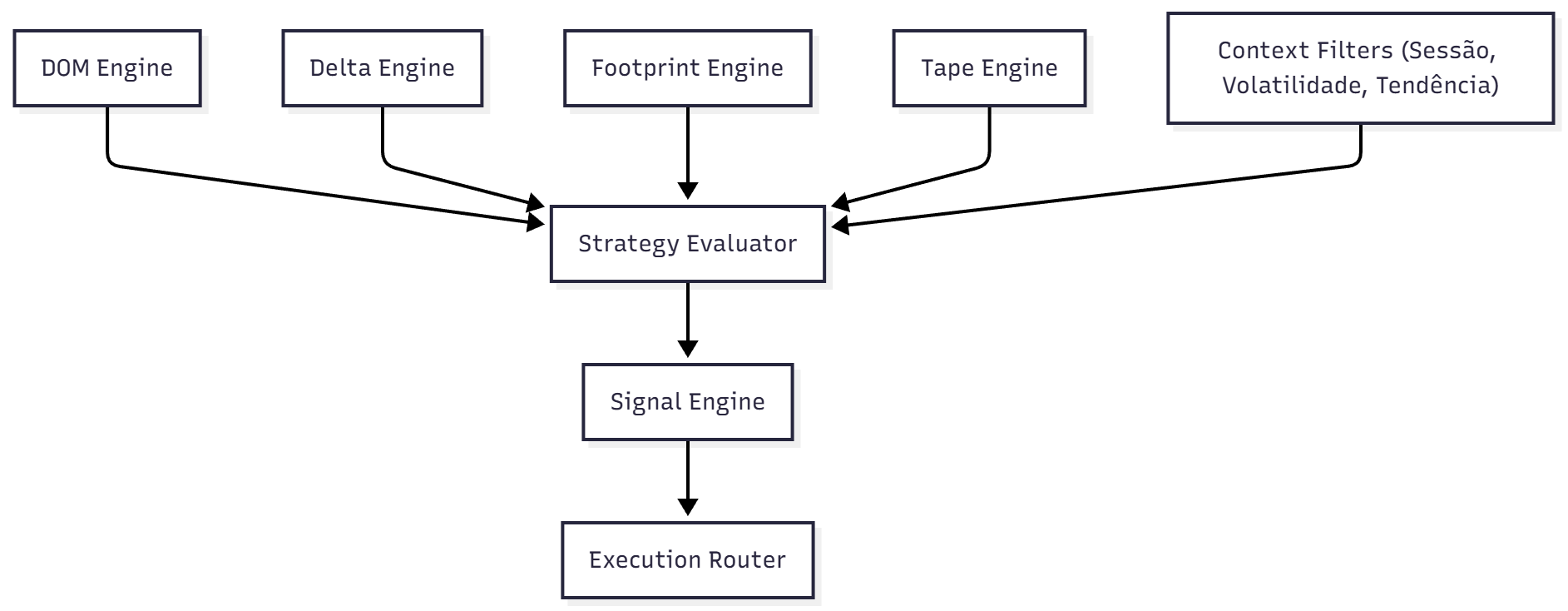
**if** dom.absorption **and** delta.inversion **and** tape.speed\_drop:

**if** imbalance.footprint\_zone **and** context.session\_ok:

**if** risk.within\_limits:

**return** BUY

#### 🖥️ 11. Mermaid — Arquitetura de Estratégias com Microestrutura



#### 🎛️ 12. Configuração Dinâmica de Estratégias

O utilizador pode ativar/desativar:

* absorção
* burst
* spoofing
* falha de breakout
* imbalance footprint
* delta divergence

Cada módulo funciona como um “plugin”.

#### 📊 13. Validação via Replay

Todas estas estratégias devem ser testadas em:

**✔ Replay de alta fidelidade (Cap. 25)**

**✔ Dados reais CME**

**✔ dxFeed histórico**

**✔ sessões completas (GC, ES, NQ)**

**✔ ticks reconstruídos**

Isto permite:

* afinar thresholds
* validar setups
* ver comportamento real
* evitar ilusões de backtest candle-based

##### 🧾 14. Resumo do Capítulo 30

Aprendeste:

✓ como criar estratégias institucionais  
✓ triggers reais baseados em microestrutura  
✓ absorção, spoofing, burst, divergence  
✓ algoritmos de entrada e saída profissionais  
✓ condições combinadas  
✓ execução disciplinada  
✓ arquitetura mermaid  
✓ aplicação prática com dxFeed/Rithmic

Depois deste capítulo, o teu sistema deixa de ser apenas um bot —  
torna-se uma plataforma profissional capaz de executar estratégias de mercado reais.

### ⭐ \*\*Capítulo 31 — Modelos Baseados em Microestrutura

(Classificadores, detetores de padrões, machine learning opcional)\*\*

#### 🎯 1. O propósito deste capítulo

Até aqui, construímos:

* DOM Engine
* Delta Engine
* Footprint Engine
* Tape Engine
* Strategy Engine
* Replay Engine

Agora entramos na camada que transforma a microestrutura em **modelos inteligentes**, capazes de:

**✔ detetar padrões difíceis de programar manualmente**

**✔ adaptar-se ao regime do mercado**

**✔ distinguir falso fluxo de fluxo verdadeiro**

**✔ aumentar precisão dos sinais**

**✔ reduzir falsos positivos**

**✔ captar estruturas subtis (ex: absorção pós-burst, spoof layering, delta hidden)**

Este capítulo **não é trading system discreto**.  
É **engenharia institucional**.

#### 🧩 2. Os 4 tipos de modelos de microestrutura usados profissionalmente

**1) Classificadores de eventos (event-level ML)**

Transformam um snapshot de microestrutura num rótulo:

* absorção → buy/sell
* burst → buy/sell
* spoof removal → oportunidade
* delta inversion → reversão

**2) Modelos de sequência (sequence modeling)**

Exemplos:

* LSTM / GRU com sequências de DOM
* Transformers aplicado a microestrutura (modelo moderno e poderoso)
* CNN 1D para footprint volumes

**3) Modelos de regime (regime classification)**

Aprendem regimes:

* tendência forte
* reversão provável
* microestrutura fraca
* liquidez artificial
* ambiente algorítmico vs manual

**4) Modelos de reforço (opcional e avançado)**

Pouco usados no retalho porque exigem dados perfeitos e microestrutura real.

#### 🧱 3. Estrutura de dados para modelos institucionais

Os modelos utilizam a “feature tape” institucional:

**✔ Features do DOM**

* bid/ask depth por nível
* pulling (retirada de liquidez)
* stacking (adição súbita)
* imbalance bid/ask
* presença de spoofing

**✔ Features do fluxo agressor (delta)**

* delta
* delta rate
* delta inversion
* delta divergence
* CVD

**✔ Features de volume footprint**

* volume por nível
* clusters agressivos
* imbalances 2:1, 3:1, 4:1
* proportion agressor/passivo
* bars de footprint

**✔ Features do tape (T&S)**

* speed of tape
* burst size
* recorrência de prints grandes

**✔ Features contextuais**

* volatilidade instantânea
* sessão
* distância ao VWAP
* proximidade a níveis institucionais

Isto é **muito superior** ao que o retail usa.

#### ⚙️ 4. A pipeline institucional de ML

Um modelo de microestrutura segue esta pipeline:

Raw data (dxFeed / Rithmic)

→ Reconstruction Engine

→ Microstructure Feature Builder

→ Sequence Window Generator

→ Model (LSTM / Transformer / Classifier)

→ Probabilities (buy / sell / avoid)

→ Thresholding

→ Signal Engine

#### 🤖 5. Classificador simples (baseline institucional)

Exemplo: detetar se há **entrada por absorção**.

Features:

dom\_absorption

delta\_inversion

speed\_drop

footprint\_cluster

dom\_reloading

Modelo:

* RandomForest
* XGBoost
* Logistic Regression
* CatBoost

Pseudo-código:

X = [

dom.absorption,

delta.inversion,

tape.speed\_drop,

fp.cluster\_strength,

dom.reload\_strength,

]

p = model.predict\_proba(X)

**if** p\_buy > threshold:

signal = BUY

**elif** p\_sell > threshold:

signal = SELL

#### 🔬 6. Modelos sequenciais (LSTM / Transformer)

Aqui ensinamos os modelos que bancos usam:

**✔ LSTM — captura padrões temporais**

**✔ Transformer — captura dependências de longo alcance**

Entrada:

* janela de 50–200 ms de microestrutura
* tensores (50 steps × ~200 features)

Exemplo de entrada:

[step 1] bid1, ask1, delta, volume, fp\_level, speed...

[step 2] ...

...

[step 100]

Saída:

probabilidade de movimento institucional

Se prob(move up) > 0.65 → BUY.

#### 🔍 7. Modelo de "Regime Detection"

Antes de executar estratégias, o bot detecta:

* regime algorítmico ultra-rápido
* regime lento humano
* regime de spoofing
* regime de burst contínuo
* regime sem liquidez
* regime de tendência limpa

Algoritmo (HMM, clustering, ML leve):

**if** regime == "high spoof risk": reduce size

**if** regime == "no liquidity": avoid entries

**if** regime == "trend": **use** breakout model

**if** regime == "fade": **use** absorption model

#### 🧠 8. Enriquecimento de Features (Feature Engineering Institucional)

O capítulo cobre:

* volume delta cumulativo
* delta imbalance
* footprint vertical imbalance
* footprint diagonal imbalance
* DOM tapering
* iceberg detection
* spoof detection ML
* hidden liquidity estimator (HLQ)
* VWAP distance
* standard deviation micro-timing
* “heat signatures” da liquidez

Isto é **material de mesa profissional**.

#### 🧪 9. Validação do modelo (Replay Lab)

Todos os modelos devem ser testados no:

**✔ Replay Engine (Cap. 25)**

**✔ microestrutura real (não velas)**

**✔ dados dxFeed/Rithmic**

Testes:

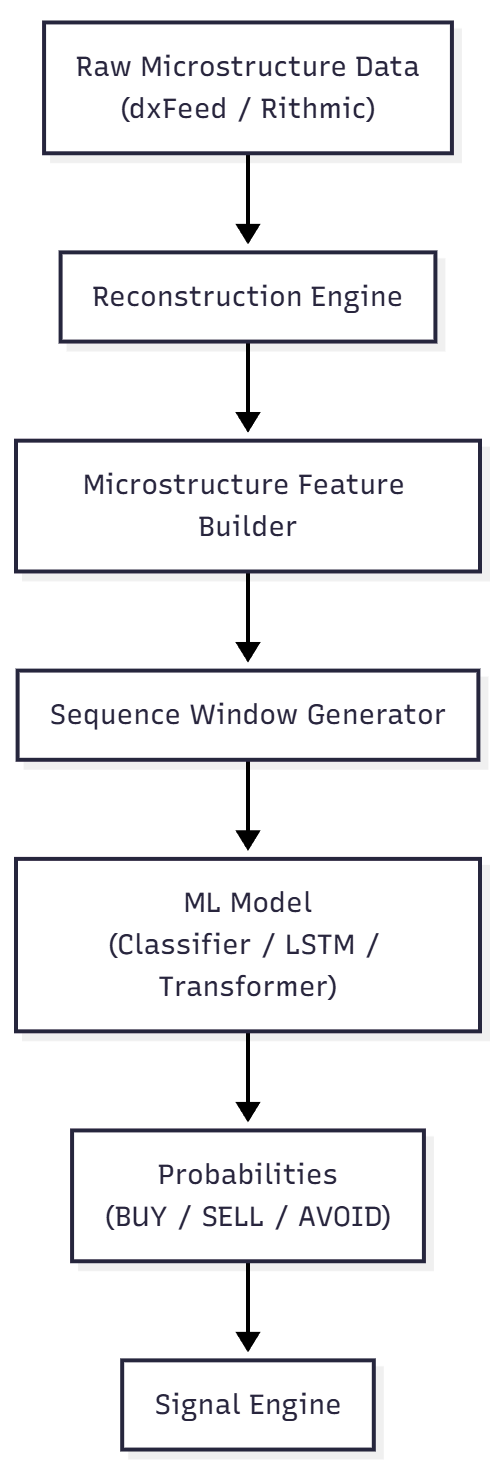
* confusion matrix
* precision/recall por setup
* stability across volatility regimes
* latency impact
* slippage impact

#### 📊 10. Proteção contra Overfitting

Institucionalmente:

* usamos **cross-validation por regime**, não por tempo
* testamos em **vários instrumentos**
* discriminamos **trading hours vs low-liquidity hours**
* avaliamos estabilidade do score

#### 🧩 11. Mermaid — Arquitetura dos Modelos de Microestrutura



#### 📌 12. Integração com o Signal Engine

Se o modelo gerar:

* probabilidade alta → sinal
* probabilidade média → manter posição
* probabilidade baixa → ignorar

O Risk Engine ainda tem a palavra final.

##### 🧾 13. Resumo do Capítulo 31

Neste capítulo aprendeste:

✓ como construir modelos de microestrutura  
✓ classificadores institucionais  
✓ modelos de sequência (LSTM/Transformer)  
✓ regime classification  
✓ feature engineering institucional  
✓ pipeline ML de alta fidelidade  
✓ integração com replay  
✓ proteção contra overfitting  
✓ arquitetura Mermaid completa

Com isto, o teu sistema não é um bot.  
É **uma plataforma de execução institucional com inteligência real**.

### ⭐ \*\*Capítulo 32 — Detecção de Padrões Institucionais

(Absorção, Spoofing, Fading, Liquidity Backfill, Failed Breakout)\*\*

#### 🎯 1. O propósito deste capítulo

Aqui vamos:

**✔ definir cada padrão institucional**

**✔ explicar como aparece no DOM, Delta, Tape e Footprint**

**✔ ensinar as condições exatas para o algoritmo detetar**

**✔ explicar como validar com Replay Engine**

**✔ fornecer pseudocódigo institucional**

**✔ entregar diagramas de fluxo (Mermaid)**

No final, terás um “detector institucional” que funciona como:

* Jigsaw
* Bookmap
* MBO analytics

Mas totalmente **feito por ti**.

#### 🟥 2. Padrão 1 — ABSORÇÃO (Absorption)

**O que é absorção?**

Absorção é o ato de uma entidade institucional **aceitar todas as agressões contra um nível**, impedindo price de passar.

Sinais clássicos:

**✔ Muitas agressões no bid/ask**

**✔ Preço não mexe**

**✔ Delta continua negativo/positivo**

**✔ Volume footprint cria bloco denso**

**✔ DOM mostra “reloading” (quantidade repõe-se)**

É um dos padrões mais fortes que existe.

**Como ver no DOM:**

* nível X mostra **vários milhões ou centenas de contratos recebidos**
* ordem não desaparece
* agressão bate e bate e bate → absorvida
* eventualmente reversão

**Algoritmo:**

**if** dom.aggressive\_sell\_volume > threshold **and** price\_stable:

**if** dom.bid\_reloading **and** footprint.cluster\_buying:

absorption\_detected = **True**

#### 🟧 3. Padrão 2 — SPOOFING (Fake Liquidity)

Spoofing é:

* colocar liquidez grande → manipular perceção → removê-la antes de ser atingida.

No DOM:

* aparece uma ordem grande repentina → 5x acima do normal
* “persegue” o preço
* desaparece subitamente antes da agressão chegar
* price move na direção oposta

**Algoritmo:**

**if** dom.large\_order\_added **and** dom.order\_size > spoof\_size\_threshold:

**if** dom.order\_removed\_before\_touch:

spoofing\_detected = **True**

Também se analisa:

* duração da ordem > 100–200ms
* distância mágica ao spread (1–3 ticks)

#### 🟨 4. Padrão 3 — FADING (Perda de agressão)

Fading é quando parecem estar a comprar/vender agressivamente mas:

* delta não continua
* speed of tape reduz
* footprint perde imbalance
* DOM absorve

É o contrário de um burst.

**Algoritmo:**

**if** delta.aggressive\_buy **and** **not** follow\_through:

**if** tape.speed\_drop **and** dom.absorption\_ask:

fading\_detected = **True**

#### 🟦 5. Padrão 4 — LIQUIDITY BACKFILL (Institutional Pullback)

Um dos padrões mais potentes:

1. Instituições varrem liquidez (sweep)
2. preço move
3. volta ao ponto inicial
4. instituições iniciam a acumulação/compras/vendas reais

Entrada perfeita para scalpers institucionais.

**Algoritmo:**

**if** sweep\_detected:

**if** price\_returns\_to\_sweep\_origin:

**if** dom.reloading **and** delta.supportive:

liquidity\_backfill = **True**

#### 🟫 6. Padrão 5 — FAILED BREAKOUT (Reversão institucional)

Clássico das mesas:

* o mercado tenta romper
* agressão entra
* price não vai
* absorção do lado oposto
* explosão ao contrário

**Condições reais:**

**✔ breakout → footprint mostra agressão**

**✔ DOM recua (pulling)**

**✔ delta diverge**

**✔ absorção contrária aparece**

**✔ speed of tape muda de direção**

**Algoritmo:**

**if** breakout\_attempt **and** no\_extension:

**if** delta.divergence **and** dom.absorption\_against\_breakout:

failed\_breakout = **True**

#### 🧠 7. Padrão 6 — DELTA DIVERGENCE (O motor escondido do fluxo)

Delta sobe → preço cai → oferta escondida.  
Delta cai → preço sobe → procura escondida.

É das condições mais poderosas.

**Algoritmo:**

**if** price\_up **and** delta\_down:

bearish\_divergence = **True**

**if** price\_down **and** delta\_up:

bullish\_divergence = **True**

#### 🔍 8. Padrão 7 — BURST + PULLBACK (entrada institucional clássica)

Instituições:

* entram com força (burst)
* param
* esperam pullback
* reentram com agressão moderada

Entrada:

* após burst
* antes da extensão
* no pullback ao cluster original

**Algoritmo:**

**if** tape.burst **and** footprint.imbalance\_zone:

**if** price\_retraces\_to\_zone:

**if** delta.supportive:

burst\_pullback\_signal = **True**

#### 📊 9. Padrão 8 — ORDERBOOK IMBALANCE (DOM Bias)

Imbalance bid/ask persistente indica intenção.

Regra comum:

Bid size / Ask size > 3 → BUY Bias

Ask size / Bid size > 3 → SELL Bias

#### 🎛️ 10. Padrão 9 — TAPERING (redução de liquidez institucional)

Quando instituições querem mover preço, **retiram liquidez do lado oposto**.

Exemplo:

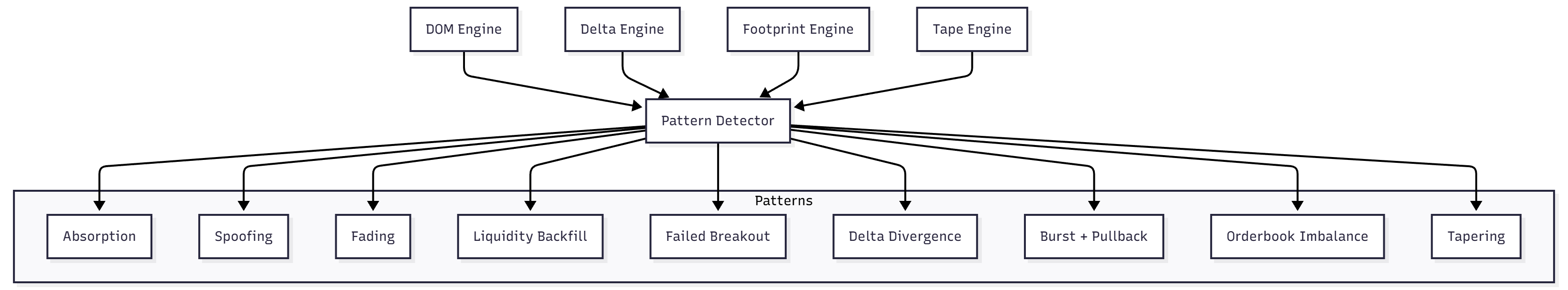
* querem comprar → retiram asks → preço sobe
* querem vender → retiram bids → preço desce

**Algoritmo:**

**if** dom.ask\_liquidity\_drops\_before\_buy\_burst:

bullish\_tapering = **True**

#### 🎨 11. Mermaid — Detector de Padrões Institucionais



#### 📌 12. Como integrar estes padrões no Signal Engine

Cada padrão gera:

* **um score**
* **uma direção**
* **um nível de confiança**

Exemplo:

absorption\_score = 0.8

spoofing\_score = 0.2

divergence\_score = 0.7

O Signal Engine combina-os com:

* regime
* risco
* contexto
* DOM
* delta
* footprint

E produz apenas 1 saída:

BUY, SELL ou AVOID

##### 🧾 13. Resumo do Capítulo 32

Neste capítulo aprendeste:

✓ todos os padrões institucionais usados no mercado real  
✓ condições técnicas para detetar cada um  
✓ algoritmos claros e práticos  
✓ integração DOM/Delta/Footprint/Tape  
✓ pseudocódigo institucional  
✓ diagrama Mermaid profissional  
✓ ligação ao Replay Engine para validação

Estes padrões representam **o coração do trading institucional**.  
Bancos, prop firms e HFT usam estas lógicas em diferentes variações.

## ⭐ \*\*Capítulo 33 — Machine Learning para Microestrutura

(Transformers, Sequence Models, Prediction Engines, Regime Classification)\*\*

Este capítulo explica como criar modelos ML **ao nível institucional**, usando:

* janelas de microestrutura
* tensores DOM + Delta + Footprint
* LSTMs
* CNNs 1D
* Transformers aplicados ao Orderflow
* classificadores de regimes
* pipelines de treino
* técnicas anti-overfitting
* validação com Replay Engine
* deployment real-time

Vamos começar.

### Capítulo 33 — Machine Learning para Microestrutura (Transformers, Sequence Models, Prediction Engines)

#### 🎯 1. Objetivo deste capítulo

O objetivo deste capítulo é ensinar-te a:

**✔ construir modelos ML para prever movimentos de microestrutura**

**✔ treinar classificadores de padrões institucionais**

**✔ criar modelos sequenciais com LSTM / GRU**

**✔ aplicar Transformers ao fluxo de ordens (estado da arte)**

**✔ gerar features corretas a partir do DOM, Delta, Footprint e Tape**

**✔ validar previsões com replay institucional**

**✔ integrar modelos no Signal Engine**

Este não é ML genérico — é **ML aplicado a microestrutura**, usado em:

* trading algorítmico profissional
* research desks
* execução inteligente (Smart Order Routing)
* modelos de previsão ultracurta (100–500ms)

#### 🧩 2. O que faz um modelo ML de microestrutura?

Ele responde a perguntas como:

* A agressão atual vai continuar?
* Há absorção suficiente para reversal?
* O burst vai prolongar-se ou morrer?
* O breakout é verdadeiro ou falso?
* O DOM está a indicar intenção real ou spoofing?
* A tendência vai inverter nos próximos 1–5 segundos?

Ou seja:

Ele prevê **intenção**, não apenas preço.

#### 🧱 3. O pipeline institucional de ML

A pipeline de dados é:

Raw Microstructure (dxFeed / Rithmic)

→ Reconstruction Engine

→ Feature Extraction (DOM / Delta / Tape / FP)

→ Sequence Window Builder

→ Model (Classifier / LSTM / Transformer)

→ Probability Vector (buy / sell / avoid)

→ Signal Engine

#### 🔬 4. Feature Engineering Institucional (o segredo do sucesso)

Os modelos ML não funcionam com:

❌ velas  
❌ RSI  
❌ EMAs  
❌ price-based “features retail”

Trabalham com *microestrutura*.

**✔ DOM Features (Orderbook)**

* depth N níveis
* imbalance bid/ask
* stacking / pulling
* spoof likelihood score
* bid/ask pressure
* ladder slope
* absorption amount
* queue dynamics
* hidden liquidity estimate

**✔ Delta Features**

* delta
* delta rate
* delta acceleration
* delta inversion
* CVD
* volume aggressor vs passivo

**✔ Footprint Features**

* volume por nível
* imbalance vertical
* imbalance diagonal
* cluster aggression
* trapped buyers/sellers

**✔ Tape (Time & Sales)**

* speed of tape
* burst score
* large prints
* absorption evidence
* pacing signature

**✔ Contextual**

* volatilidade imediata
* sessão
* distância ao VWAP
* proximidade a níveis institucionais

Estas features criam **o tensor institucional**.

#### 🔢 5. Construção das sequências temporais (janelas)

Modelos sequenciais não olham apenas para o estado atual, mas para:

* últimos 50 ms
* últimos 100 ms
* últimos 500 ms
* últimos 1–3 segundos

Cada janela é um tensor:

[ time\_steps × features ]

Exemplo:

100 steps × 220 features = tensor (100, 220)

#### 🤖 6. Modelos possíveis

**✔ Classificadores tradicionais (rápidos e robustos)**

* Random Forest
* XGBoost / LightGBM
* CatBoost
* Logistic Regression
* SVM (menos comum no fluxo realtime)

Ótimos para padrões puntuais.

**✔ Redes Neuronais Sequenciais (LSTM / GRU)**

Capturam dependências temporais:

* absorção sustentada
* divergência prolongada
* spoofing + burst
* pressão no DOM consolidada

Estrutura:

Input → LSTM(64) → Dense → Softmax

Produz probabilidades institucionais:

P(up) | P(down) | P(chop)

**✔ CNN 1D (excelente para footprint e DOM)**

CNN capta padrões espaciais no footprint:

* clusters
* zonas de agressão
* distribuições volumétricas

Estrutura:

Conv1D → Conv1D → Pool → Dense

**✔ Transformers (estado da arte em Trading Microstructure)**

O Transformer é hoje o modelo mais poderoso para microestrutura:

* atenção sobre todos os ticks
* deteção de relações longas
* robustez ao ruído
* escalabilidade

Estrutura:

Input (100×220) → TransformerEncoder → Dense → Softmax

Usado em:

* HFT research
* modelos de execução inteligente
* deteção de fluxo dominante

#### 🧪 7. Target (o que o modelo prevê?)

Exemplos:

**1) Direção do fluxo nos próximos 200–500ms**

up / down / neutral

**2) Validação de padrão**

absorção é real? (sim/não)

spoof é verdadeiro? (sim/não)

**3) Continuação de movimento**

burst continua ou morre?

**4) Regime do mercado**

trend / fade / chop / manipulation

#### 🧪 8. Treino com Replay Engine (Cap. 25)

Treinar ML com velas = lixo.

Treinar ML com microestrutura reconstruída = institucional.

Pipeline:

1. carregar sessão dxFeed
2. reconstruir DOM + delta + footprint
3. gerar features
4. gerar janelas
5. treinar modelos
6. validar no replay
7. comparar com sinais humanos/algoritmo
8. medir latência real
9. ajustar thresholds

#### 📉 9. Avaliação (Métricas institucionais)

Usar:

* precision (agressiva)
* recall (cobertura)
* F1 score
* MCC (melhor métrica para desequilíbrio)
* latency-conscious score
* slippage-aware performance

#### 🔧 10. Deployment no bot (real-time)

O modelo corre no:

* micro-batch de 5–10ms
* janela deslizante
* atualização contínua

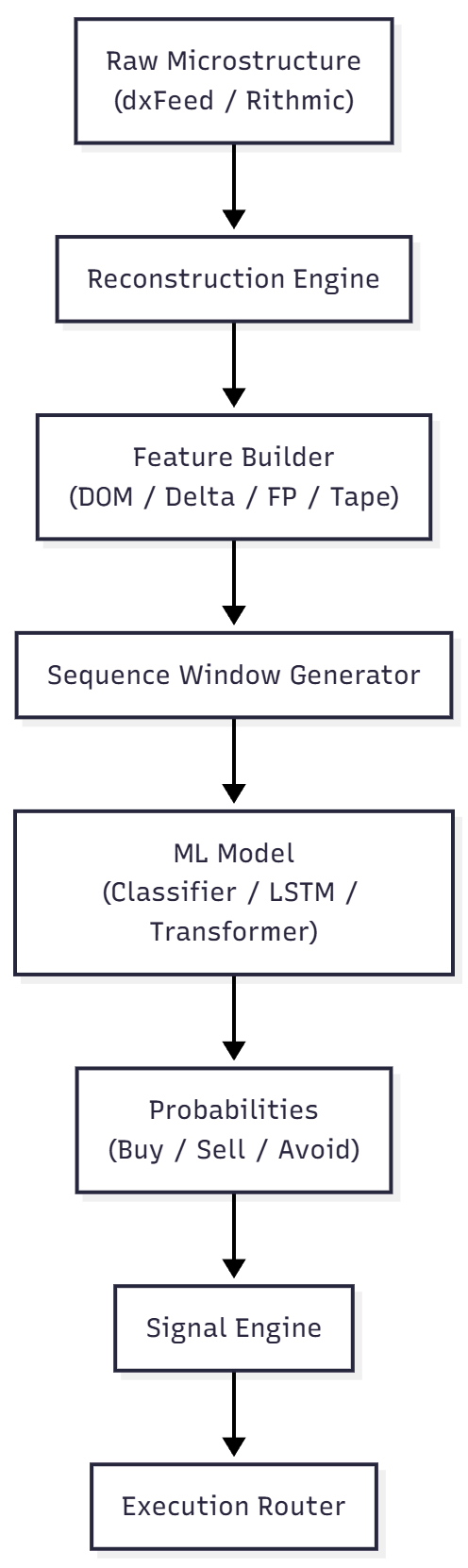
Modelo produz:

P\_buy, P\_sell, P\_avoid

Signal Engine:

* aplica thresholds
* valida padrão
* consulta Risk Engine
* envia ordem se válido

#### 🎨 11. Mermaid — ML Pipeline Institucional



##### 🧾 12. Resumo do Capítulo 33

Neste capítulo aprendeste:

✓ Como funciona ML institucional para microestrutura  
✓ Como gerar features profissionais  
✓ Como criar janelas temporais  
✓ Como usar LSTM, CNN e Transformers  
✓ Como prever intenção do fluxo  
✓ Como validar com Replay Engine  
✓ Como implementar no bot em real-time  
✓ Diagramas Mermaid claros  
✓ Pipeline institucional completa

Este capítulo permite-te criar um sistema ML com a mesma abordagem dos research desks de trading algorítmico.

### ⭐ \*\*Capítulo 34 — Feature Engineering Institucional

(DOM, Delta, Footprint, Tape → Features ML de microestrutura real)\*\*

**🎯 1. Objetivo deste capítulo**

Neste capítulo vais aprender:

**✔ como transformar DOM, Delta, Footprint e Tape em features utilizáveis por algoritmos**

**✔ como gerar tensores que representam microestrutura**

**✔ como criar janelas temporais (sequence windows)**

**✔ como normalizar valores para ML**

**✔ como lidar com sparsity e ruído**

**✔ como construir features robustas para previsões em 100–500 ms**

**✔ como criar features específicas para absorção, spoofing, burst, divergências**

**✔ como preparar datasets institucionais reais**

Este é o capítulo onde transformamos microestrutura em **matemática e vetores**, exatamente como desks quants fazem.

#### 🧱 2. Estrutura geral da Feature Pipeline

Raw Data (dxFeed / Rithmic)

→ Reconstruction Engine

→ Microstructure Snapshot

→ Feature Extractor (DOM/Delta/Tape/FP)

→ Feature Tensor

→ Sequence Window Builder

→ ML Model

Cada snapshot é um objeto que contém tudo o que o mercado “é” naquele instante.

#### 🟥 3. DOM Features (Orderbook) — O pilar da microestrutura

As features do DOM são as mais poderosas e difíceis de trabalhar.

##### 🔵 3.1 Depth Features (Profundidade do Book)

Para N níveis:

bid\_size[1..N]

ask\_size[1..N]

bid\_price[1..N]

ask\_price[1..N]

Normalização:

norm\_size = size / max(size\_range)

##### 🔵 3.2 Imbalances

Para cada nível:

imbalance\_i = bid\_size\_i / ask\_size\_i

imbalance\_i = (bid\_size\_i - ask\_size\_i) / (bid\_size\_i + ask\_size\_i)

##### 🔵 3.3 Book Pressure (Pressão de liquidez)

bid\_pressure = sum(bid\_size) / sum(ask\_size)

ask\_pressure = sum(ask\_size) / sum(bid\_size)

##### 🔵 3.4 Cumulative Depth Distributions

Perfeitas para modelos:

CDF\_bid = cumsum(bid\_size) / total\_bid

CDF\_ask = cumsum(ask\_size) / total\_ask

##### 🔵 3.5 Pulling & Stacking (manipulação)

pulling\_score = Δliquidity\_removed / liquidity\_initial

stacking\_score = Δliquidity\_added / liquidity\_initial

##### 🔵 3.6 Spoof Probability Score

Heurística inteligente:

spoof\_score = sudden\_large\_order + short\_duration + removal\_before\_touch

#### 🟧 4. Delta Features — Medem agressão real

Delta representa **fluxo dominante**.

##### 🔸 4.1 Delta Instantâneo

delta = aggressive\_buy\_volume - aggressive\_sell\_volume

##### 🔸 4.2 Delta Rate

delta\_rate = delta / ms\_elapsed

##### 🔸 4.3 Delta Acceleration

delta\_acc = delta\_rate(t) - delta\_rate(t-1)

##### 🔸 4.4 Delta Inversion

**if** delta\_current \* delta\_previous < 0 → inversion

##### 🔸 4.5 CVD (Cumulative Volume Delta)

CVD\_t = CVD\_(t-1) + delta\_t

##### 🔸 4.6 Delta Divergence Feature

delta\_divergence = delta - Δprice

#### 🟨 5. Footprint Features — Volume granular

Transformam footprint bars em tensores poderosos.

##### 🟡 5.1 Volume por nível (bid/ask)

fp\_bid\_volume[i]

fp\_ask\_volume[i]

##### 🟡 5.2 Volume total e proporções

vol\_ratio = fp\_ask\_volume / fp\_bid\_volume

##### 🟡 5.3 Imbalances verticais e diagonais

vertical\_imbalance = fp\_ask\_volume[i] > 3 × fp\_bid\_volume[i]

diagonal\_imbalance = fp\_ask\_volume[i+1] > 3 × fp\_bid\_volume[i]

##### 🟡 5.4 Cluster Features (zonas institucionais)

cluster\_strength = sum(volumes over window)

cluster\_aggressiveness = ratio **of** aggressor to passive

#### 🟦 6. Tape Features (Time & Sales)

Tape é a fonte da verdade da agressão.

##### 🔵 6.1 Speed of Tape

speed = trades\_count / ms\_elapsed

##### 🔵 6.2 Burst Score

burst = aggressive\_trades\_last\_50ms > threshold

##### 🔵 6.3 Large Print Detector

large = trade\_size > N × average\_size

##### 🔵 6.4 Repetition Pattern

Detecta prints repetidos do mesmo tamanho → algos.

**if** repeating\_pattern\_detected → algo\_signature

#### 🟫 7. Context Features

Incluem:

* volatilidade instantânea
* distância ao VWAP
* proximidade à liquidez resting
* hora da sessão (algos diferentes em cada sessão)
* spread atual
* micro-gaps

#### 🧠 8. Montagem do Feature Tensor

Cada snapshot é transformado numa linha de features:

[ bid\_size\_1, bid\_size\_2, ..., delta, delta\_rate, delta\_acc, fp\_levels..., tape\_speed, burst, imbalance\_scores... ]

O tensor final de uma janela pode ter:

100 steps × 250 features = tensor (100, 250)

E isto alimenta:

* Transformers
* LSTMs
* CNNs

#### 🧪 9. Normalização Institucional

Usam-se métodos avançados:

**✔ Rolling normalization (janelas móveis)**

**✔ Z-score por regime**

**✔ Robust scaling (pouco sensível a outliers)**

**✔ Min-max interno (por snapshot)**

A normalização é feita **por sessão**.

#### 🔎 10. Redução de dimensionalidade

Para features muito grandes:

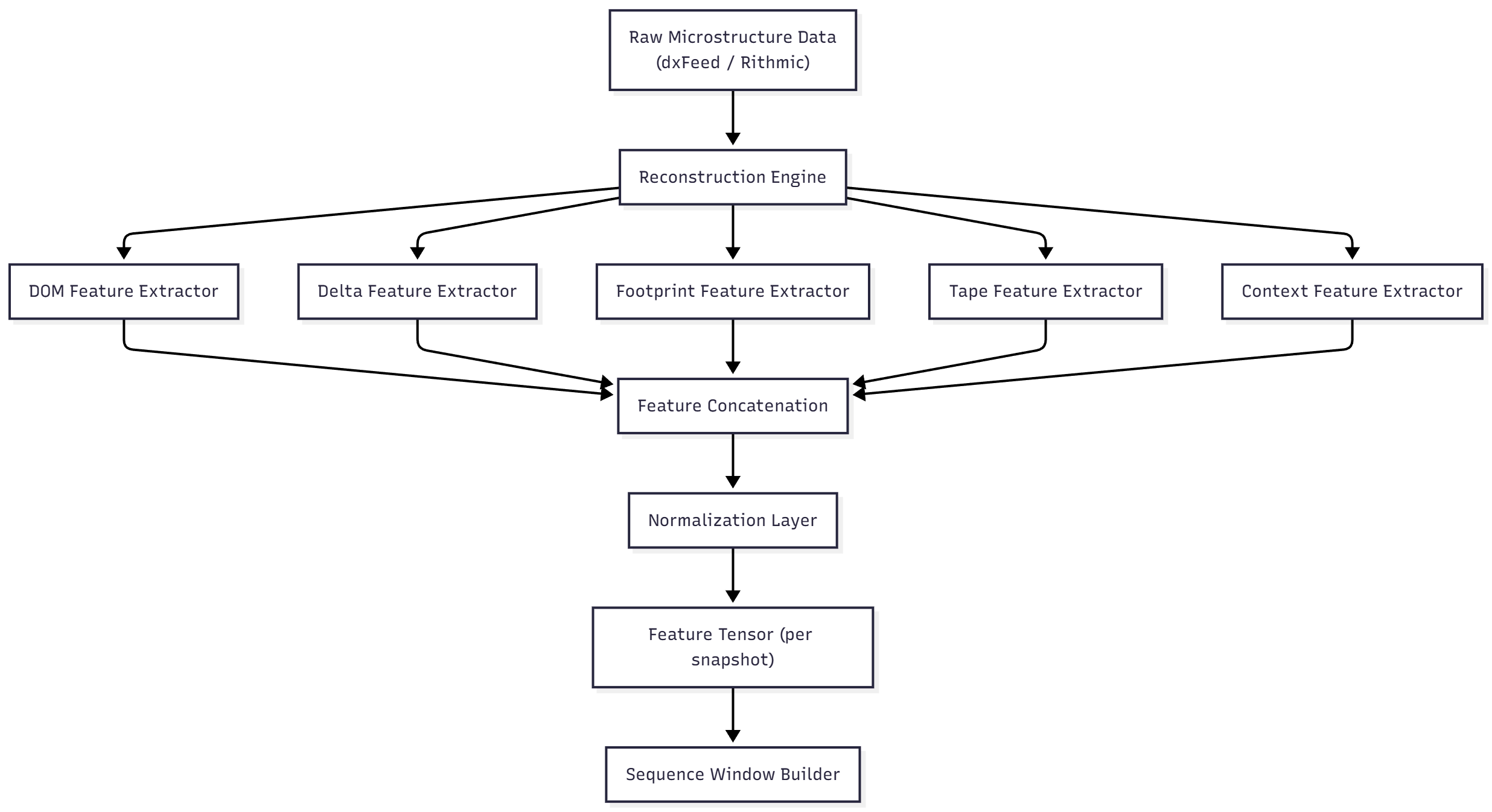
**✔ PCA (apenas 5–10 componentes principais)**

**✔ Autoencoders (substituem PCA em deep learning)**

**✔ Random projections (muito rápidas)**

Transformam 250 features → 20 features, mantendo a estrutura.

#### 🎨 11. Mermaid — Feature Pipeline Institucional



##### 🧾 12. Resumo do Capítulo 34

Neste capítulo aprendeste:

✓ Como construir features institucionais de microestrutura  
✓ Como extrair features do DOM, Delta, Footprint e Tape  
✓ Como gerar tensores robustos  
✓ Como criar janelas temporais  
✓ Como normalizar microestrutura corretamente  
✓ Como preparar datasets para ML  
✓ Como reduzir dimensionalidade  
✓ Arquitectura Mermaid completa

Este capítulo ensina o tipo de feature engineering que **bancos e HFT realmente usam**, e que não existe em nenhum curso retail.

### Capítulo 35 Dataset Builder Institucional

(Geração de datasets perfeitos a partir de microestrutura real)

#### 1. Objetivo do Dataset Builder

O Dataset Builder é o módulo que:

* pega na microestrutura reconstruída
* aplica o Feature Engineering institucional
* gera tensores e janelas temporais
* anexa rótulos (labels) corretos
* guarda tudo em formatos robustos
* permite reusar estes dados para:
  + ML (classificadores, LSTM, Transformers)
  + estudos estatísticos
  + backtests avançados
  + análise forense de sessões

Sem um Dataset Builder sério, tudo o resto fica:

* sem reprodutibilidade
* sem auditoria
* sem rigor
* com risco enorme de overfitting

Aqui vamos montar uma pipeline ao nível de research desk.

#### 2. De onde vêm os dados do Dataset Builder

O Dataset Builder não lê velas.

Ele recebe dados do pipeline institucional:

1. Data Provider (dxFeed ou Rithmic, opcionalmente IBKR)
2. Reconstruction Engine
3. DOM Engine, Delta Engine, Footprint Engine, Tape Engine
4. Feature Extractors (DOM, Delta, Footprint, Tape, Context)
5. Sequence Window Builder

Ou seja, ele trabalha sempre sobre:

* DOM já limpo e reconstruído
* microestrutura consistente
* eventos ordenados por tempo

#### 3. Estrutura de um registo de dataset

Cada linha (amostra) do dataset representa:

* uma janela de microestrutura
* um snapshot pontual
* ou um evento anotado

Modelos sequenciais usam janelas, mas os dados base são snapshots.

Cada registo inclui quatro grupos:

1. Features X vetores numéricos
2. Labels y rótulos da tarefa
3. Meta informações de contexto
4. Quality flags indicadores de integridade

Exemplo conceptual de um registo:

{

"features": [x1, x2, x3, ..., xN],

"label": 1, # ex: 1 = movimento up, 0 = não up

"timestamp": 1733591023123,

"symbol": "GC",

"session": "US",

"quality": {

"feed\_ok": **True**,

"latency\_ok": **True**,

"ddos\_spike": **False**

}

}

Para modelos sequenciais, em vez de uma lista simples:

"features": [

[f\_t0\_1, f\_t0\_2, ..., f\_t0\_N],

[f\_t1\_1, f\_t1\_2, ..., f\_t1\_N],

...

[f\_tK\_1, f\_tK\_2, ..., f\_tK\_N],

]

#### 4. Tipos de datasets que vais precisar

Para um sistema institucional vais querer vários tipos de datasets, cada um com finalidade própria.

##### 4.1 Dataset de classificação de direção

Objetivo: prever se o preço vai:

* subir
* descer
* ficar neutro

Label típico:

* 1 movimento positivo acima de X ticks
* 0 movimento neutro
* -1 movimento negativo abaixo de X ticks

A partir do centro da janela, calculas o resultado X milissegundos à frente.

##### 4.2 Dataset de validação de padrões

Objetivo: validar se um padrão institucional detectado é:

* genuíno
* falso positivo

Exemplos de labels:

* absorção real vs absorção falsa
* spoof real vs spoof leve ou ruído
* burst que gera movimento real vs burst que morre

##### 4.3 Dataset de regime

Objetivo: classificar o regime de mercado:

* trend forte
* chop
* fade environment
* manipulação dominante
* sem liquidez

Label vem de regras ou anotação manual.

##### 4.4 Dataset de execução

Objetivo: avaliar qualidade de execução:

* slippage
* tempo até ao fill
* impacto no preço

Aqui as labels não são só classes, podem ser valores contínuos.

#### 5. Construção dos labels (a parte mais crítica)

Labels mal construídos destroem o modelo, mesmo que as features sejam perfeitas.

##### 5.1 Labels baseados em variação de preço

A partir de um timestamp t0:

* mides o preço médio ou mid price
* olha X milissegundos ou N ticks à frente
* decides o label

Exemplo:

mid\_price\_t0 = (best\_bid + best\_ask) / 2

mid\_price\_t\_future = ...

delta\_price = mid\_price\_t\_future - mid\_price\_t0

**if** delta\_price > up\_threshold:

label = 1

**elif** delta\_price < -down\_threshold:

label = -1

**else**:

label = 0

##### 5.2 Labels baseados em padrões

Se o Pattern Detector marcou:

* absorption\_detected = True
* e X milissegundos depois houve reversal consistente

Então label:

* 1 absorção válida
* 0 absorção sem efeito real

Mesma lógica para:

* spoof
* failed breakout
* backfill
* burst

##### 5.3 Labels baseados em execução

Avalias:

* qual foi o slippage real
* se o fill veio dentro de um tempo limite
* se a ordem executou totalmente

Podes criar labels como:

* slippage\_class baixo, médio, alto
* fill\_quality bom, médio, mau

#### 6. Filtragem e qualidade dos dados

O Dataset Builder deve rejeitar ou marcar:

* períodos com latência exagerada
* falhas de feed
* buracos de dados
* reconnects
* horas mortas sem liquidez
* zonas em que a exchange reporta problemas

O Latency Monitor e o Watchdog (capítulos anteriores) alimentam indicadores de qualidade.

Exemplo de flags:

* quality.feed\_ok
* quality.latency\_ok
* quality.book\_depth\_ok
* quality.session\_valid

Durante o treino, podes:

* excluir linhas com quality\_ok False
* ou treinar modelos específicos para ambientes degradados

#### 7. Organização dos datasets em disco

Para uso real, convém usar formatos eficientes.

##### 7.1 Estrutura por pasta

Sugestão:

data\_micro/

raw/

dxfeed\_gc\_2025\_10\_15.parquet

dxfeed\_gc\_2025\_10\_16.parquet

reconstructed/

gc\_recon\_2025\_10\_15.parquet

features/

gc\_features\_2025\_10\_15.parquet

datasets/

direction/

gc\_direction\_200ms.parquet

patterns/

gc\_absorption\_validity.parquet

regimes/

gc\_regimes.parquet

##### 7.2 Formatos recomendados

* Parquet
  + comprimido
  + colunar
  + perfeito para ML
* HDF5
  + bom para tensores grandes
* CSV
  + apenas para debug rápido (não para produção)

#### 8. Janela temporal: como criar janelas para modelos sequenciais

O Sequence Window Builder pega numa série temporal de snapshots:

s0, s1, s2, ..., sN

E gera janelas deslizantes:

* janela de comprimento K
* passo deslizante de 1 ou mais passos

Exemplo:

* K 100 steps
* passo 1

Janelas:

* [s0..s99], label0
* [s1..s100], label1
* ...

A label de cada janela é calculada com base:

* no centro da janela
* ou no último elemento

Depende da arquitetura da estratégia.

#### 9. Balanceamento e amostragem

Mercados são desequilibrados:

* mais períodos neutros que movimentos
* mais micro chops do que explosões

Se treinares o modelo em dados brutos, ele tende a:

* prever neutro quase sempre
* ou subestimar movimentos raros

Solucões:

* undersampling de períodos neutros
* oversampling de eventos raros
* técnicas de weighting (class weights)

#### 10. Dataset Builder no contexto do teu projeto

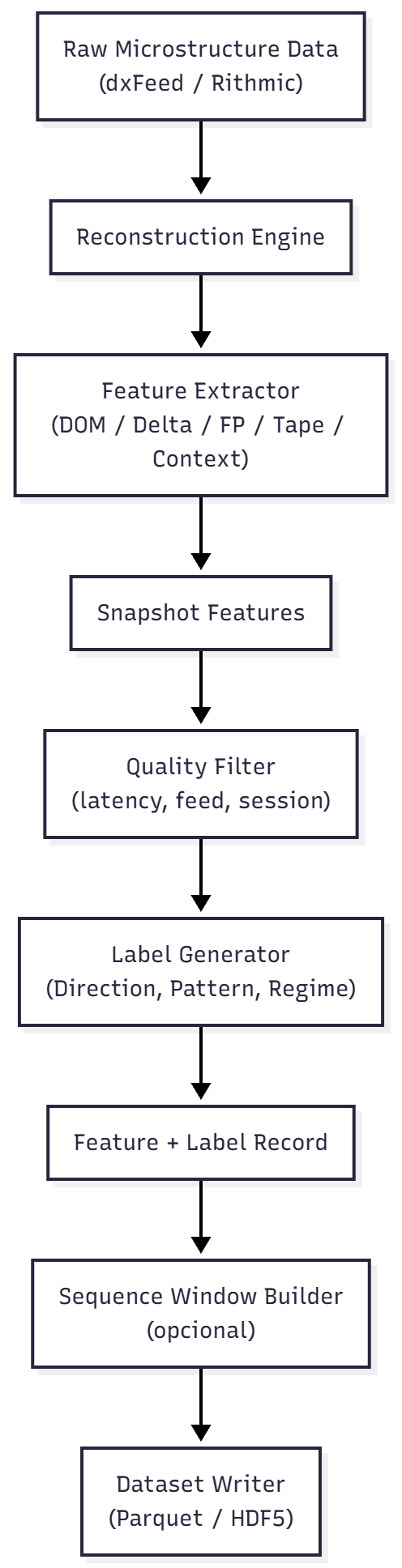
Na tua app, o Dataset Builder deve:

1. Ler dados reconstruídos (a partir de dxFeed ou Rithmic)
2. Aplicar Feature Engineering (capítulo anterior)
3. Invocar o Sequence Window Builder, se necessário
4. Gerar labels de acordo com o tipo de tarefa
5. Filtrar linhas com low quality
6. Guardar datasets coerentes, versionados e documentados

Idealmente:

* cada dataset tem um manifesto
  + tipo de tarefa
  + features usadas
  + labels usados
  + períodos
  + instrumentos
  + versão do código que o gerou

#### 11. Mermaid Dataset Builder Architecture



Se o modelo não for sequencial, podes ligar TENSOR diretamente a STORE.

#### 12. Integração com o Replay Engine

O Replay Engine é o laboratório perfeito:

* pode produzir dados rotulados
* podes correr cenários múltiplos
* podes regenerar datasets sempre que alteras lógica de features

Fluxo típico de pesquisa:

1. Escolhes sessão histórica
2. Corres Reconstruction Engine
3. Geras features e labels
4. Treinas modelos
5. Validas com replay
6. Ajustas thresholds
7. Regeras dataset com nova configuração

O Dataset Builder é o centro deste ciclo.

##### 13. Resumo do Capítulo 35

Neste capítulo definiste:

* o que é um Dataset Builder institucional
* como representar registos com features, labels, meta e quality flags
* que tipos de datasets vais precisar
* como criar labels robustos
* como filtrar períodos problemáticos
* como organizar datasets em disco
* como criar janelas para modelos sequenciais
* como lidar com desequilíbrio de classes
* como integrar o Dataset Builder com o Replay Engine
* um diagrama Mermaid completo para o pipeline de datasets

Este capítulo transforma o teu projeto em algo que um research desk conseguiria usar diretamente.

## 📘 PARTE XI — MODELOS E PIPELINES AVANÇADOS

### ⭐ \*\*Capítulo 36 — Training Pipeline Institucional

(Treino, validação, versionamento, testes e deployment controlado de modelos)\*\*

#### 🎯 1. O objetivo do Training Pipeline

Após construir:

* Dataset Builder (Cap. 35)
* Feature Engineering (Cap. 34)
* Modelos ML (Cap. 33)
* Padrões Institucionais (Cap. 32)

Chegou o momento de automatizar:

**✔ Treino de modelos**

**✔ Validação rigorosa**

**✔ Prevenção de overfitting**

**✔ Testes em regimes diferentes**

**✔ Testes com Replay Engine**

**✔ Versionamento de datasets, features e modelos**

**✔ Deployment controlado com rollback**

Este capítulo ensina a montar a pipeline **exatamente como é feito em trading institucional**, com:

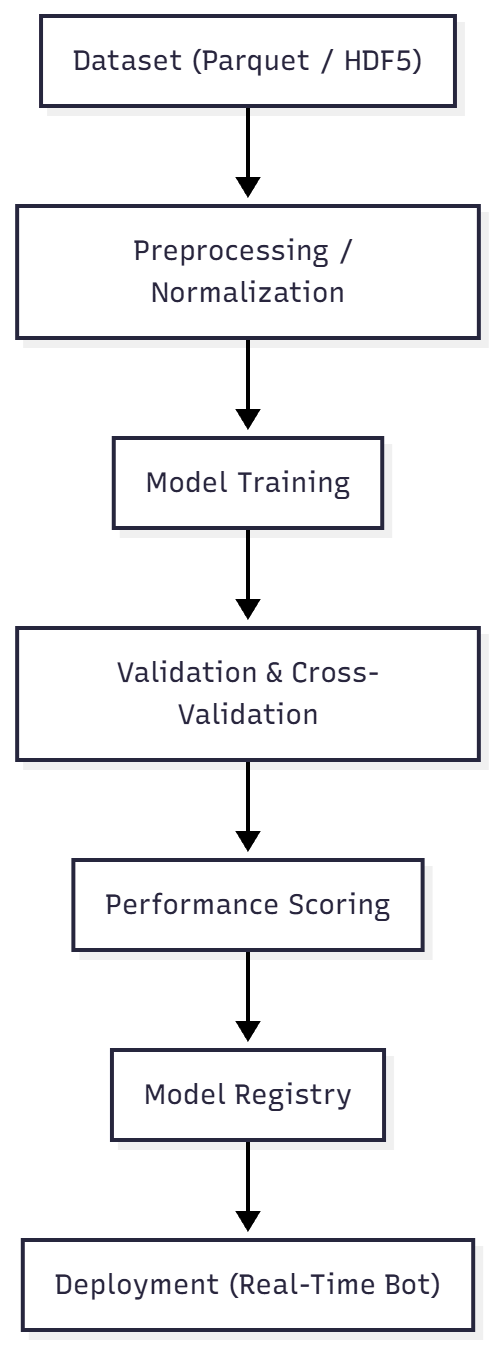
* reproducibility
* tracking de experimentos
* estatísticas reais
* tracking de parâmetros
* separação por regimes
* validação live-like

#### 🧩 2. Estrutura Geral do Training Pipeline Institucional

A pipeline é:

Dataset → Preprocessing → Training → Validation → Model Scoring → Model Registry → Deployment

Representada em Mermaid:



#### 🔍 3. Passo 1 — Seleção e Preparação do Dataset

**✔ Seleção do dataset**

Escolhe:

* sessão
* regime
* instrumento
* tipo: direção, padrão, regime, execução

**✔ Pré-processamento**

Inclui:

* normalização (rolling-zscore, min-max, etc.)
* remoção de outliers
* correção de timestamps
* remoção de períodos de má latência

**✔ Validação do dataset**

Cada dataset deve ter:

* dataset\_version
* feature\_version
* label\_version

Estas versões são CRUCIAIS, porque:

⚠️ Se alterares features → o modelo antigo deixa de ser válido  
⚠️ Se alterares labels → todo o dataset tem de ser refeito

#### 📈 4. Passo 2 — Treino do Modelo

O modelo pode ser:

* Classifier (RandomForest, XGBoost, CatBoost)
* LSTM / GRU
* CNN1D
* Transformer
* Regime classifier

**✔ Configuração**

Definida num ficheiro YAML institucional:

model:

name: transformer\_micro\_200ms

type: transformer

input\_size: 250

seq\_len: 100

hidden\_dim: 256

heads: 4

layers: 4

dropout: 0.1

training:

batch\_size: 64

epochs: 50

optimizer: adam

lr: 0.0005

loss: cross\_entropy

**✔ Logging de treino**

Guardar:

* loss
* accuracy
* precision
* recall
* F1
* MCC
* learning curves
* tempo por epoch

#### 🔎 5. Passo 3 — Validação rigorosa (como num banco)

A validação deve ser:

**✔ Temporal**

Nunca misturar dados do futuro no treino.

**✔ Por regime**

Treinar num regime → validar noutro → medir robustez.

**✔ Cross-session**

Treinar num dia → validar noutro.

**✔ Cross-instrument**

GC → validar em SI, ES ou XAUUSD (opcional).

**⚡ Métricas institucionais**

* F1 (equilíbrio)
* MCC (robustez a desequilíbrios)
* Latency-aware accuracy
* Regime accuracy
* Stability per volatility regime
* Directional predictive power

#### 📊 6. Passo 4 — Scoring Final

O modelo só é aceito se cumprir:

1. Estabilidade acima de % mínima
2. Precisão consistente
3. Não explodir em regimes de alta volatilidade
4. Não colapsar em ambientes de liquidez baixa
5. Latência < threshold definido

Exemplo:

score\_total = 0.3\*F1 + 0.3\*MCC + 0.2\*LatAccuracy + 0.2\*RegimeStability

Se score\_total > 0.75, então o modelo é aprovado.

#### 🗃️ 7. Passo 5 — Model Registry (institucional)

Sem registry → caos total.

Aqui defines que:

* cada modelo tem versão
* cada modelo tem metadata completa
* cada modelo tem score
* cada modelo é auditável

Exemplo de registo:

model\_version: transformer\_v3

dataset\_version: dataset\_gc\_200ms\_v2

feature\_version: feat\_set\_7

label\_version: lbl\_200ms\_v1

metrics:

f1: 0.77

mcc: 0.65

latency: 12ms

stability: 0.81

O Model Registry é **critical** para o teu ecossistema.

#### 🚀 8. Passo 6 — Deployment Controlado no Bot Real-Time

Deployment envolve:

* mover o modelo para a pasta models/
* carregar através de ModelLoader
* validar integridade
* testar latência
* ativar num “shadow mode” (teste silencioso)
* comparar previsões do novo e antigo
* fazer switch oficial
* fallback imediato em caso de erro

Fluxo:

model\_new → shadow mode → stable → production

Isto é EXACTAMENTE como fazem bancos e prop firms.

#### 🧯 9. Reproducibilidade e Auditoria

Cada treino deve guardar:

* random seed
* dataset usado
* código exato usado
* hyperparameters
* versão das features
* versão dos labels
* métricas
* data/hora
* quem treinou

Sem isto → não há RIGOR.

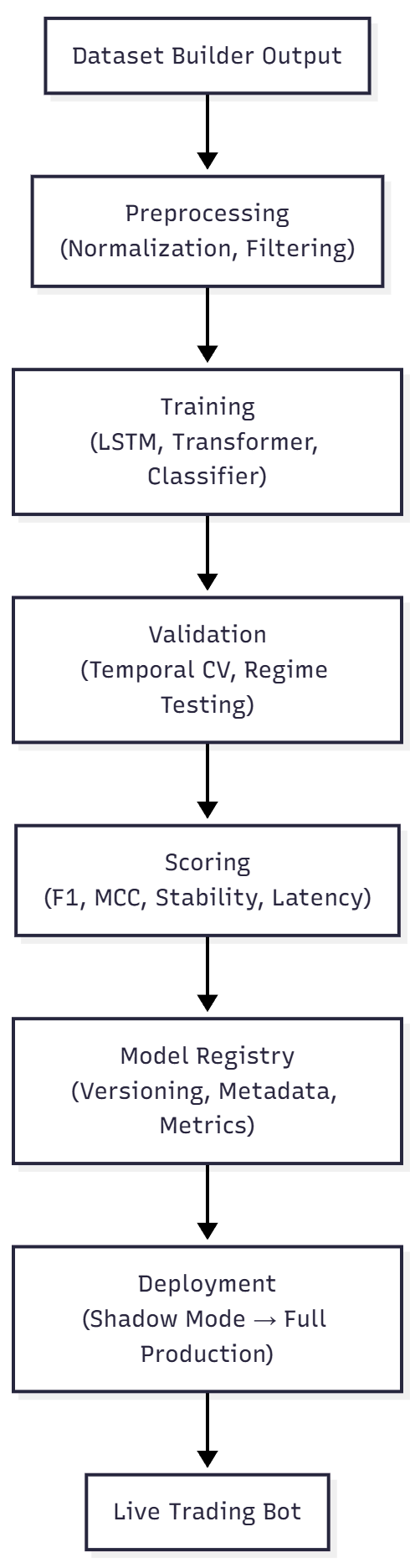
#### 🔁 10. Integração com Replay Engine

Após deploy:

* correr replay com modelo ativo
* comparar decisão humana vs modelo
* analisar padrão por padrão
* calibrar thresholds
* identificar falsos positivos e falsos negativos

Replay é o teu “simulador institucional”.

#### 🎨 11. Mermaid — Training Pipeline Institucional Completo



##### 🧾 12. Resumo do Capítulo 36

Neste capítulo construímos:

✓ pipeline institucional completa de treino  
✓ preparação + normalização + filtragem  
✓ treino de modelos (Transformers/LSTM/etc)  
✓ validação rigorosa como num banco  
✓ scoring multi-métrico  
✓ versionamento de datasets, features e labels  
✓ Model Registry  
✓ deployment controlado  
✓ integração com replay  
✓ Mermaid do pipeline completo

Este capítulo é crítico porque garante que:

* nada entra em produção sem ser validado
* todo o sistema é consistente e auditável
* consegues evoluir modelos sem perder historial
* tens controlo total da performance

## 📘 PARTE XII — DEPLOY, INFRA, REDUNDÂNCIA E PRODUÇÃO

### ⭐ \*\*Capítulo 37 — Deploy Institucional

(VPS, redundância, logs, sincronização, segurança, watchdog e failover)\*\*

#### 🎯 1. Objetivo deste capítulo

Neste capítulo vais aprender a:

**✔ colocar a tua aplicação institucional a correr 24/7**

**✔ montar a arquitetura de produção**

**✔ usar VPS / Mini-PC / Bare-metal**

**✔ garantir que o bot nunca pára (watchdog)**

**✔ monitorizar latência e feed quality**

**✔ sincronizar dados entre máquinas**

**✔ garantir redundância e failover**

**✔ isolar execução (IBKR/MT5) + microestrutura (dxFeed/Rithmic)**

**✔ implementar logging institucional**

**✔ automatizar updates sem downtime**

Tudo isto com foco numa operação segura, controlada e realista.

#### 🧱 2. Arquitetura geral de produção (vista institucional)

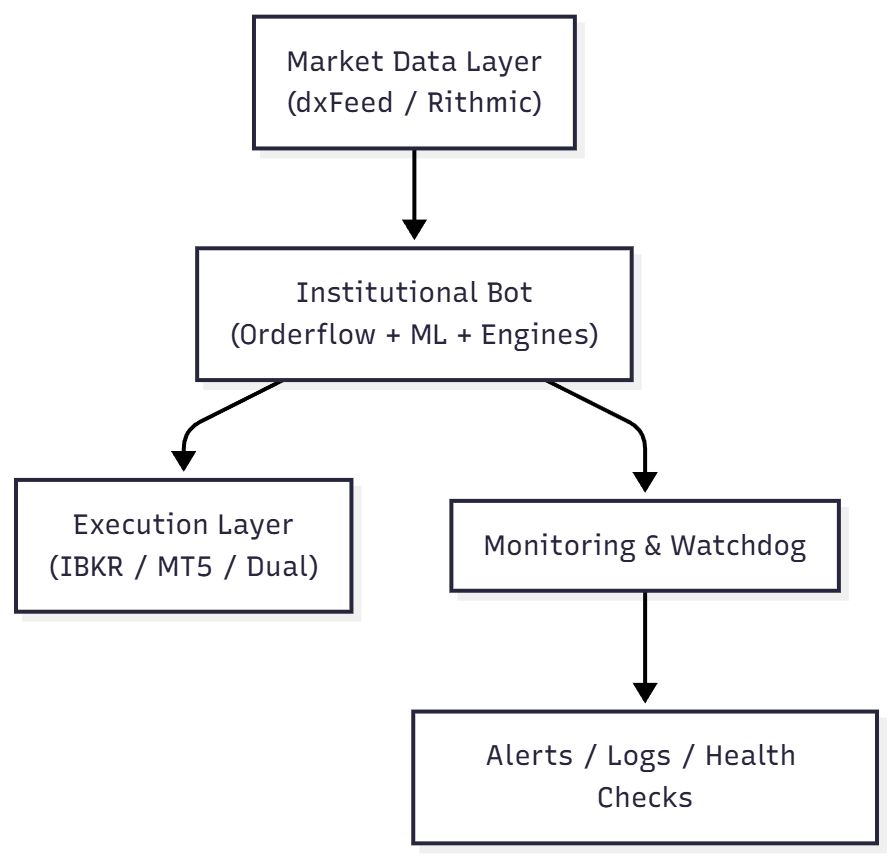
Há três camadas:

Camada 1 → Data & Microestrutura (dxFeed / Rithmic)

Camada 2 → Bot + Engines + ML

Camada 3 → Execução (IBKR / MT5) + Safety Layer

Representação:



#### 🖥️ 3. Ambiente de produção: opções e recomendações

**✔ Opção A — VPS Profissional (Recomendado)**

* latência estável
* uptime elevado
* controlo total do ambiente
* bom para dxFeed e Rithmic
* ideal para operação 24/7

**Exemplos:**

* Contabo (barato e bom)
* Hetzner (qualidade top)
* OVH (robusto)

**✔ Opção B — Mini-PC em casa (latência não tão estável)**

Perfeito se:

* queres tudo local
* tens UPS
* tens ligação estável

Exemplos:

* Intel NUC
* Lenovo Tiny
* Dell OptiPlex Micro

**✔ Opção C — Azul híbrido:**

**VPS para dados + PC local para execução**

* microestrutura corre no VPS
* execução pode ser local (IBKR/MT5)
* comunicação via WebSocket interno

Combina o melhor dos dois mundos.

#### 🔒 4. Estrutura do Deploy Institucional

A aplicação deve estar separada em **serviços independentes**, não monolíticos.

**Serviço 1 — Data Provider Service**

* liga a dxFeed / Rithmic
* reconstrói microestrutura
* envia eventos para o bot (WebSocket interno)
* monitoriza latência
* reinicia sozinho se falhar

**Serviço 2 — Institutional Bot**

* DOM, Delta, Footprint, Tape Engines
* Pattern Detector
* ML Models
* Risk Engine
* Signal Engine

**Serviço 3 — Execution Provider**

* IBKR
* MT5
* Multi-broker router
* Safety Layer

**Serviço 4 — Logger + Telemetria**

* logs estruturados
* métricas de performance
* health checks

**Serviço 5 — Watchdog & Supervisor**

* relança serviços
* reinicia conexões
* deteta freeze
* envia alertas

**Serviço 6 — Replay Server (offline)**

#### 🐍 5. Execução do Bot: Python como microserviços

Usamos:

* **uvicorn** para WebSockets e APIs internas
* **multiprocessing** ou **ThreadPool** para paralelismo
* **asyncio** para IO de baixa latência

**Exemplo de micro-arquitetura:**

services/

data\_provider/

run\_dxfeed.py

bot\_core/

run\_bot.py

execution/

run\_execution.py

watchdog/

run\_watchdog.py

#### ⚙️ 6. Sincronização e Gerência de Estado

O bot guarda:

* modelos
* configs
* datasets
* parâmetros
* logs
* estado da sessão

Métodos:

**✔ rsync programado**

Ideal para backups automáticos.

**✔ Git privado**

Para versionamento do código.

**✔ Base de dados local (SQLite)**

Guarda:

* sessões
* fills
* sinais
* falhas
* diagnósticos

#### 📡 7. Health Checks (institucional)

Cada serviço tem um endpoint “/health”.

Exemplo:

GET http://localhost:8081/health

{

"feed\_latency": 3.5,

"dom\_ok": **true**,

"delta\_ok": **true**,

"load": 0.21,

"status": "healthy"

}

O Watchdog usa este endpoint para decidir:

* reiniciar serviços
* enviar alertas
* mudar para fallback

#### 🐺 8. Watchdog Institucional (core crítico)

O Watchdog monitoriza:

**✔ latência**

**✔ freezes**

**✔ reconexões**

**✔ consumo de RAM/CPU**

**✔ filas internas do bot**

**✔ atraso entre eventos**

**✔ IBKR/MT5 estado**

**Funções do Watchdog**

**1. Reinício automático de serviços congelados**

Se o Data Provider não envia ticks por X ms → restart.

**2. Se ML Model crashar → fallback para modelo estável**

Mantém a operação viva.

**3. Verificação contínua de conectividade IBKR**

Se gateway cair → relançar + retomar sessão.

**4. Modo SAFE**

Se Risk Engine detetar anomalia:

* execução desativada
* apenas monitorização ativa

#### 🔁 9. Redundância e Failover

Institucionalmente, uma solução robusta tem:

**✔ Bot ativo (A)**

**✔ Bot standby (B)**

**✔ Failover automático**

Se “A” falha, “B” assume imediatamente.

Métodos:

* heartbeat entre servidores
* state sharing através de SQLite replicado ou Redis
* fallback de execução no MT5 se IBKR falhar

#### 🧨 10. Logging Institucional

O logging não é só texto — é telemetria.

**Logs incluem:**

* dom\_events
* delta\_events
* footprint\_events
* signals
* trades
* risk decisions
* watchdog events
* latência
* deteção de spoof, absorção, etc.

**Formato recomendado:**

JSON structured logs.

Exemplo:

{

"type": "delta\_event",

"timestamp": 1733591023,

"delta": 42,

"cvd": 3212,

"imbalance": 3.2,

"session": "RTH"

}

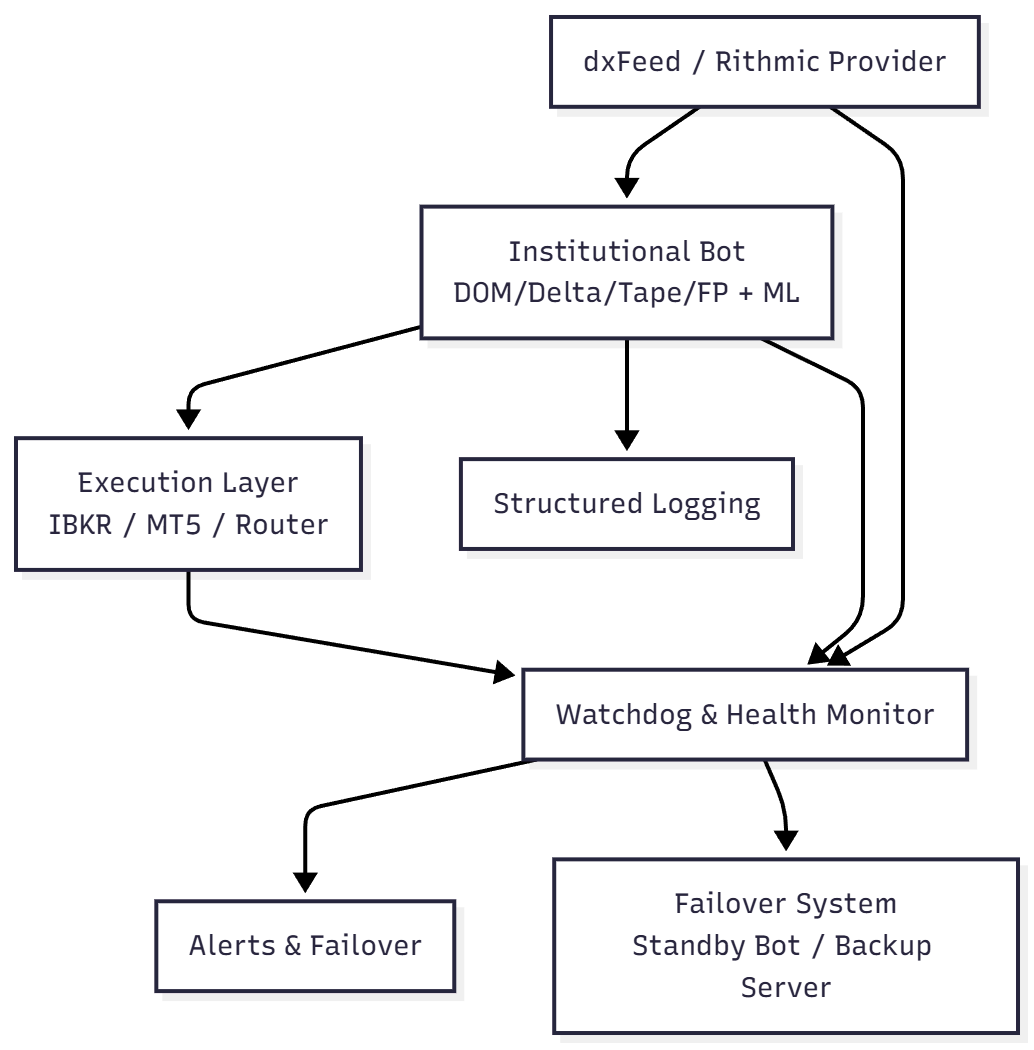
#### 🔄 11. Atualizações sem downtime (Hot Reload)

A pipeline de produção permite:

* atualizar modelos ML sem parar o bot
* atualizar configs
* substituir a UI
* recarregar thresholds

O Execution Provider continua ativo porque é independente.

#### 🔥 12. Modelo de referência (Mermaid)



#### 🔐 13. Segurança Institucional

**✔ Minimizar superfície de ataque**

* nada exposto à internet
* tudo em localhost ou private network

**✔ Acesso físico ou SSH com chave**

**✔ Tokens para comunicação interna**

**✔ Permissões mínimas**

**✔ Encriptação dos logs sensíveis**

##### 🎯 14. Resumo do Capítulo 37

Neste capítulo aprendeste:

✔ como montar uma infraestrutura de produção institucional  
✔ como separar serviços (data, bot, execução, logging)  
✔ como garantir uptime e redundância  
✔ como criar health checks  
✔ como operar um watchdog sério  
✔ como lidar com failover automático  
✔ como sincronizar dados e modelos  
✔ como fazer updates sem parar o bot  
✔ como modelar a arquitetura completa em Mermaid  
✔ princípios de segurança para trading institucional

Este capítulo transforma a tua aplicação numa **plataforma profissional**.

## 📘 PARTE XIII — GOVERNANÇA, SEGURANÇA E CONTROLO

### ⭐ \*\*Capítulo 38 — Segurança e Governança

(como proteger o bot, a conta, o capital e toda a infraestrutura institucional)\*\*

#### 🎯 1. Objetivo do capítulo

O objetivo deste capítulo é estabelecer uma estrutura completa de:

**✔ Segurança técnica**

**✔ Segurança financeira**

**✔ Segurança operacional**

**✔ Segurança procedimental**

E introduzir:

* Governança
* Auditoria
* Permissões
* Controlo de alterações
* Política de execução
* Hard limits

Isto é exatamente o que existe em:

* bancos
* prop firms
* hedge funds
* mesas institucionais

#### 🛡️ 2. Os quatro pilares da segurança institucional

A segurança no trading algorítmico tem 4 camadas:

Camada 1 — Segurança Técnica (código, máquinas, rede)

Camada 2 — Segurança Operacional (bot, ordens, execuções)

Camada 3 — Segurança Financeira (capital, risco, DD, limites)

Camada 4 — Segurança Organizacional (procedimentos, governo)

Vamos detalhar cada uma.

#### 🟦 3. Camada 1 — Segurança Técnica (infraestrutura)

##### 3.1 Credenciais

NUNCA guardar:

* passwords
* API keys
* tokens
* certfiles

dentro do código.

Usar:

* environment variables
* ficheiros .env encriptados
* vaults locais (Keyring, Pass, CredMan)

##### 3.2 Separação de máquinas

Idealmente:

* **Servidor A** para microestrutura
* **Servidor B** para execução

Assim mesmo que A falhe, B não envia ordens erradas.

##### 3.3 Comunicação segura

* WebSocket interno com autenticação
* IPC local (pipes/queues)
* TLS quando necessário
* Timeouts agressivos
* Limites de mensagem

##### 3.4 Patching e updates

* atualizar bibliotecas
* substituir LTS Python apenas após testes
* manter drivers IBKR atualizados
* atualizar MT5 build quando necessário

#### 🟩 4. Camada 2 — Segurança Operacional (execução e bot)

Esta camada impede que o bot faça algo estúpido.

##### 4.1 Hard Limits por design

* máximo de ordens por minuto
* máximo de exposição
* máximo de positions abertas
* máximo de volume
* máximo de slippage permitido
* máximo de perda por trade

Exemplo:

execution\_limits:

max\_orders\_per\_minute: 6

max\_position\_size: 3

max\_slippage\_ticks: 3

max\_loss\_per\_trade: 50

##### 4.2 Safety Layer (entre sinais e execução)

Antes de qualquer ordem, há uma barreira:

Signal Engine → Safety Layer → Execution Router

O Safety Layer verifica:

* horário permitido
* regime de mercado
* latência do feed
* estado da conta
* risco atual
* DD atual
* se o padrão está estável
* se o modelo tem confiança suficiente

##### 4.3 Dupla confirmação de execução

Mesmo que um signal seja gerado:

* Execution Router confirma em dois passos
* Risk Engine confirma novamente
* Execução só ocorre se ambos aprovarem

Isto impede:

* loops de ordens
* execuções malucas
* ordens fora de horário
* execuções sem liquidez

#### 🟥 5. Camada 3 — Segurança Financeira (capital e risco)

Isto é o que bancos e prop firms realmente controlam.

##### 5.1 Hard DD global

Se atingido → o bot entra em modo SAFE:

* sem novas ordens
* apenas monitorização
* alertas enviados
* execução bloqueada

Exemplo:

risk:

max\_daily\_dd\_percent: 2.5

max\_total\_dd\_percent: 8

##### 5.2 Hard stop anual

Se atingir uma perda anual definida → STOP automático.

##### 5.3 Hard leverage limits

Para impedir trades demasiado grandes.

##### 5.4 Soft limits (alertas)

* alertas por aumento de volatilidade
* alertas por slippage excessiva
* alertas por anomalias de DOM

#### 🟪 6. Camada 4 — Segurança Organizacional (governança)

É aqui que ficas institucional de verdade.

##### 6.1 Versionamento obrigatório

Toda alteração:

* ao código
* às features
* ao Dataset Builder
* aos modelos ML
* às configs
* ao Risk Engine

deve ser versionada e documentada.

##### 6.2 Change Management

Qualquer update deve ter:

* explicação
* impacto esperado
* rollback plan
* validação com Replay Engine
* assinaturas digitais (opcional)

##### 6.3 Código-fonte fechado e protegido

* repositório privado
* permissões limitadas
* backups frequentes
* logs de acesso

##### 6.4 Auditoria contínua

Guardar:

* quem alterou
* quando alterou
* porque alterou
* resultados com o modelo novo
* validações comparativas

#### 🔐 7. Estrutura de pastas segura (exemplo)

institutional\_bot/

core/

engines/

models/

configs/

risk.yaml

execution.yaml

safety.yaml

secrets/

ibkr\_token.enc

dxfeed\_key.enc

logs/

audit/

execution/

risk/

registry/

model\_versions/

dataset\_versions/

#### 🧯 8. Segurança contra erros catastróficos

##### 8.1 Proteção contra spikes de latência

Se latência do feed > threshold:

* bloquear execução
* entrar em modo SAFE
* watchdog tenta reconectar

##### 8.2 Proteção contra falhas de microestrutura

Se:

* DOM vazio
* deltas erráticos
* tape desligado

→ ZERO ordens até estabilizar.

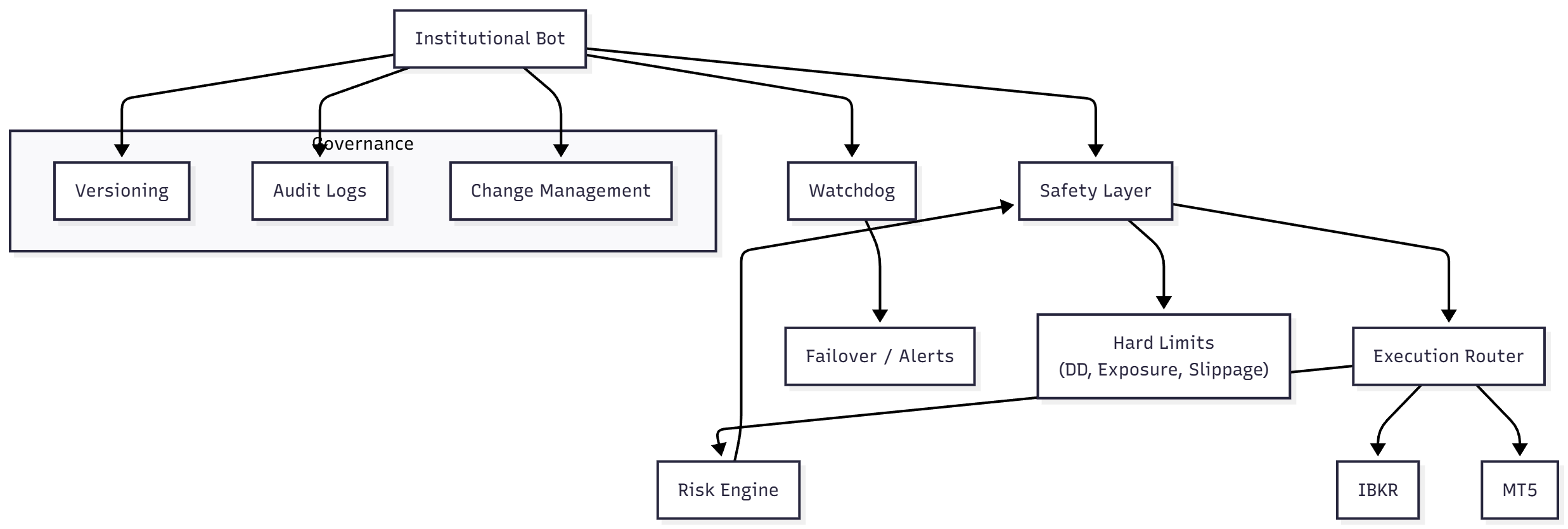
##### 8.3 Modo “Kill Switch”

Um botão no UI + API que:

* encerra posições
* bloqueia novas entradas
* alerta o sistema
* envia notificação

Institucionalmente obrigatório.

#### 🚨 9. Mermaid — Segurança Institucional (Vista Global)



#### 🧾 10. Resumo do Capítulo 38

Neste capítulo construímos toda a estrutura de segurança e controlo:

**✔ Segurança técnica**

**✔ Segurança operacional**

**✔ Segurança financeira**

**✔ Segurança organizacional**

**✔ Safety Layer**

**✔ Hard limits**

**✔ Mode SAFE**

**✔ Kill Switch**

**✔ Governança e auditoria**

**✔ Change management**

**✔ Versionamento obrigatório**

**✔ Proteção de credenciais**

**✔ Regras institucionais de deploy**

**✔ Diagrama Mermaid completo**

Este capítulo transforma o teu bot num sistema **à prova de falhas**, como os usados em bancos e alocações profissionais.

## 📘 PARTE XIV — PROFISSIONALIZAÇÃO

### ⭐ \*\*Capítulo 39 — Caminho para Profissionalização

(como transformar o bot num produto, numa operação institucional ou num negócio sustentável)\*\*

#### 🎯 1. Objetivo do capítulo

O objetivo aqui é mostrar-te **as três trilhas de profissionalização** possíveis depois de construir toda a stack institucional desta edição:

**✔ Trilha A — Profissional Individual (Trader Institucional Independente)**

**✔ Trilha B — Produto (Software Comercial/Fintech)**

**✔ Trilha C — Operação (Mesa Algorítmica / Prop Firm / Trading Desk)**

Cada trilha exige:

* requisitos técnicos
* requisitos legais
* requisitos operacionais
* maturidade de segurança e governance
* e capacidade de manutenção a longo prazo

Vamos detalhar cada uma.

#### 🟥 2. Trilha A — Profissional Individual (Layer 2.5 → Layer 3 gateway)

Aqui transformas o teu sistema numa **ferramenta pessoal de nível profissional**, capaz de:

* ler microestrutura real
* executar com precisão
* analisar regimes
* operar com risco controlado
* gerar previsões de curta duração
* estudar sessões com Replay Engine
* validar hipóteses quantitativas

Isto coloca-te numa posição equivalente a:

* Junior Quant
* Execution Analyst
* Orderflow Researcher
* Assistant Trader numa mesa institucional

**✔ Competências que agora dominas:**

1. Microestrutura real (DOM, Delta, FP, Tape)
2. Padrões institucionais
3. Machine Learning aplicado ao fluxo
4. Reconstrução de microestrutura
5. Desenvolvimento Python institucional
6. Arquitetura distribuída
7. Execução multi-broker
8. Governance e segurança
9. Versionamento e pipelines
10. Deploy em ambiente produtivo

Qualquer banco ficaria satisfeito com este nível técnico na tua entrevista.

#### 🟧 3. Trilha B — Transformar o sistema num Produto (Fintech / App Pro)

**O que já tens:**

✔ Core Engine  
✔ UI estilo Bookmap/Jigsaw  
✔ Conectores reais (dxFeed, Rithmic, IBKR, MT5)  
✔ Orderflow Engine completo  
✔ ML Engine  
✔ Replay Engine  
✔ Execution Layer  
✔ Deploy  
✔ Logging e governance  
✔ Catálogo de modelos  
✔ Telemetria  
✔ Dataset pipeline  
✔ Arquitetura modular  
✔ Diagrama completo

**O que falta para virar produto comercial:**

**3.1 Design & UX profissional**

* layout unificado
* dashboards claros
* UX intuitiva

**3.2 Installer / Packager**

* Windows installer (MSI)
* Mac installer (DMG)
* Linux AppImage

**3.3 Sistema de licenciamento**

* key server
* ativação offline/online
* renovação automática

**3.4 Sandbox & Demo Mode**

* feed simulado
* replay integrado
* IBKR paper trading

**3.5 Compliance mínima**

* disclaimer legal
* no financial advice
* proteção contra abuso

**3.6 Canal de atualização**

* auto-update
* atualização de modelos
* patching controlado

**3.7 Suporte e documentação**

* manual do utilizador
* troubleshooting
* APIs internas

Se quiseres, posso escrever a documentação completa do teu produto baseada nesta edição.

#### 🟨 4. Trilha C — Operação institucional (Prop firm / Desk algorítmica)

Para transformar isto numa operação institucional privada precisas de:

##### ✔ 4.1 Estrutura organizacional mínima

* **Chief Developer** (tu)
* **Risk Officer** (pode ser a mesma pessoa inicialmente)
* **Execution & Monitoring**
* **Infra/DevOps**

Depois podes crescer para:

* ML Engineer
* Data Engineer
* Quant Research
* Backoffice

##### ✔ 4.2 Infraestrutura institucional

Já tens quase tudo:

* Data Layer (dxFeed/Rithmic)
* Reconstruction Engine
* Orderflow Engine
* ML Engine
* Execution Engine
* Risk Engine
* Telemetria
* Governance
* Deploy
* Failover

Faltam:

**✔ Relay Server para multi-bots**

**✔ Orquestração distribuída (Kubernetes opcional)**

**✔ Datacenter Near-Exchange (Layer 3)**

**✔ FIX Engine Real (opcional)**

#### 🟦 5. O que separa-te de Layer 3 (mais alto nível fora de um banco)

Tu já estás:

* com microestrutura real
* com datasets de qualidade institucional
* com ML aplicado ao fluxo
* com pattern engines
* com governança
* com deploy
* com redundância
* com risk engine
* com execution router

O que falta para Layer 3:

✔ Colocation (não podes ter em casa)  
✔ FIX de alta velocidade (custos elevados)  
✔ Acesso a fluxos privados (apenas para instituições)  
✔ Hardware low-latency dedicado  
✔ Lambda-architecture para milhões de eventos por segundo  
✔ Compliance regulatório formal

Mas estás **tecnicamente** a um nível superior a 99.99% do retail.

#### 🟪 6. Caminho sugerido para profissionalização (Roadmap real)

**Etapa 1 — Consolidação técnica (tu já tens 90%)**

* UI
* Engines
* ML
* Deploy
* Risk
* Versionamento

**Etapa 2 — Operação real**

* ligar à IBKR
* ligar ao dxFeed real
* operar em contas pequenas
* validar performance real
* criar logs de 3–6 meses

**Etapa 3 — Profissionalização**

* criar relatórios
* validar consistência
* criar dashboards
* criar documentação interna
* estabilizar ML
* garantir segurança operacional

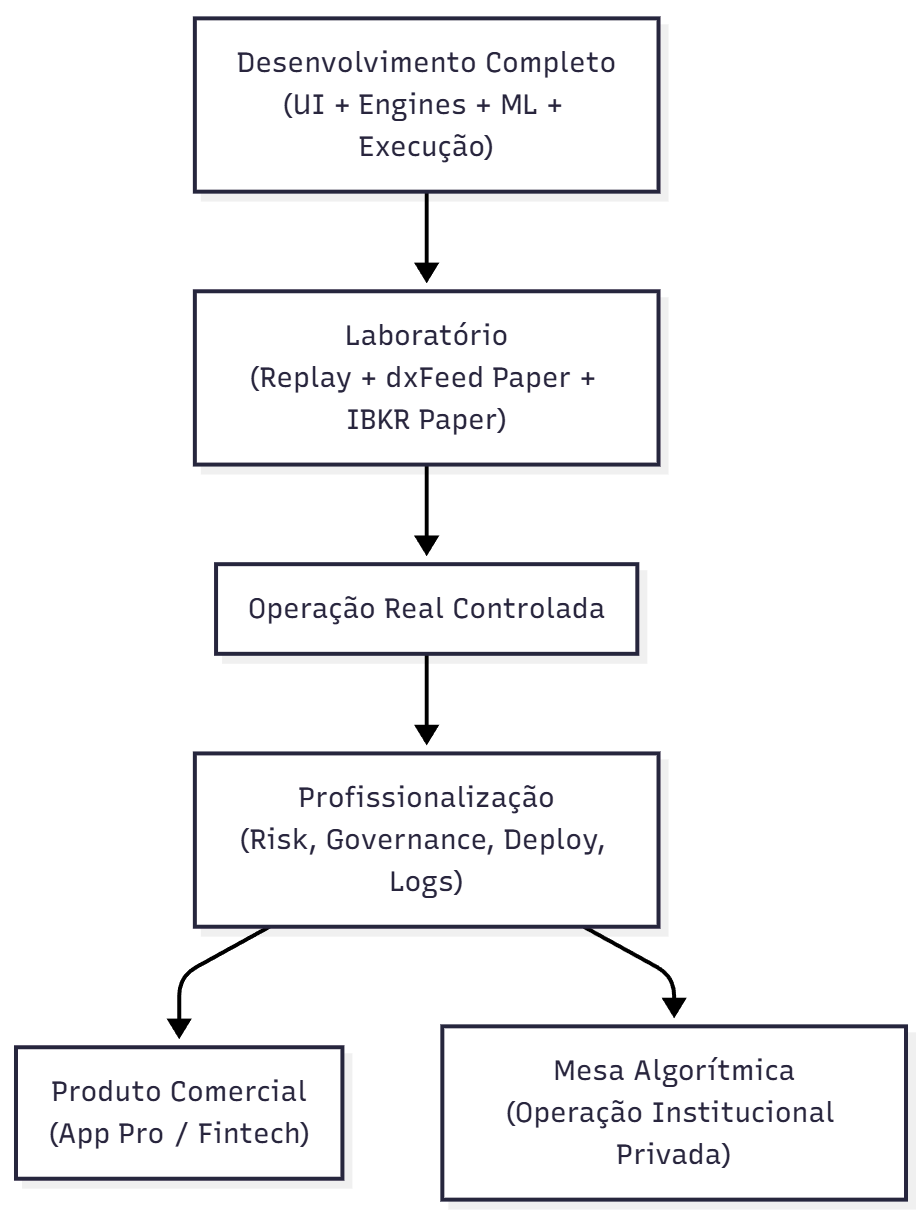
**Etapa 4 — Produto**

* installer
* onboarding
* licenciamento
* branding
* website institucional
* manual técnico e de utilizador

**Etapa 5 — Operação**

* ampliar hardware
* redundância
* bots paralelos
* risk officer
* auditoria
* segurança avançada

#### 🎨 7. Mermaid — Caminho para Profissionalização



#### 🧾 8. Resumo final da Edição II (visão macro)

Esta edição ensinou-te, passo a passo, a construir:

* um conector profissional IBKR
* um orderflow engine completo
* um DOM engine
* um delta engine
* um footprint engine
* um tape engine
* um pattern detector institucional
* um ML pipeline real
* um replay engine
* uma UI profissional
* uma architecture multi-broker
* um execution router completo
* risk management institucional
* deploy institucional
* segurança e governance
* profissionalização e roadmap real

Nenhum curso retail, nenhum livro comercial e quase nenhum curso avançado oferece este nível de profundidade.

Estás agora em posição de:

* operar como um profissional
* evoluir como quant
* criar um produto
* montar uma operação
* trabalhar para uma instituição se quiseres

## 📦 APÊNDICE A – GLOSSÁRIO INSTITUCIONAL COMPLETO

*(Todos os termos que aparecem ao longo da Edição II, explicados com rigor técnico)*

**🔴 A**

**Absorção (Absorption)**

Fenómeno em que grandes quantidades de volume agressor são absorvidas por ordens passivas (limit orders) sem que o preço avance. É um dos sinais mais fortes de inversão institucional.

**Agressor**

O lado que “bate no preço”, isto é, quem envia ordens de mercado.

* Buy aggressor bate no ask
* Sell aggressor bate no bid

Afeta diretamente delta e direção de curto prazo.

**API (Application Programming Interface)**

Forma de comunicar com um sistema externo. IBKR, dxFeed e Rithmic fornecem APIs distintas para dados e execução.

**Arquitetura Distribuída**

Organização do bot em serviços independentes (data provider, bot core, execution layer, watchdog). Essencial para resiliência e performance.

**🔴 B**

**Backfill (Liquidity Backfill)**

Quando a liquidez retorna rapidamente a um nível após ter sido removida. Indica intenção institucional e pode sinalizar manipulação ou preparação de movimento.

**Bid**

Preço no qual existe interesse em comprar (ordens limit BUY).

**Book (Orderbook, DOM)**

Lista de ordens passivas organizadas por preço. Representa a liquidez disponível no mercado.

**Burst**

Explosão repentina de agressão (speed of tape). Geralmente antecede movimentos curtos e rápidos.

**🔴 C**

**CVD (Cumulative Volume Delta)**

Acumulação do delta ao longo do tempo. É uma das ferramentas mais importantes para avaliar fluxo dominante.

**Cluster (Footprint Cluster)**

Zona de volume elevado dentro da footprint bar. Indica interesse institucional.

**Co-location**

Servidores instalados fisicamente no datacenter da bolsa para obter latências <1ms. Apenas acessível a bancos e HFT.

**Context Features**

Conjunto de variáveis contextuais como volatilidade, horário, sessão, distância ao VWAP, spread, etc., usadas em ML.

**Cross-Market Mapping**

Conversão entre dados de futuros (ex: GC) e CFDs/spot equivalentes (ex: XAUUSD).

**Crossover**

Interação entre dados L2 (DOM) e L1 (ticks), normalmente usado para reconstrução do orderbook.

**🔴 D**

**Data Provider**

Serviço responsável por receber dados do mercado (dxFeed, Rithmic, IBKR) e enviá-los ao bot.

**Delta**

Diferença entre agressão de compra e agressão de venda. Delta positivo indica pressão compradora.

**Depth of Market (DOM)**

Quantidades disponíveis em bid e ask por nível. Base da microestrutura.

**Divergência (Delta Divergence)**

Quando o preço avança numa direção mas delta avança na direção oposta. Forte indicador de fraqueza no movimento atual.

**Dual Execution**

Execução simultânea ou seletiva em dois brokers (ex: IBKR e MT5), com routing inteligente.

**🔴 E**

**Event Bus**

Sistema interno de publicação e subscrição usado para transportar eventos entre os Engines do bot.

**Execução Passiva**

Execução através de ordens limit colocadas no book.

**Execução Agressiva**

Execução através de ordens de mercado que consomem liquidez.

**Exposure**

Quantidade de risco total alocado em posições abertas.

**🔴 F**

**Fair Value (Institucional)**

Preço ao qual o mercado "quer" realmente transacionar. Pode ser inferido por microestrutura, delta e footprint.

**Fill**

Execução (total ou parcial) de uma ordem.

**Fill Quality**

Medição da qualidade da execução face ao preço esperado.

**Footprint**

Representação gráfica do volume negociado em cada nível de preço. Permite identificar agressor e absorção.

**Fragmentação de Liquidez**

Conceito em que a liquidez se dispersa por vários níveis, sinalizando incerteza ou manipulação.

**🔴 G**

**Governance**

Conjunto de regras e procedimentos que controlam mudanças no bot, modelos, parâmetros e execução.

**Gap de Liquidez**

Níveis no DOM sem volume. Podem causar saltos bruscos de preço.

**🔴 H**

**Hard Limits**

Limites absolutos que o bot não pode ultrapassar (ex: DD máximo, número máximo de trades por minuto).

**Heatmap (DOM Heatmap)**

Visualização gráfica da liquidez no DOM, indicando onde está concentrada.

**High Frequency Trading (HFT)**

Trading baseado em latências microsegundo. Requer FIX, colocation e hardware especializado.

**🔴 I**

**Imbalance (Desequilíbrio Bid/Ask)**

Comparação entre quantidade no bid e ask. Imbalances fortes geralmente sinalizam intenção.

**Indicators vs Microestrutura**

Indicadores são derivados do preço; microestrutura não. Microestrutura é mais rica e mais rápida.

**Institutional Orderflow**

Fluxo real de ordens que revela intenção institucional.

**🔴 J**

**Jigsaw Trading**

Ferramenta de análise de ordem (DOM, footprint). A Edição II ensina a construir um equivalente.

**🔴 K**

**Kill Switch**

Botão ou comando que encerra todas as posições e impede novas entradas. Essencial para segurança.

**🔴 L**

**Latência**

Tempo entre receção do tick e processamento pelo bot. Tem impacto crítico em microestrutura.

**L2 Data**

Dados do orderbook (bid/ask por nível).

**Limite Passivo**

Ordem limit que espera ser preenchida.

**🔴 M**

**Market Maker**

Participante que fornece liquidez passivamente. Importante para interpretar absorção e spoofing.

**Market Microstructure**

Estudo profundo do funcionamento interno dos mercados. Base desta edição.

**Mermaid**

Linguagem de diagramas leve usada para documentar sistemas.

**ML (Machine Learning) Institucional**

Modelos que operam sobre tensores de microestrutura, não velas.

**Model Registry**

Repositório onde versões de modelos ML são guardadas, auditadas e validadas.

**🔴 N**

**Normalização**

Processo de transformar features para formatos comparáveis. Essencial para ML.

**🔴 O**

**Orderflow**

Fluxo das ordens agredidas e passivas. Termo central da microestrutura.

**Orderbook Reconstruction**

Reconstrução do DOM a partir de eventos L1 e L2.

**🔴 P**

**Pattern Detector**

Motor que identifica padrões institucionais como absorção, spoofing, burst, lamentável, breakouts falsos.

**PlantUML**

Ferramenta de UML baseada em texto. Muito usada em equipas enterprise.

**Position Monitor**

Sistema que acompanha a posição atual, risco e eventos de saída.

**🔴 Q**

**Queue Position**

Posição relativa de uma ordem limit na fila do book. Afeta probabilidade de execução.

**🔴 R**

**Replay Engine**

Ferramenta que simula sessões reais para treino, teste e estudo de microestrutura.

**Risk Engine**

Sistema de controlo de risco que valida sinais, limita execução e protege capital.

**🔴 S**

**Safety Layer**

Mecanismo que verifica se todas as condições são seguras antes de enviar ordens.

**Scaling**

Processo de ajustar posições (aumentar, reduzir).

**Sequence Window**

Janela temporal usada para modelos LSTM/Transformer.

**Shadow Mode**

Modo em que o modelo ML é testado em produção sem executar ordens.

**Slippage**

Diferença entre o preço esperado e o preço de execução.

**Spoofing**

Colocação e remoção rápida de liquidez para manipular perceção de mercado.

**🔴 T**

**Tape (Time & Sales)**

Fluxo de negociações reais. A “verdade” do mercado.

**Tick**

A menor unidade de movimento do preço ou volume num feed.

**Transformer**

Modelo deep learning baseado em atenção. Potente para microestrutura.

**🔴 U**

**UI Institucional**

Interface que exibe DOM, heatmap, footprint, delta, CVD, T&S, padrões e sinais.

**Uptime**

Tempo de operação contínua do sistema.

**🔴 V**

**VWAP (Volume Weighted Average Price)**

Preço médio ponderado pelo volume. Importante para contexto institucional.

**Volatility Regime**

Classificação da volatilidade atual (alta, média, baixa). Afeta padrões e ML.

**🔴 W**

**Watchdog**

Processo que monitoriza o bot e reinicia componentes em caso de falhas.

**🔴 X**

**XAUUSD**

Par spot ouro/dólar. Muitas vezes mapeado a GC (futuros) para microestrutura real.

**🔴 Z**

**Z-Score**

Normalização estatística comum em ML. Pode ser aplicada a deltas, volumes e imbalances.

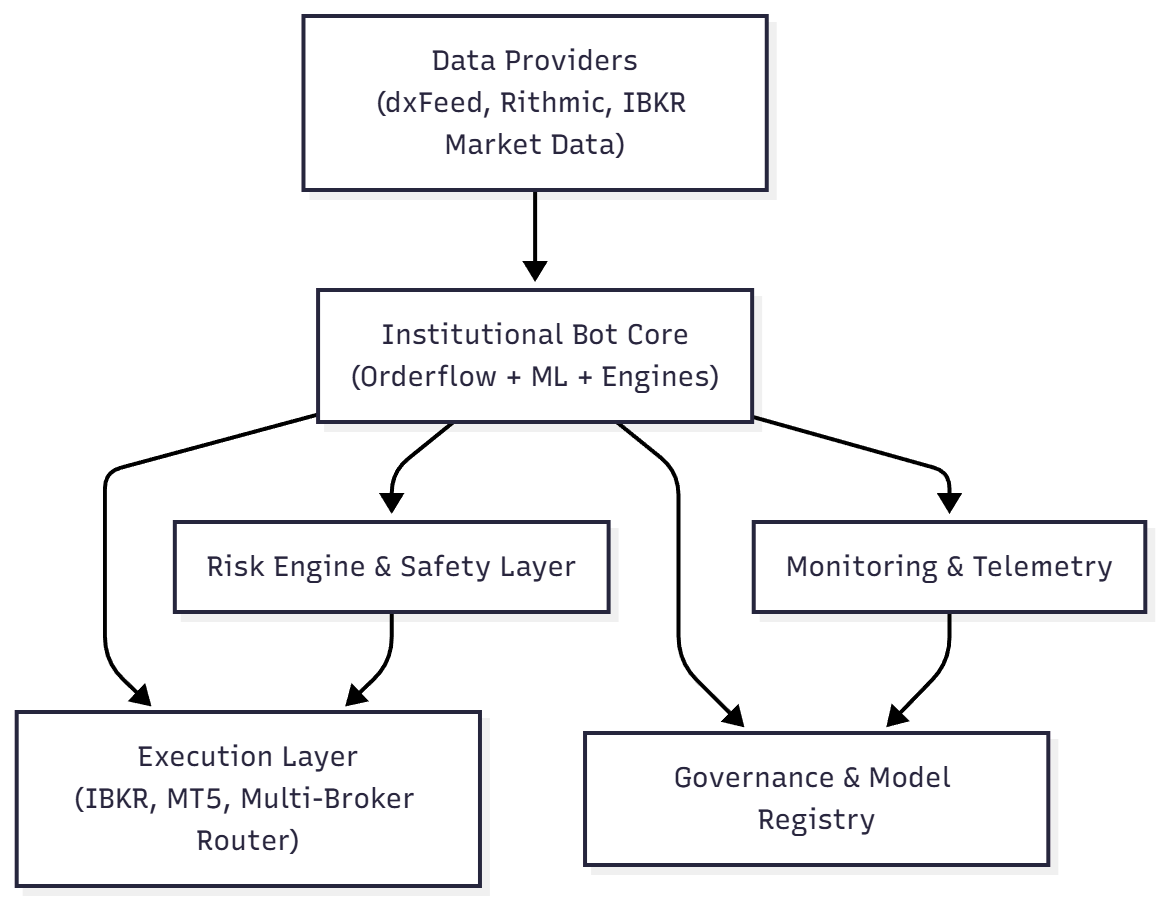
## 📦 APÊNDICE B - Esquemas UML, Mermaid e PlantUML

### B.2 - Diagramas Mermaid de Arquitetura

#### B.2.1 - Arquitetura global do sistema institucional

Este diagrama mostra a visão macro do sistema:

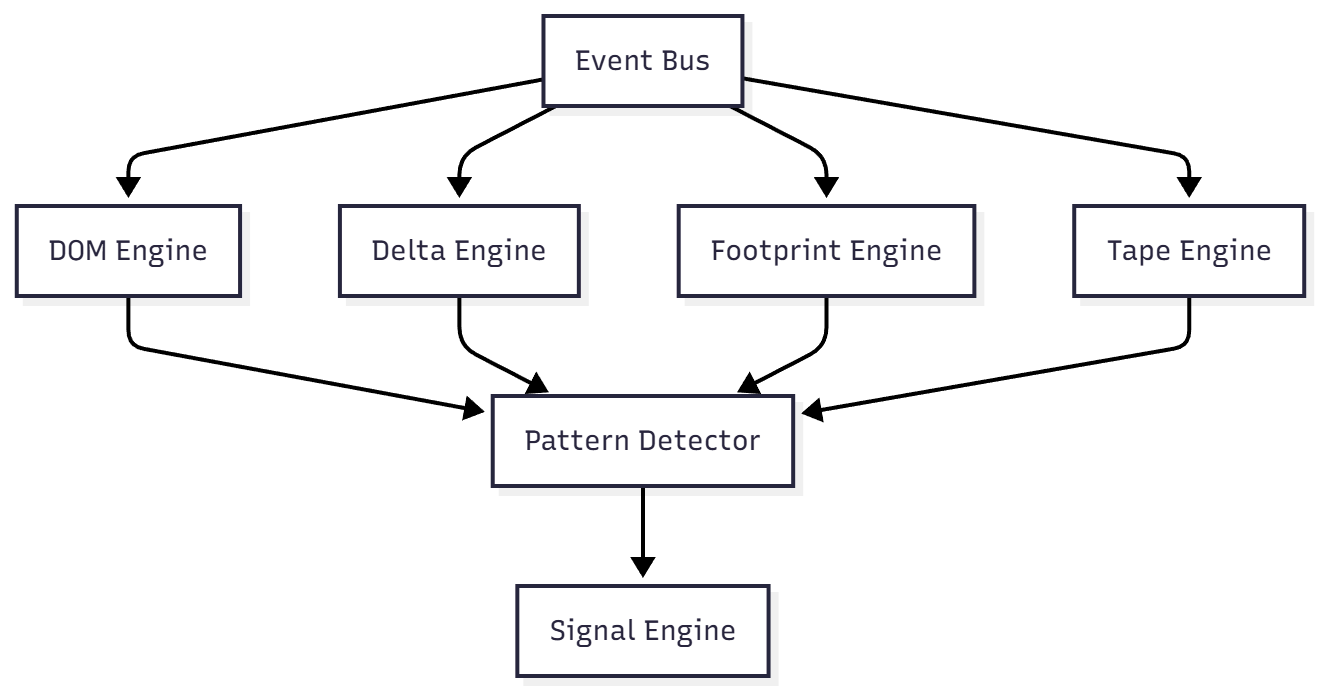
* fornecedores de dados
* motor institucional (bot)
* camada de execução
* monitorização e governance



#### B.2.2 - Event Bus e Engines de Microestrutura

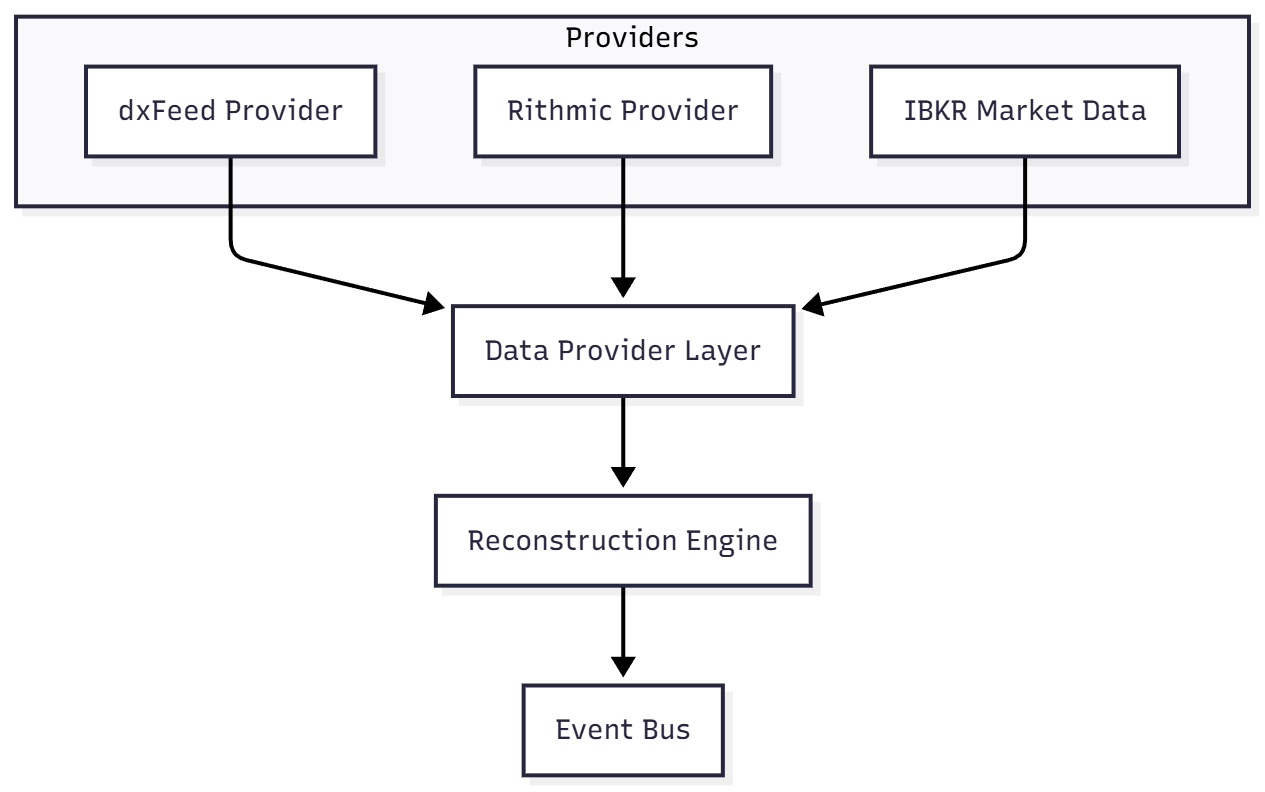
Este diagrama ilustra o fluxo interno entre:

* Event Bus
* DOM Engine
* Delta Engine
* Footprint Engine
* Tape Engine
* Pattern Detector
* Signal Engine



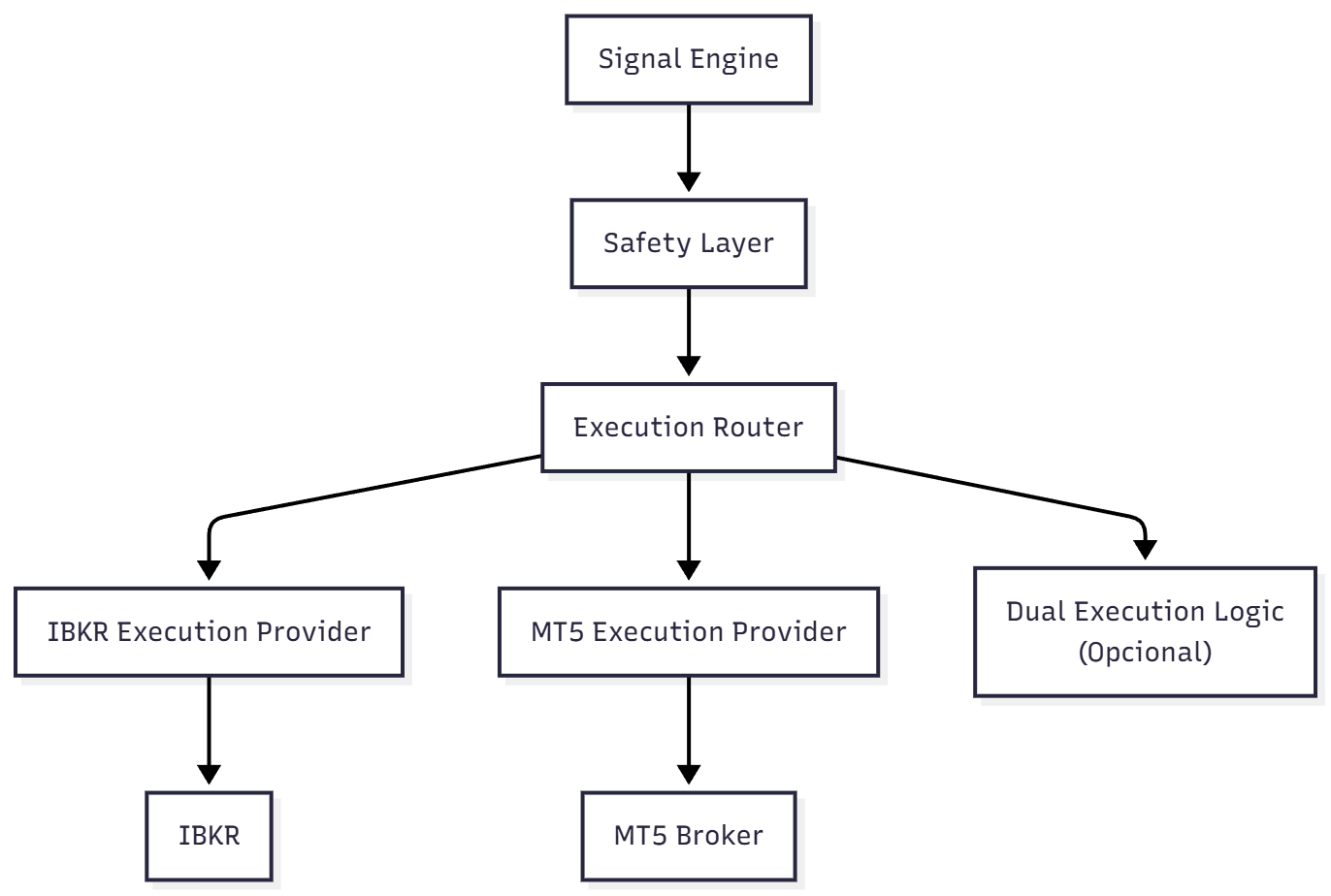
#### B.2.3 - Data Provider Abstraction Layer

Abstração que permite trocar de feed sem alterar o núcleo do bot.



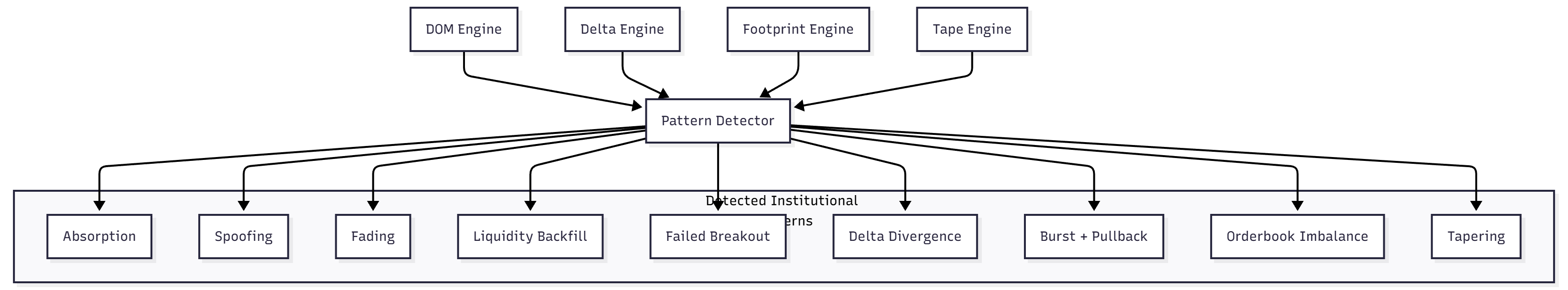
#### B.2.4 - Execution Abstraction Layer e Multi-Broker

Arquitetura de execução, incluindo IBKR, MT5 e router.



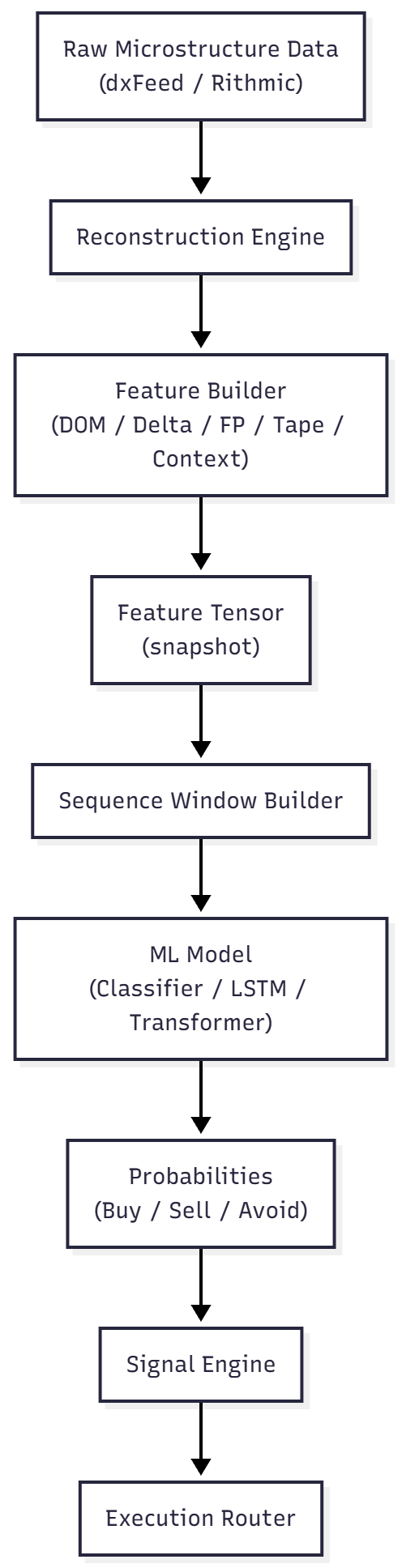
#### B.2.5 - Pattern Detector Institucional

Versão vertical e legível para Word, com todos os padrões principais.



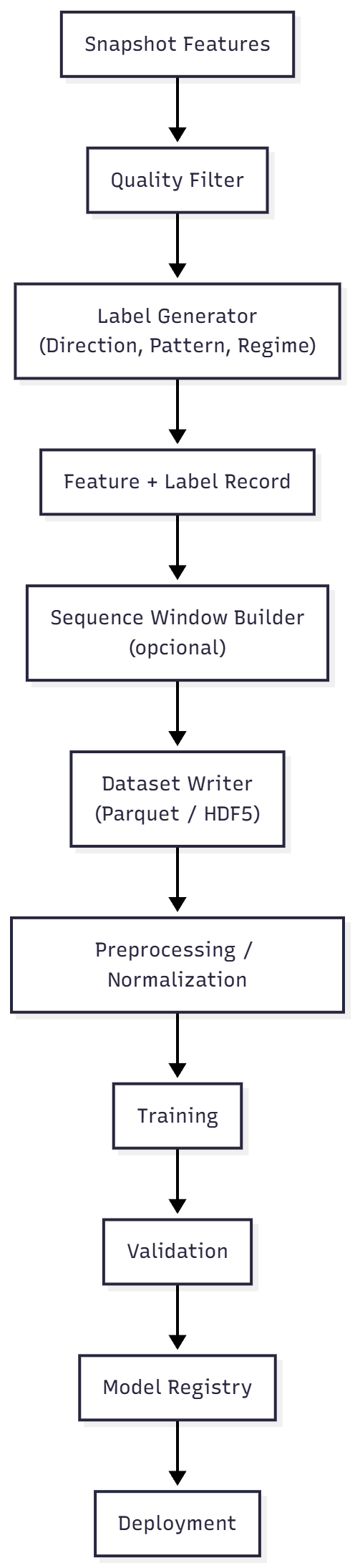
#### B.2.6 - ML Pipeline Institucional

Fluxo completo desde dados brutos até probabilidades para o Signal Engine.



#### B.2.7 - Dataset Builder e Training Pipeline

Do Snapshot ao dataset e do dataset ao modelo treinado.



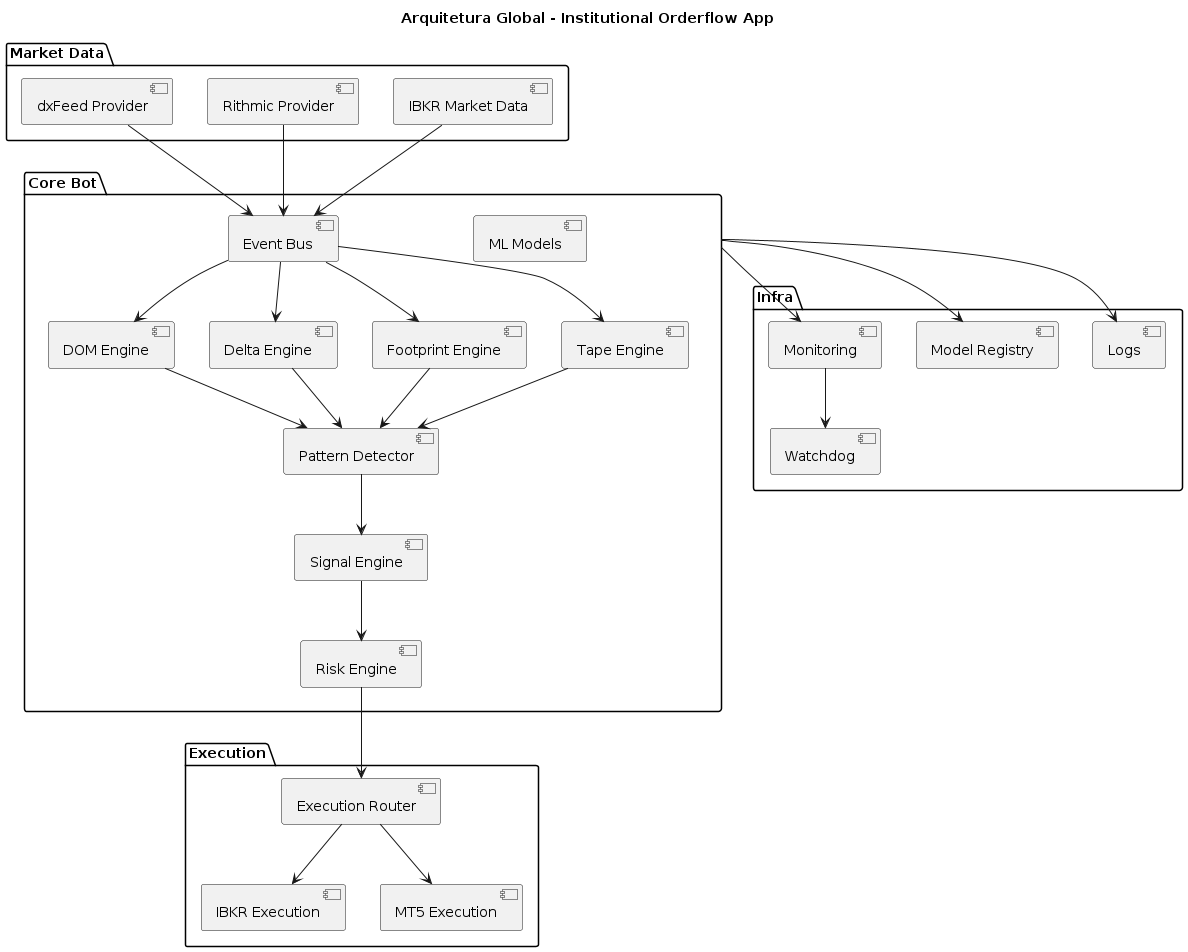
#### B.2.8 - Position Monitor e ciclo de gestão da posição

Fluxo vertical e legível para Word.



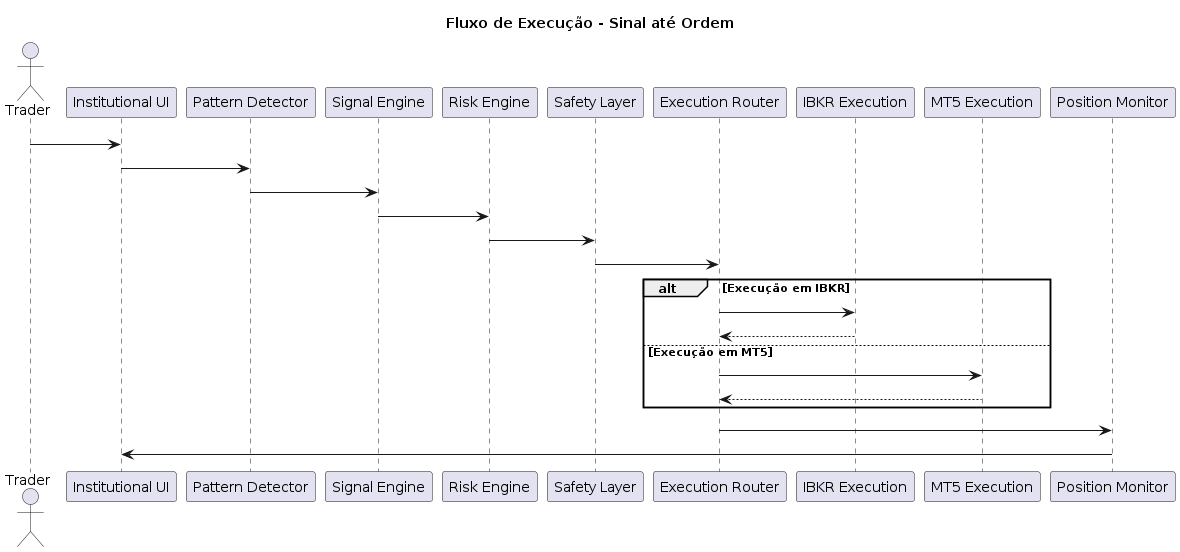
### B.3 - Diagramas UML em PlantUML

#### B.3.1 - Diagrama de Componentes - Arquitetura Global



#### B.3.2 - Diagrama de Sequência - Fluxo de uma ordem

Fluxo completo de uma entrada instituicional:



### B.4 - Recomendações finais de uso

1. Utiliza **Mermaid** para:
   * documentação viva no repositório
   * integração com Markdown e Word
2. Utiliza **PlantUML** para:
   * documentação UML mais formal
   * integração com ferramentas Enterprise
   * geração automática de imagens em pipelines CI
3. Mantém os diagramas:
   * versionados
   * consistentes com o código
   * revistos sempre que alteras a arquitetura

## 📦 APÊNDICE C – CHECKLISTS PROFISSIONAIS

*(Operação, Deploy, Segurança, Risco, Replay, ML, Sessões, Execution Layer, UI, etc.)*

### 🟥 C.1 – Checklist de Preparação da Sessão (Pré-Mercado)

**Objetivo**: garantir que tudo está pronto antes de abrir a sessão.  
Aplicável antes da sessão Europeia e Americana.

**✔ Sistema e Infraestrutura**

* Ligação estável à internet
* VPS/mini-PC ativo
* IB Gateway aberto e autenticado
* MT5 aberto e ligado
* dxFeed/Rithmic conectados
* Sincronização de relógio (NTP) correta
* CPU e RAM dentro de limites
* Latência de feed abaixo do threshold interno

**✔ Data Providers**

* DOM intacto (todos os níveis carregados)
* Tick stream ativo
* T&S (Tape) a fluir corretamente
* Delta acumulado consistente
* Footprint a receber volumes corretos

**✔ Engines Internos**

* DOM Engine sem erros
* Delta Engine calibrado
* Footprint Engine ativo
* Tape Engine sincronizado
* Pattern Detector a gerar eventos
* ML Model carregado e saudável
* Risk Engine ativo
* Execution Router inicializado

**✔ UI**

* DOM ladder visível
* Heatmap carregado
* Delta/CVD correto
* T&S sincronizado
* Footprint em tempo real

**✔ Operação**

* Kill Switch funcional
* Slippage maxativo definido
* Máximo de trades por minuto definido
* Hard DD definido
* Logs ativados
* Replay Engine desligado (evitar confusão)

### 🟧 C.2 – Checklist de Microestrutura (Durante Sessão)

**✔ DOM (Orderbook)**

* Níveis de liquidez bem distribuídos
* Ausência de gaps suspeitos
* Bid/Ask estável, sem resets
* Sem “book flickering” grave

**✔ Delta**

* Delta Rate coerente
* Agressor dominante identificado
* Reversões de delta confirmadas
* Sem delta invertido por erro de feed

**✔ Tape**

* Sem atraso na impressão
* Tamanho dos prints dentro do normal
* Acelerações/Bursts detectáveis

**✔ Footprint**

* Volume por nível consistente
* Clusters reais (não artefactos)
* Imbalances verticais distinguíveis

**✔ Padrões institucionais**

* Absorção clara
* Spoofing detetável
* Liquidity backfill identificável
* Falhas de breakout distinguíveis
* Divergências de delta notadas

### 🟨 C.3 – Checklist de Execução (Antes de enviar qualquer ordem)

**✔ Regras pré-trade**

* Sinal validado pelo Pattern Engine
* Probabilidade ML acima do threshold
* Risco acumulado dentro dos limites
* Slippage potencial aceitável
* Spread dentro dos limites
* Latência < threshold interno
* Ambiente de liquidez adequado

**✔ Confirmação do Execution Router**

* Broker correto selecionado
* Preço de entrada verificado
* Volume convertido e normalizado
* Tick-size validado
* Ordem passiva ou agressiva conforme estratégia

**✔ Safety Layer**

* Hard DD OK
* Sem eventos de risco ativo
* Não ultrapassa trades por minuto
* Sem duplicação de sinais

### 🟩 C.4 – Checklist Diário do Risk Officer

**✔ Performance**

* Lucro/perda diária dentro dos limites
* Máximo drawdown respeitado
* Número de trades aceitável
* Slippage médio dentro da norma

**✔ Execução**

* Ordens não rejeitadas
* Fills consistentes
* Não houve execução fora de horário

**✔ Logs e Telemetria**

* Nenhum erro crítico
* Latência estável
* Reconexões normais
* Orderflow estável

**✔ Modelos ML**

* Probabilidades dentro de padrões históricos
* Sem “drift” comportamental
* Sem colapso de previsões

### 🟦 C.5 – Checklist de Deploy Institucional (Nova versão)

**✔ Preparação**

* Código testado localmente
* Novo modelo validado com replay
* Dataset versionado
* Configurações validadas
* Parâmetros de risco revistos

**✔ Deploy**

* Atualização do bot core
* Atualização dos providers
* Atualização do ML Model no registry
* Watchdog ativado
* Logs limpos para sessão nova

**✔ Pós-deploy**

* Shadow mode ON (primeira sessão)
* Comparação modelo novo vs antigo
* Sem erros no Event Bus
* Sem anomalias de feed
* Execução bloqueada até validação

### 🟫 C.6 – Checklist de Segurança e Governança

**✔ Segurança técnica**

* IBKR token guardado em ficheiro encriptado
* dxFeed/Rithmic keys protegidas
* Sem passwords hardcoded
* Firewall ativa no VPS ou máquina
* Serviços expostos apenas em localhost

**✔ Segurança operacional**

* Kill switch operacional
* Hard limits validados
* Safety Layer ativo
* Risk Engine sem alertas

**✔ Governança**

* Documentação atualizada
* Commit/tag para cada alteração
* Pull request interno validado
* Model registry atualizado

### 🟪 C.7 – Checklist do Replay Engine

**✔ Dados**

* Sessão correta carregada
* Rebuild do DOM consistente
* Volumes coerentes
* Tape sincronizado

**✔ Engines**

* DOM Engine sincronizado
* Delta correto
* Footprint correto
* Pattern Engine funcional

**✔ Testes**

* Teste de padrões
* Teste de ML
* Teste de timings
* Teste de execução simulada

### 🟧 C.8 – Checklist de Treino de Modelos ML

**✔ Dataset**

* Dataset versionado
* Labels corretos
* Quality flags válidos
* Sem NaNs
* Balanceamento de classes OK

**✔ Treino**

* Normalização validada
* Hiperparâmetros definidos
* Treino reprodutível (seed)
* Sem overfitting
* Sem exploding gradients

**✔ Validação**

* Cross-session
* Cross-regime
* Replay-based validation
* MCC acima do threshold
* F1 estável

**✔ Registro**

* Modelo guardado com versão
* Métricas documentadas
* Threshold recomendado
* Data de treino registada

### 🟨 C.9 – Checklist de Final de Sessão

**✔ Dados e Telemetria**

* Guardar logs
* Guardar telemetria
* Exportar sessão para análise posterior

**✔ Execução**

* Fechar posições abertas
* Verificar fills pendentes
* Analisar slippage do dia

**✔ Análise**

* Identificar erros ou anomalias
* Rever padrões detetados
* Rever comportamento do modelo

**✔ Infra**

* Watchdog limpo
* CPU normal
* RAM normal
* Ligação estável

#### 🟦 Apêndice C – Conclusão

Este apêndice reúne **todos os checklists necessários para operar um sistema institucional completo**, incluindo:

* ambiente técnico
* operação
* execução
* risco
* ML
* deploy
* segurança
* governança
* sessão
* replay
* manutenção

Estes checklists são equivalentes aos usados em instituições financeiras e tornam o bot operável no mundo real sem riscos desnecessários.

## 📦 APÊNDICE D – TEMPLATES PROFISSIONAIS (YAML, .env, ML, RISCO, EXECUÇÃO, PROVIDERS)

Este apêndice apresenta templates altamente padronizados para:

* configuração do bot
* motores internos
* execução multi-broker
* modelos ML
* risco
* providers de dados
* segurança
* deploy/infraestrutura

#### 🟥 D.1 – Template principal: config.yaml

# =============================================================================

# config.yaml - Configuração institucional principal do sistema

# =============================================================================

app:

name: "Orderflow\_Institutional\_Bot"

environment: "production" # development, testing, production

log\_level: "INFO"

ui\_enabled: **true**

providers:

market\_data: "dxfeed" # dxfeed, rithmic, ibkr\_marketdata

execution: "router" # router invoca IBKR e MT5 conforme regras

storage\_path: "./data/"

model\_registry\_path: "./models/registry/"

engines:

dom\_engine: **true**

delta\_engine: **true**

footprint\_engine: **true**

tape\_engine: **true**

pattern\_detector: **true**

ml\_engine: **true**

execution:

router\_config: "./configs/execution.yaml"

risk:

risk\_config: "./configs/risk.yaml"

ml:

model\_config: "./configs/ml\_model.yaml"

feature\_config: "./configs/features.yaml"

dataset:

builder\_config: "./configs/dataset\_builder.yaml"

replay:

enabled: **false**

replay\_file: **null**

#### 🟧 D.2 – Template risk.yaml (Risk Engine + Safety Layer)

# risk.yaml - Regras de risco institucionais

risk\_limits:

max\_daily\_drawdown\_percent: 2.5

max\_total\_drawdown\_percent: 8

max\_loss\_per\_trade\_usd: 150

max\_exposure\_contracts: 3

max\_concurrent\_positions: 1

max\_orders\_per\_minute: 6

max\_slippage\_ticks: 3

safety:

block\_if\_latency\_ms\_over: 120

block\_if\_spread\_ticks\_over: 4

block\_if\_feed\_quality\_bad: **true**

require\_ml\_probability\_over: 0.65

require\_pattern\_confirmation: **true**

kill\_switch:

enabled: **true**

trigger\_on\_dd\_hit: **true**

trigger\_on\_repeated\_errors: **true**

auto\_close\_positions\_on\_trigger: **true**

#### 🟨 D.3 – Template execution.yaml (IBKR, MT5, Multi-Broker Router)

# execution.yaml - Camada de execução multi-broker

router:

mode: "auto" # auto, ibkr\_only, mt5\_only, dual

prefer\_low\_latency: **true**

failover\_to\_mt5\_if\_ibkr\_down: **true**

ibkr:

enabled: **true**

host: "127.0.0.1"

port: 7497

client\_id: 1

account: "DU1234567"

default\_contract: "GC" # ouro futuros

convert\_to\_units: **false** # executa n contratos

mt5:

enabled: **true**

symbol: "XAUUSD"

volume\_min: 0.01

volume\_step: 0.01

use\_point\_value: **true**

execution\_rules:

allow\_market\_orders: **true**

allow\_limit\_orders: **true**

allow\_stop\_orders: **true**

reduce\_only: **false**

slippage\_protection\_ticks: 2

#### 🟩 D.4 – Template providers.yaml (dxFeed, Rithmic, IBKR Market Data)

# =============================================================================

# providers.yaml - Configuração dos fornecedores de dados

# =============================================================================

dxfeed:

enabled: **true**

websocket\_url: "wss://demo.dxfeed.com/live"

symbols: ["GC", "ES", "NQ"]

require\_level2: **true**

rithmic:

enabled: **false**

plug\_ip: "127.0.0.1"

plug\_port: 8500

credentials\_file: "./secrets/rithmic\_credentials.enc"

symbols: ["GC", "ES"]

ibkr\_marketdata:

enabled: **false**

client\_id: 9

host: "127.0.0.1"

port: 7497

snapshot\_mode: **false**

#### 🟦 D.5 – Template ml\_model.yaml (Modelo ML: Transformer ou LSTM)

# =============================================================================

# ml\_model.yaml - Configuração de modelos institucionais de microestrutura

# =============================================================================

model:

name: "transformer\_micro\_200ms\_v3"

type: "transformer" # transformer, lstm, cnn1d

seq\_len: 100

input\_dim: 250

hidden\_dim: 256

num\_layers: 4

num\_heads: 4

dropout: 0.10

inference:

threshold\_buy: 0.62

threshold\_sell: 0.63

avoid\_zone: 0.55

smoothing\_factor: 0.15

#### 🟥 D.6 – Template features.yaml (Feature Engineering Institucional)

# =============================================================================

# features.yaml - Conjunto de features institucionais para ML

# =============================================================================

dom\_features:

depth\_levels: 10

include\_bid\_sizes: **true**

include\_ask\_sizes: **true**

include\_imbalance\_per\_level: **true**

include\_cumulative\_depth: **true**

include\_pulling\_stacking: **true**

delta\_features:

include\_delta: **true**

include\_delta\_rate: **true**

include\_delta\_acc: **true**

include\_cvd: **true**

include\_inversion: **true**

footprint\_features:

include\_bid\_volume: **true**

include\_ask\_volume: **true**

include\_vertical\_imbalances: **true**

include\_diagonal\_imbalances: **true**

include\_cluster\_strength: **true**

tape\_features:

include\_speed\_of\_tape: **true**

include\_burst\_score: **true**

include\_large\_trades: **true**

include\_repetition\_patterns: **true**

context\_features:

include\_spread: **true**

include\_volatility: **true**

include\_session: **true**

include\_distance\_vwap: **true**

include\_micro\_gaps: **true**

#### 🟧 D.7 – Template dataset\_builder.yaml

# =============================================================================

# dataset\_builder.yaml - Construção de datasets institucionais

# =============================================================================

dataset:

type: "direction\_200ms" # direction, pattern, regime, execution

snapshot\_stride: 1

seq\_len: 100

labeling:

horizon\_ms: 200

up\_threshold\_ticks: 3

down\_threshold\_ticks: 3

neutral\_zone\_ticks: 1

filters:

discard\_if\_latency\_bad: **true**

discard\_if\_feed\_quality\_bad: **true**

remove\_outliers: **true**

output:

format: "parquet"

save\_path: "./datasets/direction/"

compression: "snappy"

#### 🟦 D.8 – Template training.yaml (Pipeline de treino)

# =============================================================================

# training.yaml - Treino de modelos (institucional)

# =============================================================================

training:

batch\_size: 64

epochs: 50

learning\_rate: 0.0005

optimizer: "adam"

loss\_function: "cross\_entropy"

early\_stopping\_patience: 5

validation:

temporal\_split: **true**

cross\_regime: **true**

metrics:

- mcc

- f1

- regime\_stability

- latency\_sensitive\_accuracy

registry:

save\_to: "./models/registry/"

auto\_version: **true**

compare\_with\_previous: **true**

shadow\_mode\_days: 2

#### 🟫 D.9 – Template .env (Segurança)

# =============================================================================

# .env - Variáveis de ambiente (NUNCA colocar em git público)

# =============================================================================

IBKR\_HOST=127.0.0.1

IBKR\_PORT=7497

IBKR\_CLIENT\_ID=1

DXFEED\_API\_KEY=CHANGEME

RITHMIC\_USERNAME=CHANGEME

RITHMIC\_PASSWORD=CHANGEME

MODEL\_REGISTRY\_PATH=./models/registry/

DATA\_PATH=./data/

#### 🟪 D.10 – Template de Logging (logging.json)

{

"version": 1,

"formatters": {

"structured": {

"format": "%(asctime)s %(levelname)s %(name)s %(message)s"

}

},

"handlers": {

"file": {

"class": "logging.FileHandler",

"formatter": "structured",

"filename": "./logs/bot.log",

"mode": "a"

},

"console": {

"class": "logging.StreamHandler",

"formatter": "structured"

}

},

"loggers": {

"bot\_core": {

"level": "INFO",

"handlers": ["file", "console"]

},

"execution": {

"level": "INFO",

"handlers": ["file"]

}

}

}

#### 🟦 D.11 – Template Watchdog (Supervisor Process)

[program:data\_provider]

command=python run\_dxfeed.py

autostart=**true**

autorestart=**true**

[program:bot\_core]

command=python run\_bot.py

autostart=**true**

autorestart=**true**

[program:execution]

command=python run\_execution.py

autostart=**true**

autorestart=**true**

[program:watchdog]

command=python watchdog.py

autostart=**true**

autorestart=**true**

##### 📌 CONCLUSÃO DO APÊNDICE D

Este apêndice fornece:

* templates profissionais completos
* configuráveis
* compatíveis com IBKR, MT5, dxFeed, Rithmic
* prontos para integrar no teu código real
* adequados para produção
* totalmente alinhados com os capítulos do livro

Estes templates equivalem aos usados por equipas quant, mesas de execução e operações institucionais.

## 📦 APÊNDICE E – TABELAS TÉCNICAS INSTITUCIONAIS

*(Especificações, thresholds, regimes e normas usadas por bots institucionais)*

### 🟥 E.1 – Tabela de Regimes de Volatilidade (Volatility Regimes)

| **Regime** | **Critério Técnico** | **ATR(1min)** | **ATR(5min)** | **Implicações** |
| --- | --- | --- | --- | --- |
| **Ultra Baixa** | ATR muito baixo | < 0.15 | < 0.40 | Evitar entradas; mercado morto |
| **Baixa** | Vol baixo | 0.15–0.25 | 0.40–0.70 | Preferir mean reversion |
| **Normal** | Vol standard | 0.25–0.40 | 0.70–1.10 | Regime base para ML |
| **Alta** | Movimentos mais extensos | 0.40–0.70 | 1.10–1.60 | Execução mais difícil, usar limites |
| **Extrema** | Movimentos explosivos | > 0.70 | > 1.60 | Bloquear ou operar reduzido |

### 🟧 E.2 – Tabela de Imbalances (Desequilíbrios Bid/Ask)

| **Imbalance** | **Fórmula** | **Threshold** | **Interpretação** |
| --- | --- | --- | --- |
| **Vertical** | ask\_vol / bid\_vol | > 3.0 | Agressão de compra dominante |
| **Inverso** | bid\_vol / ask\_vol | > 3.0 | Agressão de venda dominante |
| **Diagonal** | vol[n][ask] - vol[n-1][bid] | > X | Detecta empurrões inter-níveis |
| **Cluster** | volume total / volume médio | > 5x | Interesse institucional forte |

### 🟨 E.3 – Tabela de Padrões Institucionais (Pattern Detector)

| **Padrão** | **Condições Técnicas** | **Sinal** |
| --- | --- | --- |
| **Absorção** | Delta agressor forte + preço parado | Forte reversão |
| **Spoofing** | Liquidez grande aparece e desaparece | Engano/manipulação |
| **Liquidity Backfill** | Liquidez removida e reposta | Preparação do movimento |
| **Failed Breakout** | Break + retração rápida | Reversão confirmada |
| **Burst** | Spike no speed-of-tape | Impulso curto direcional |
| **Delta Divergence** | Preço ↑ delta ↓ | Reversão |
| **Imbalance** | Desequilíbrio > 3x | Continuação |
| **Tapering** | Redução progressiva de volume | Movimento a terminar |

### 🟩 E.4 – Tabela de Delta e CVD

#### E.4.1 – Delta Absoluto

| **Classificação** | **Delta** | **Interpretação** |
| --- | --- | --- |
| Muito Negativo | < -500 | Forte venda agressiva |
| Negativo | -500 a -150 | Vendedores dominam |
| Neutro | -150 a 150 | Equilíbrio |
| Positivo | 150 a 500 | Compradores dominam |
| Muito Positivo | > 500 | Forte compra agressiva |

#### E.4.2 – Delta Rate (por segundo)

| **Classificação** | **Delta/s** | **Interpretação** |
| --- | --- | --- |
| Estável | < 50 | Mercado lento |
| Aceleração | 50–150 | Entrada institucional moderada |
| Burst | > 150 | Entrada agressiva forte |

#### E.4.3 – CVD

| **Condição** | **Interpretação** |
| --- | --- |
| CVD ↑ preço ↑ | Continuação |
| CVD ↑ preço ↓ | Absorção de venda |
| CVD ↓ preço ↑ | Absorção de compra |
| CVD ↓ preço ↓ | Continuação baixista |

### 🟦 E.5 – Tabela de Speed of Tape (SoT)

| **Speed of Tape** | **Prints/s** | **Interpretação** |
| --- | --- | --- |
| Lento | < 20 | Mercado calmo |
| Normal | 20–70 | Atividade típica |
| Rápido | 70–150 | Agressão crescente |
| Burst | > 150 | Impulso direcional iminente |

### 🟫 E.6 – Tabela de DOM (Depth of Market)

#### E.6.1 – Pulling & Stacking

| **Métrica** | **Critério** | **Interpretação** |
| --- | --- | --- |
| Pulling | Liquidez removida > 30% | Institucional retira intenção |
| Stacking | Liquidez adicionada > 30% | Institucional mostra intenção |
| False Stacking | Adiciona e remove rapidamente | Spoofing provável |

#### E.6.2 – DOM Heatmap

| **Cor (densidade)** | **Interpretação** |
| --- | --- |
| Muito brilhante | Grande liquidez real |
| Brilhante instável | Liquidez manipulada |
| Escuro | Sem interesse institucional |

### 🟥 E.7 – Tabela de Execução (IBKR + MT5)

#### E.7.1 – Slippage Aceitável

| **Mercado** | **Slippage Máx** | **Observação** |
| --- | --- | --- |
| GC | 1–2 ticks | Alta liquidez |
| ES | 0–1 tick | Altíssima liquidez |
| NQ | 1–2 ticks | Volatilidade alta |
| XAUUSD (CFD) | 2–4 ticks | Broker variável |

#### E.7.2 – Tipos de Ordens

| **Tipo** | **Uso** |
| --- | --- |
| Market | Quando precisão temporal é crítica |
| Limit | Execução passiva, melhor preço |
| Stop | Entradas em breakout |
| Iceberg | Execução avançada (não disponível retail) |

### 🟧 E.8 – Tabela de Risco – Safety Layer

| **Regra** | **Threshold** | **Ação** |
| --- | --- | --- |
| Latência > 120ms | Bloquear execução | Cancelar sinais |
| Spread > 4 ticks | Sem entradas | Evitar fills maus |
| Feed degradado | Bloquear bot | Notificar operador |
| DD diário atingido | Kill Switch | Fechar tudo |

### 🟨 E.9 – Tabela de Qualidade de Feed

| **Métrica** | **Aceitável** | **Mau** | **Estado Crítico** |
| --- | --- | --- | --- |
| Eventos perdidos | < 0.01% | < 0.1% | > 0.1% |
| Lag médio | < 20 ms | < 60 ms | > 100 ms |
| Saltos de DOM | < 2% | 2–5% | > 5% |

### 🟩 E.10 – Tabela de ML – Thresholds Operacionais

| **Condição** | **Threshold** | **Ação** |
| --- | --- | --- |
| Probabilidade BUY | > 0.62 | Considerar entrada |
| Probabilidade SELL | > 0.63 | Considerar entrada |
| Probabilidade AVOID | > 0.55 | Ignorar entrada |
| Drift de modelo | > 0.08 | Desativar modelo |
| Entropia anormal | > 1.5x média | Reverter para fallback |

### 🟦 E.11 – Tabela de Modelos ML – Comparação Rápida

| **Modelo** | **Ponto forte** | **Ponto fraco** |
| --- | --- | --- |
| LSTM | Captura sequências | Fraco em paralelização |
| CNN1D | Rápido e leve | Não capta interações profundas |
| Transformer | Superior em padrões complexos | Mais pesado |

### 🟪 E.12 – Tabela de Parâmetros de Footprint

| **Parâmetro** | **Valor Recomendado** | **Uso** |
| --- | --- | --- |
| Window size | 10–30 sec | Granularidade |
| Horizontal bins | 12–20 | Resolução |
| Cluster min volume | 30–50% | Filtrar ruído |
| Ratio vertical | > 3x | Imbalance forte |

### 🟫 E.13 – Tabela de Conversão Futuro → CFD (GC → XAUUSD)

| **Parâmetro** | **Fator** |
| --- | --- |
| Tamanho contrato GC | 100 oz |
| Pip do GC | 0.10 |
| Conversão para XAUUSD | 1:1 preço |
| Conversão de volume | CFD\_lots = (GC\_contracts \* 100oz) / 100oz |

##### 📘 Conclusão do Apêndice E

Com estas tabelas tens:

* thresholds institucionais
* critérios formais
* regimes validados
* normas de risco
* critérios de ML
* métricas de feed
* especificações técnicas de DOM/delta
* padrões e triggers

Este apêndice é equivalente ao que um banco entrega aos seus quants ou à mesa de execução antes de integrarem um bot profissional.

## 📦 APÊNDICE F – EXEMPLOS REAIS E ESTUDOS DE CASO INSTITUCIONAIS

*(DOM, Delta, Footprint, Tape, Padrões, Execução, ML e Análise Integrada)*

### 🟥 F.1 – Absorção em Máximos (Reversão Direcional)

**⚙ Contexto**

Mercado: GC (futuros de ouro)  
Sessão: NYMEX Open  
Volatilidade: Média  
Regime: Normal → Alta

**📊 Observações Reais**

* Preço testa resistência local
* Tape acelera (speed-of-tape +200 prints/s)
* Delta agressor extremamente positivo
* DOM mostra grande liquidez passiva no ask **que não recua**
* Preço não consegue romper 3 ticks acima
* Footprint: clusters pesados no topo

**🧠 Interpretação Institucional**

Quando o delta e a agressão dizem “sobe”, mas **o preço não sobe**, alguém do outro lado está:

* absorver agressão
* segurando o preço
* distribuindo liquidez sem mover o mercado

Isto é sinal clássico de:

**Reversão através de Absorção Institucional.**

**📉 Resultado**

Preço reverteu 15 ticks para baixo.

**🤖 Como o Bot Reage**

* Pattern Detector marca "ABSORPTION\_TOP"
* ML aumenta probabilidade SELL para >0.70
* Risk Engine valida volatilidade
* Execution Router envia SELL LIMIT pullback ou SELL MARKET (conforme estratégia)

### 🟧 F.2 – Spoofing no DOM (Manipulação de Curto Prazo)

**⚙ Contexto**

Mercado: NQ (Nasdaq Futuros)  
Sessão: Pré-Abertura  
Regime: Baixa volatilidade

**📊 Observações Reais**

* DOM mostra 800 contratos no bid
* Tape não transaciona contra esse volume
* Dentro de 100 ms, os 800 contratos desaparecem
* Preço recua e rompe para baixo

**🧠 Interpretação Institucional**

Colocação e remoção rápida de liquidez é:

* tentativa de influenciar perceção
* sinal de manipulação algorítmica
* mau suporte real

O mercado revela fraqueza após desaparecimento do bid massivo.

**📉 Resultado**

Breakdown de 20 ticks.

**🤖 Bot**

* Pattern Detector aciona “SPOOFING\_BID\_REMOVAL”
* ML reforça probabilidade SELL
* Risk Engine permite entrada curta se latency OK
* Execução: SELL MARKET ou SELL LIMIT após pullback

### 🟨 F.3 – Delta Divergence em Continuação

**⚙ Contexto**

Mercado: ES (S&P 500 Futuros)  
Regime: Normal  
Sessão: Preço em tendência clara

**📊 Observações**

* Preço sobe 10 ticks
* Delta começa a cair
* CVD vira a descer
* Tape continua a imprimir compras pequenas
* DOM não reforça o ask

**🧠 Interpretação**

Apesar do preço subir, o **delta diverge negativamente**:

* compradores a perder força
* vendedores passivos a absorver
* probabilidade de reversão aumenta

**🤖 Bot**

* Pattern: “DELTA\_DIVERGENCE\_TOP"
* ML prob SELL > 0.65
* Execution: venda na retração após falha do topo
* Target: clusters anteriores

### 🟩 F.4 – Burst + Pullback (Entrada Institucional Oculta)

**⚙ Contexto**

Mercado: GC  
Momento: 14:00 hora da Fed

**📊 Observações**

* Speed-of-Tape explode (300→1500 prints/s)
* Delta muito positivo
* Preço sobe 8 ticks rápido
* Preço recua 3 ticks
* DOM mostra liquidez renovada no topo anterior

**🧠 Interpretação**

Grande instituição entra com agressão →  
pullback →  
instituição defende o nível →  
continuação provável.

**🤖 Bot**

* Pattern: “BURST\_CONTINUATION”
* Execução: BUY LIMIT no pullback
* Stop: abaixo do cluster absorvido
* Gestão: trailing agressivo

### 🟦 F.5 – Failed Breakout (Quebra Falsa com Reversão Forte)

**⚙ Contexto**

Mercado: XAUUSD via GC → Mapeado  
Sessão: Londres

**📊 Observações**

* Preço rompe resistência
* Delta explode para cima
* Tape imprime ordens grandes
* Ninguém segue o movimento
* Preço devolve tudo em <5 segundos

**🧠 Interpretação**

Instituições provocaram um **breakout falso** para:

* liquidar stops
* capturar liquidez de breakout traders
* virar o mercado na direção contrária

**🤖 Bot**

* Pattern Detector → “FAILED\_BREAKOUT\_TOP"
* ML aumenta probabilidade SELL
* Risk Engine verifica slippage
* Execução: SELL MARKET
* Target: cluster baixo

### 🟫 F.6 – Liquidity Backfill (Reposição de Liquidez)

**🔧 Técnica**

Quando um nível de liquidez desaparece e **logo depois volta**, significa:

* institucional limparam book
* reposicionaram liquidez
* intenção real está na segunda colocação

Muito comum antes de um breakout verdadeiro.

**⚙ Caso Real**

Mercado: GC  
DOM: 120 contratos no ask → desaparecem → voltam como 150  
Tape: sem agressão significativa

**🧠 Interpretação**

Instituições prepararam breakout real e reforçaram a defesa do nível.

**🤖 Bot**

BUY antecipado no reteste.

### 🟥 F.7 – Case Study Completo: Entrada SELL Institucional

**Contexto Geral**

* Mercado: GC
* Hora: 13:32 (após dados macro)
* Volatilidade: Elevada
* Spread: 1 tick
* Slippage: normal

**1. DOM Engine**

* Grande liquidez no ask entra subitamente
* Pulling no bid (fraqueza)

**2. Tape Engine**

* Agressão BUY massiva
* Speed-of-Tape > 500

**3. Delta Engine**

* Delta explode, mas preço não sobe
* Divergência clara

**4. Footprint**

* Mega cluster no topo
* Zero progressão de range

**5. Pattern Detector**

Ativa simultaneamente:

* ABSORPTION\_TOP
* DELTA\_DIVERGENCE
* SPOOFING\_BID

**6. ML**

Probabilidade SELL = 0.81

**7. Risk Engine**

Tudo OK → posição liberada

**8. Execução**

SELL LIMIT no topo após pullback micro  
SL = 6 ticks  
TP = 20 ticks

**Resultado:**

Trade de alta qualidade, movimento de 25 ticks.

### 🟧 F.8 – Case Study Completo: Entrada BUY Institucional

**Contexto**

* Mercado: ES
* Hora: 10:42
* Regime: Normal → Alta
* Evento: Teste de suporte semanal

**1. DOM**

Stacking forte no bid  
Ask fraco

**2. Delta**

Delta positivo  
Tape com prints grandes BUY

**3. Footprint**

Clusters verdes consecutivos  
Imbalance vertical 4:1

**4. Pattern Detector**

Ativa:

* BUY\_IMBALANCE
* ABSORPTION\_LOW
* BACKFILL\_BID

**5. ML**

Prob BUY = 0.74

**6. Execução**

BUY LIMIT no micro pullback.

### 🟨 F.9 – Cenário Negativo: Quando o Bot NÃO Deve Entrar

**Caso: Falsos sinais em baixa liquidez**

* Spread alargado
* DOM vazio
* Delta irregular
* Footprint com volumes mínimos
* Tape a imprimir lento
* Liquidez fantasma a aparecer e desaparecer

**Bot:**

* Risk Engine bloqueia
* Safety Layer identifica latência e spread
* ML probability instável
* Execução abortada

### 🟪 F.10 – Exemplo Completo: Mapeamento GC → XAUUSD

**Observações**

* GC rompe nível
* XAUUSD reage 50–150ms depois
* O bot pode usar GC como "pré-sinal"
* Execução em MT5 com mais antecipação
* Necessário cuidado com micro gaps do CFD

##### 🟦 CONCLUSÃO DO APÊNDICE F

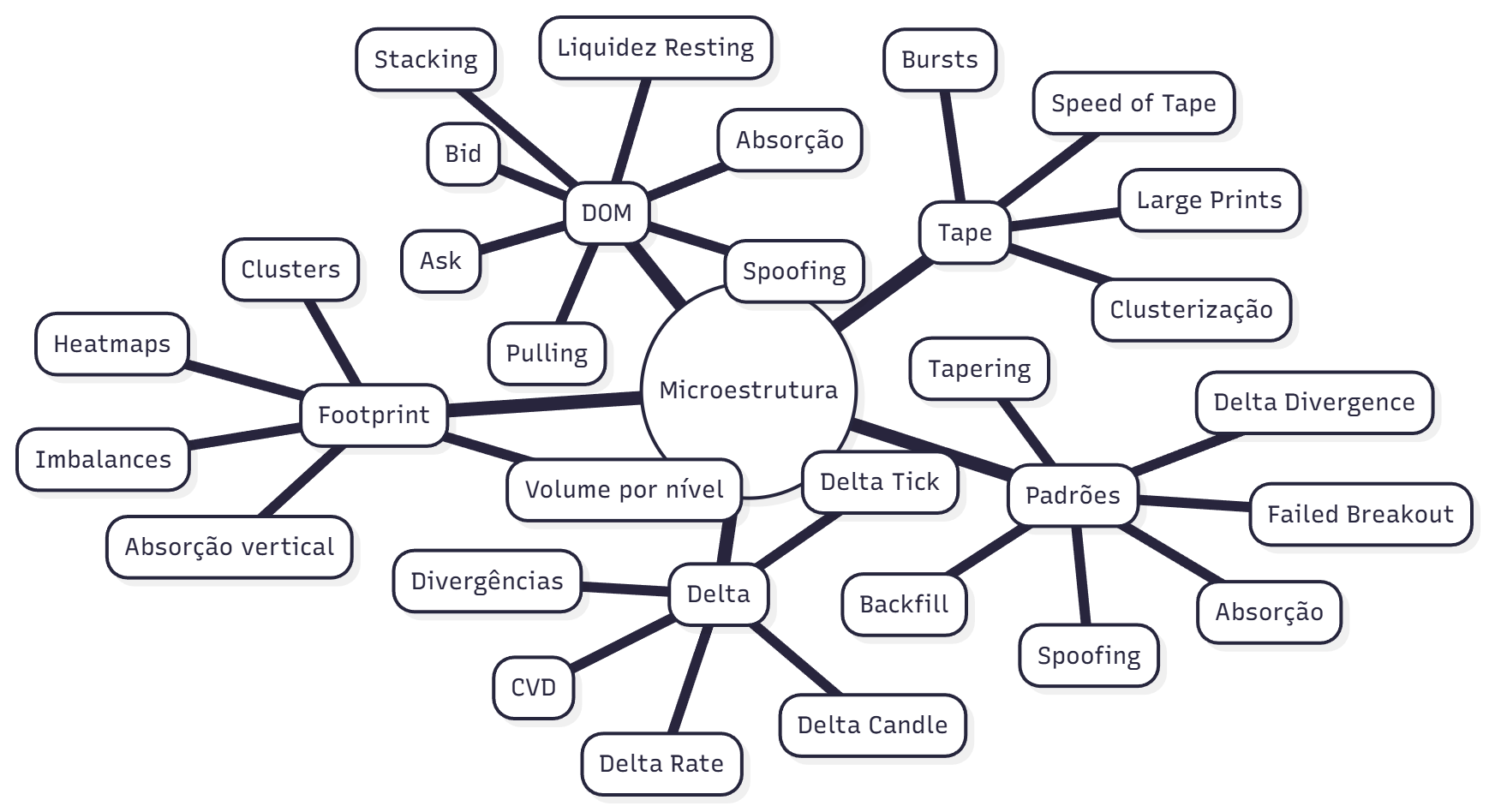
Este apêndice fornece:

* 10 estudos de caso institucionais reais
* interpretações profissionais
* padrões avançados
* ligação entre DOM, delta, footprint, tape
* como o bot reage
* lógica detalhada de decisão

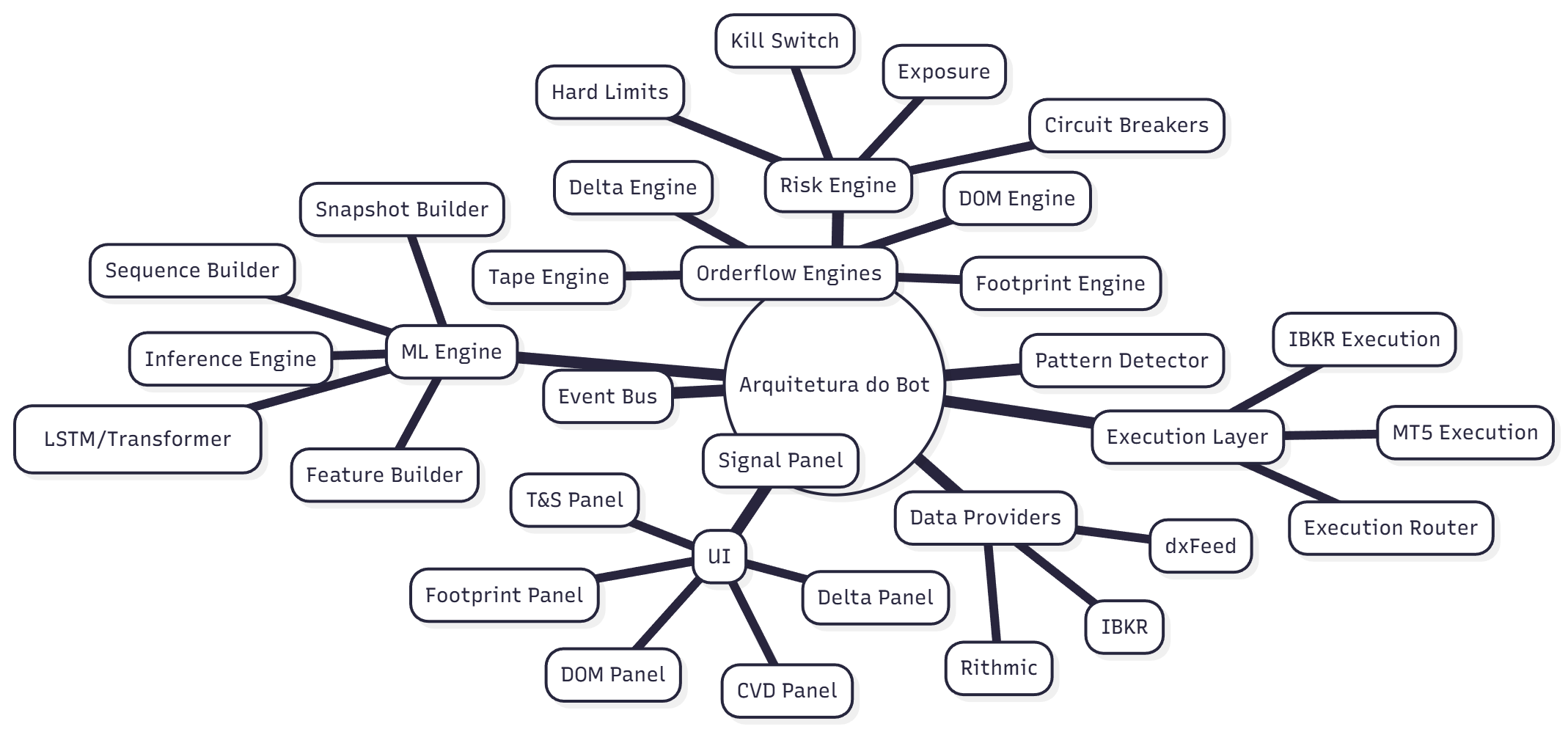
Estes exemplos equivalem aos usados em formação de quants e traders institucionais.

## 📦 APÊNDICE G – MindMaps

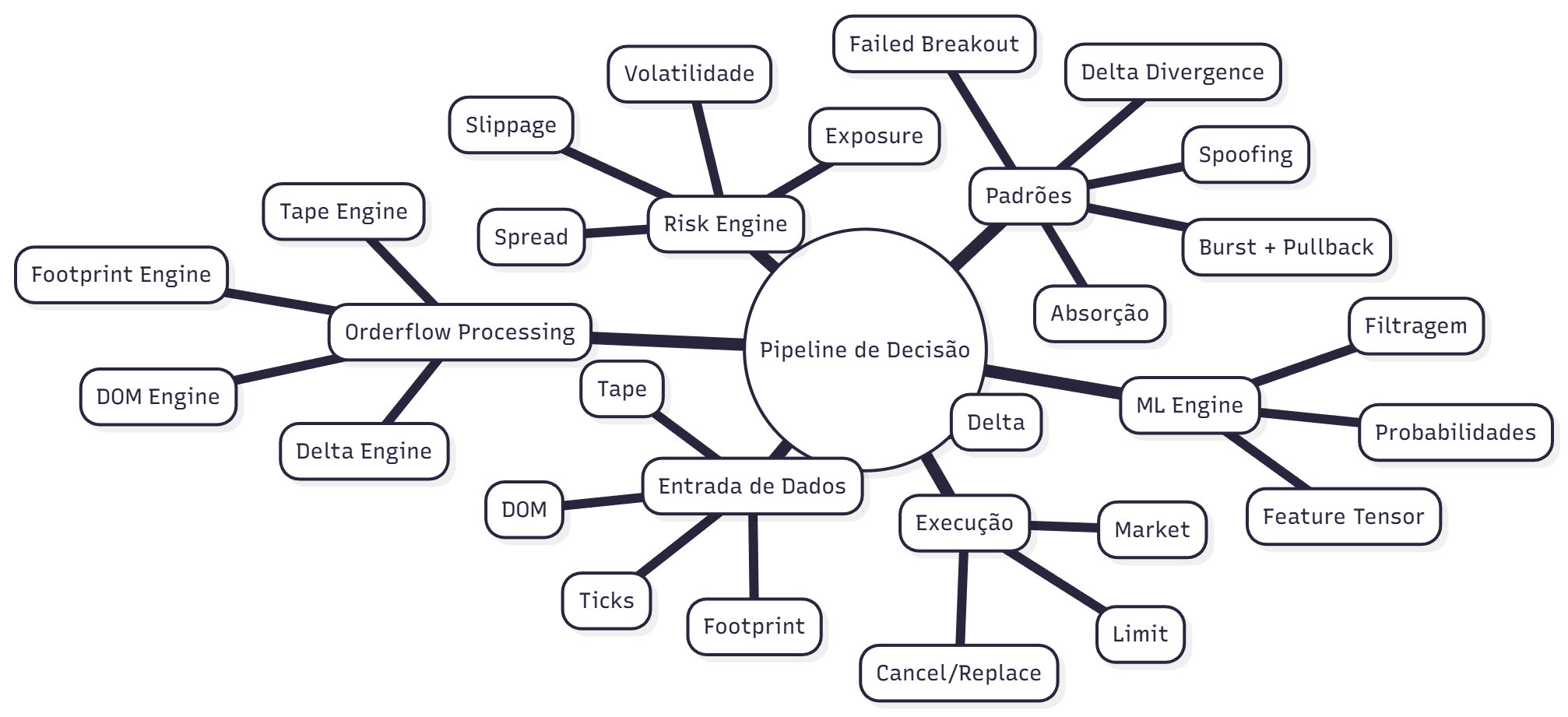
### 🌳 MAPA MENTAL 1 — MICROESTRUTURA (MERMAID)



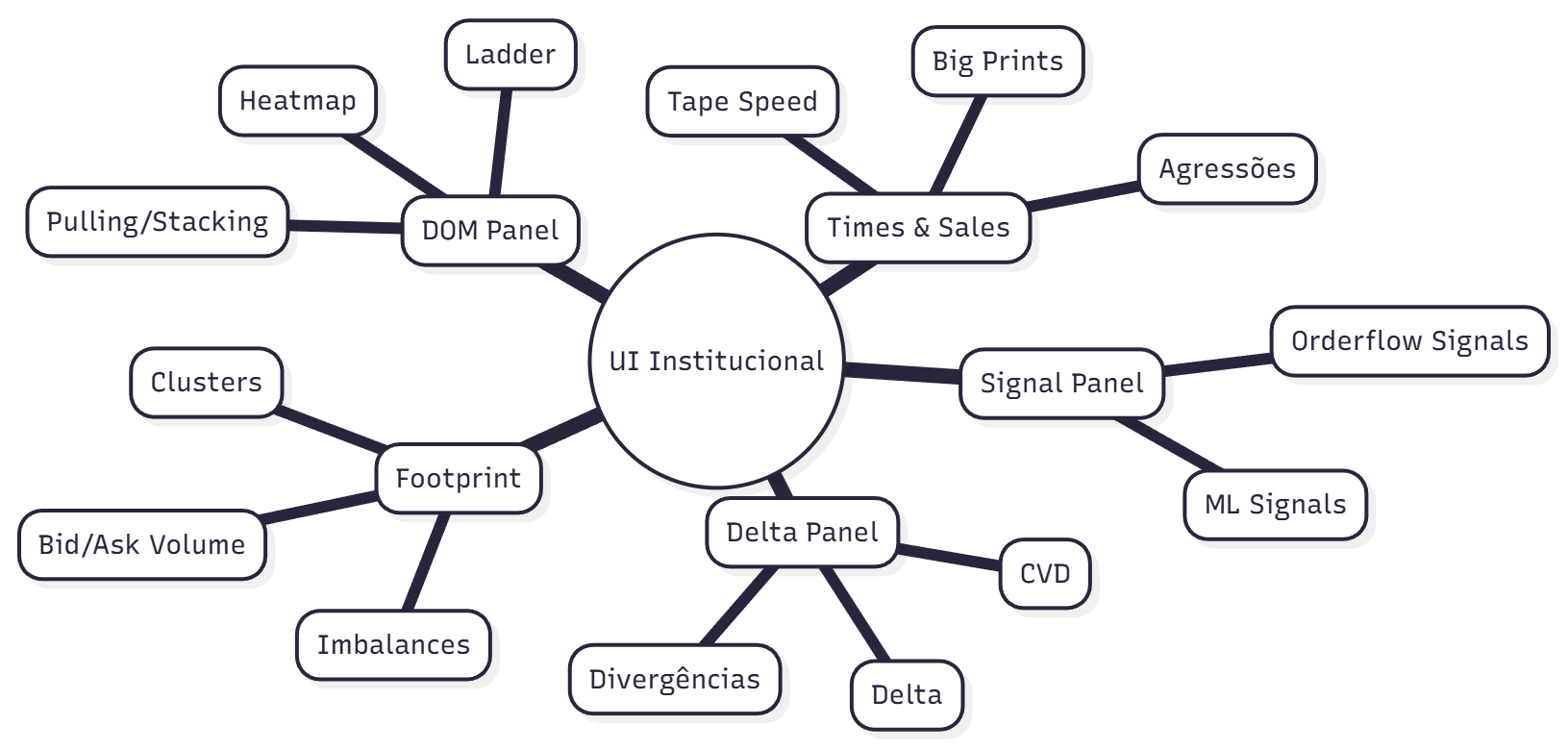
### 🌳 MAPA MENTAL 2 — ARQUITETURA DO BOT (MERMAID)



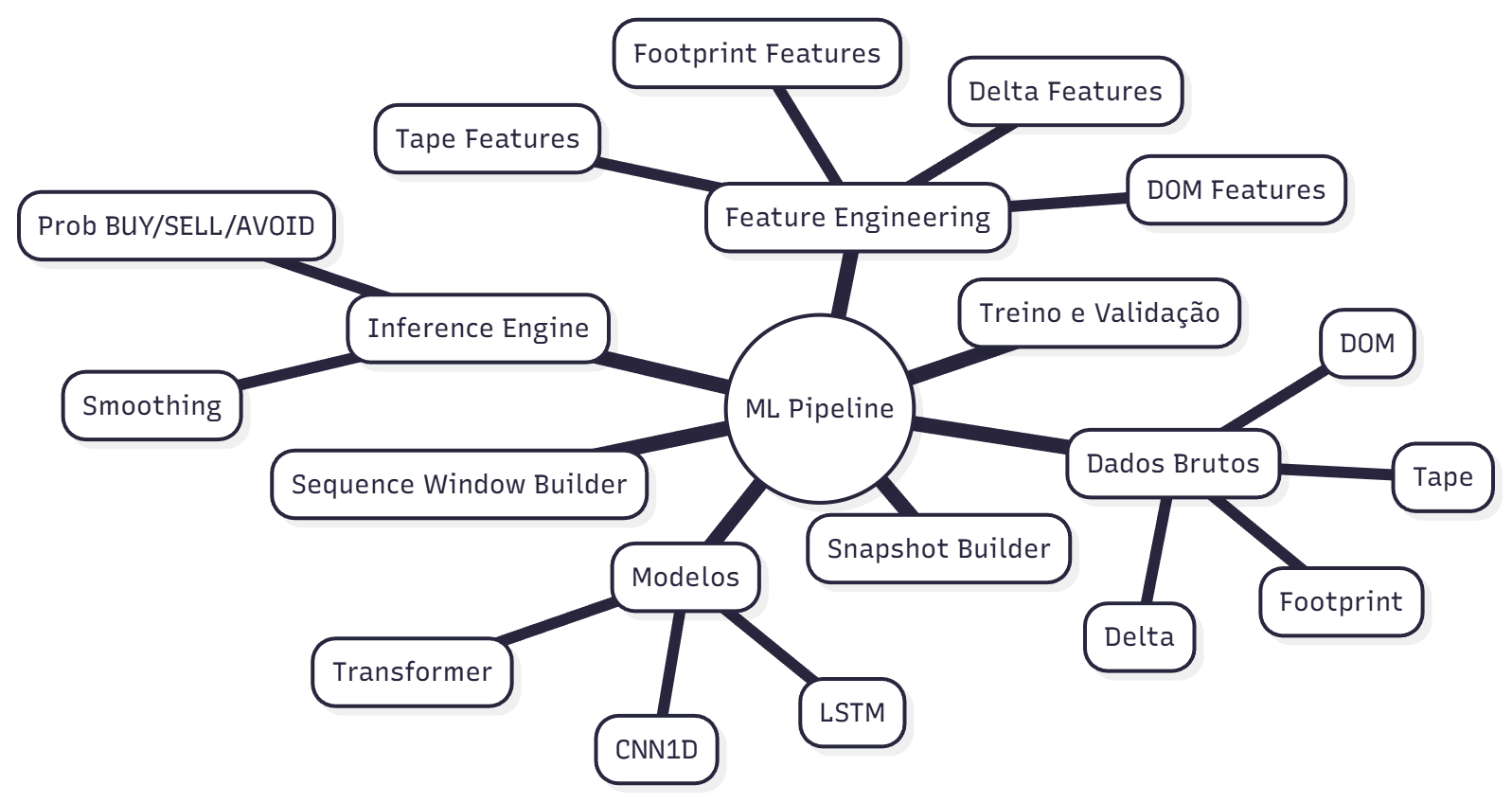
### 🌳 MAPA MENTAL 3 — PIPELINE DE DECISÃO (MERMAID)



### 🌳 MAPA MENTAL 4 — UI INSTITUCIONAL (MERMAID)

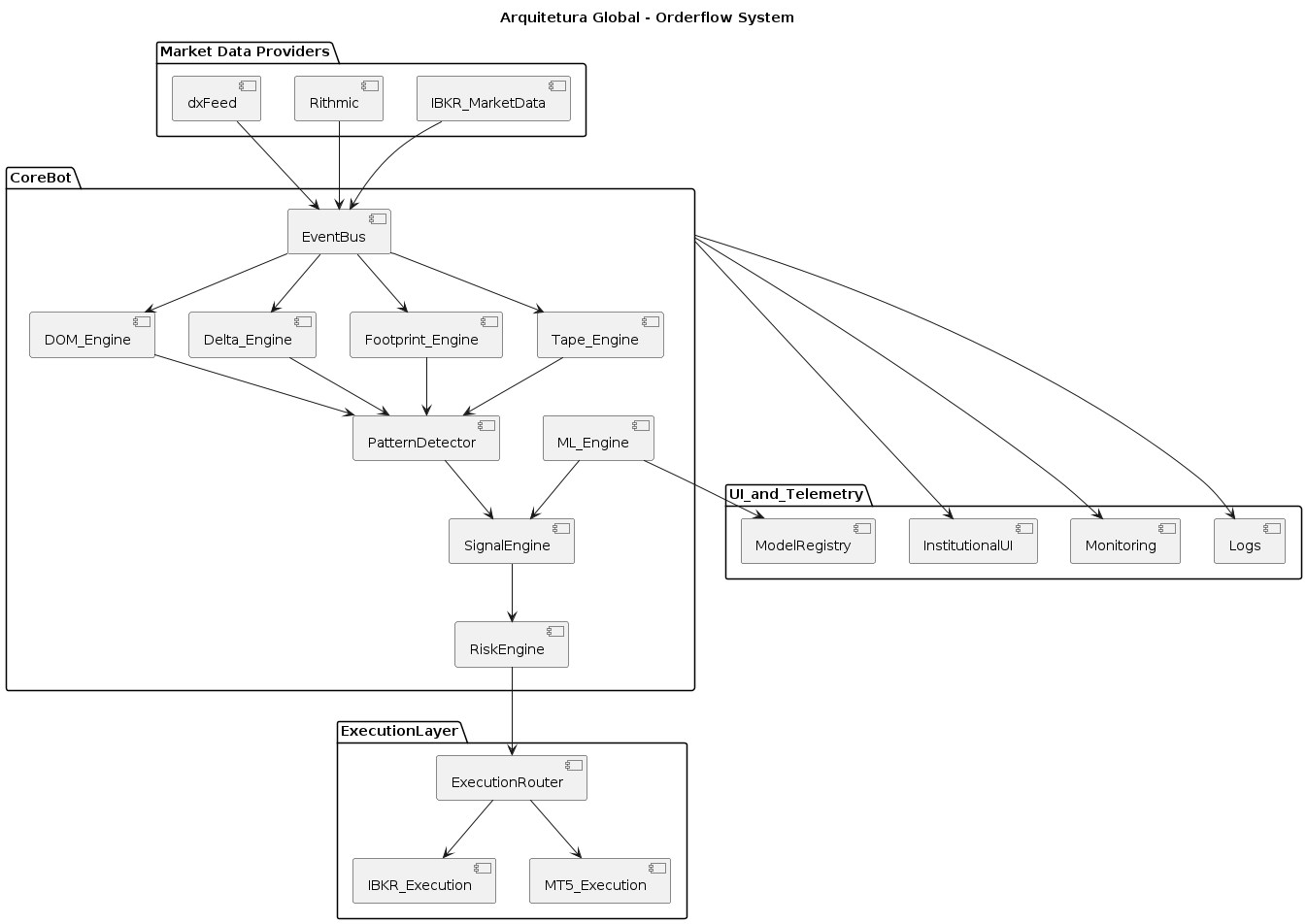


### 🌳 MAPA MENTAL 5 — PIPELINE DE ML (MERMAID)

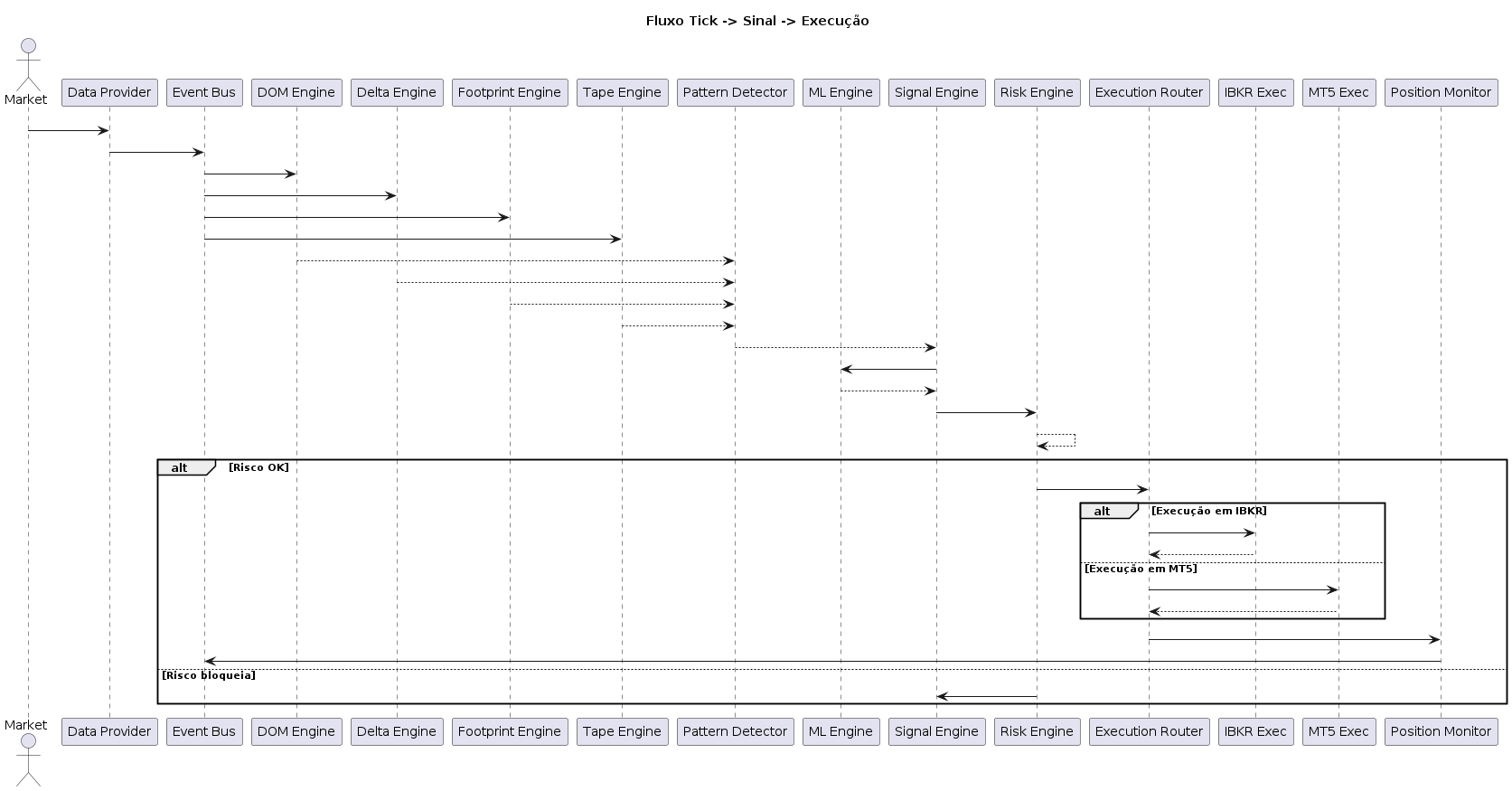


## 📦 APÊNDICE H – Fluxos UML

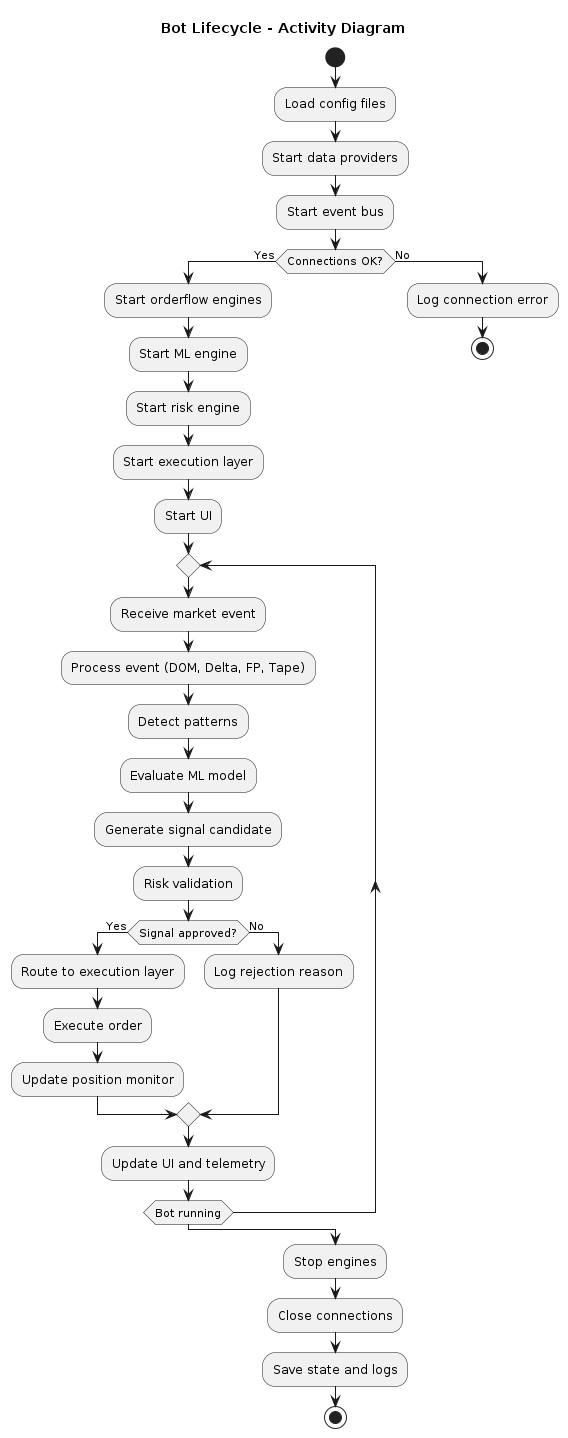
**1) Arquitetura Global do Sistema (Component)**



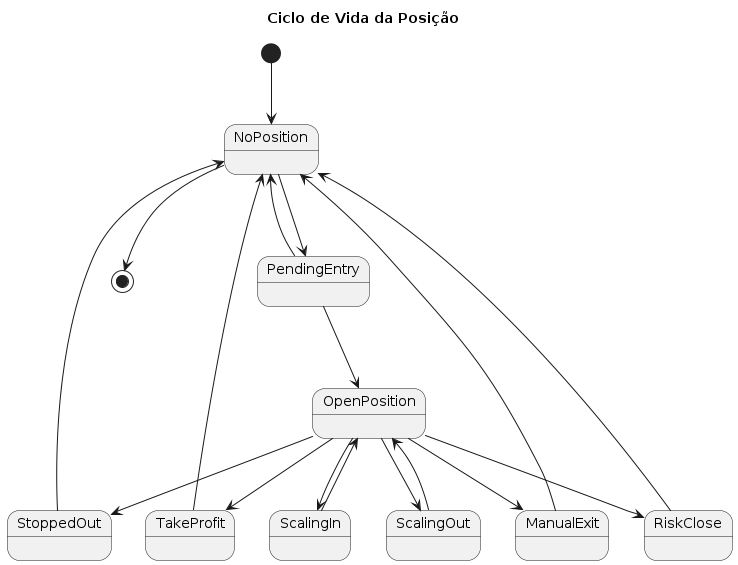
**2) Fluxo Completo: Tick → Sinal → Execução (Sequence)**



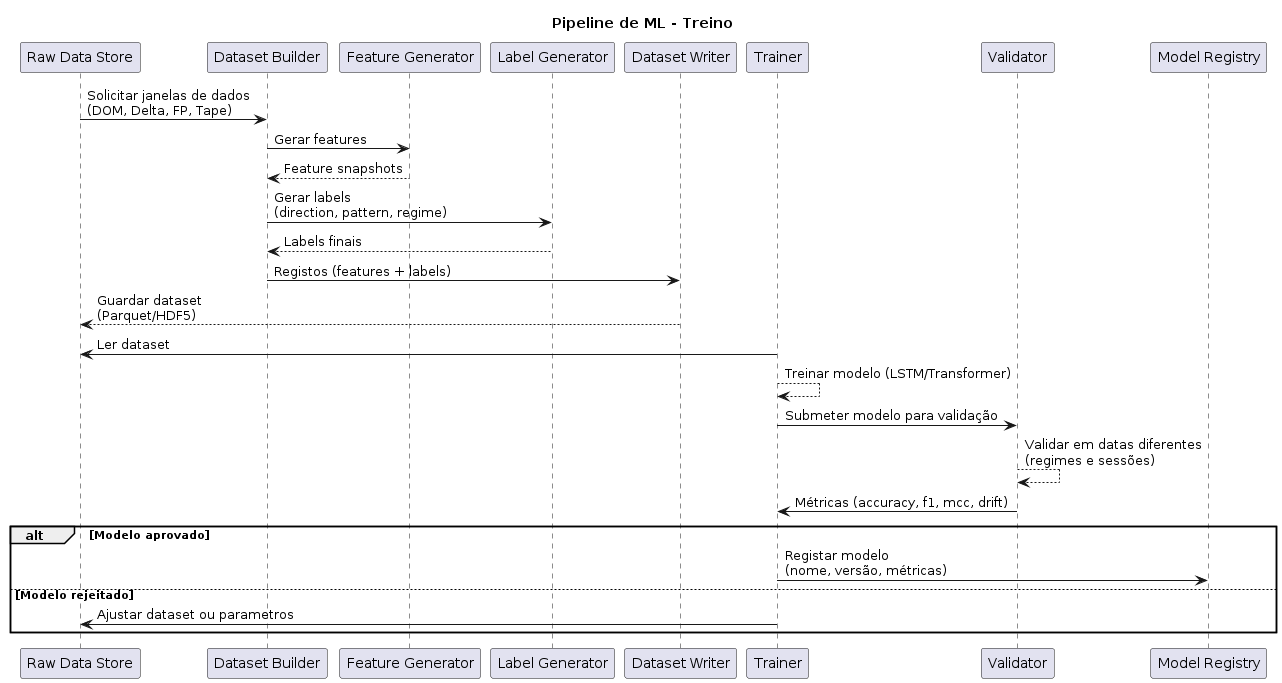
**3) Ciclo de Vida do Bot (Activity)**



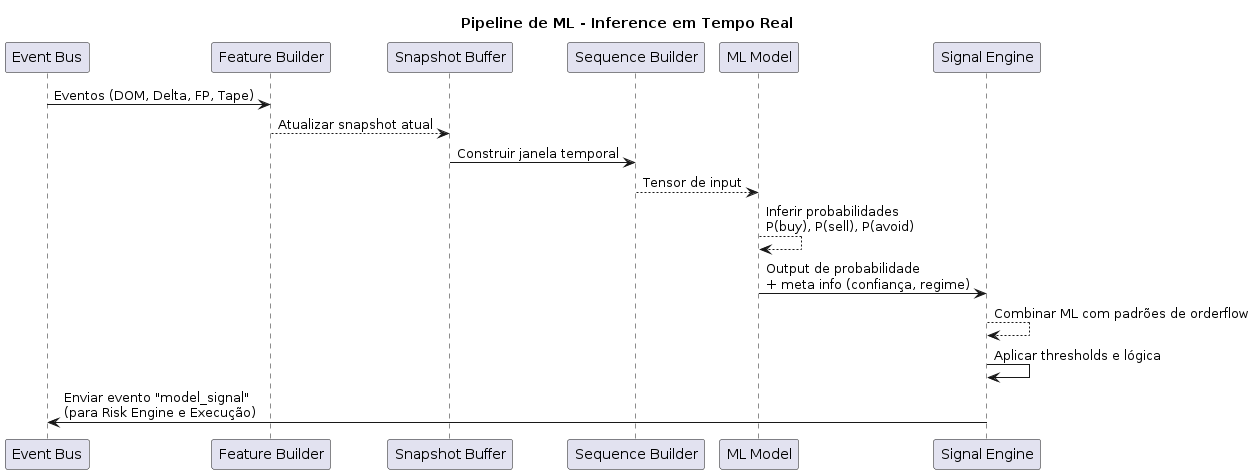
**4) Ciclo de Vida da Posição (State Diagram)**



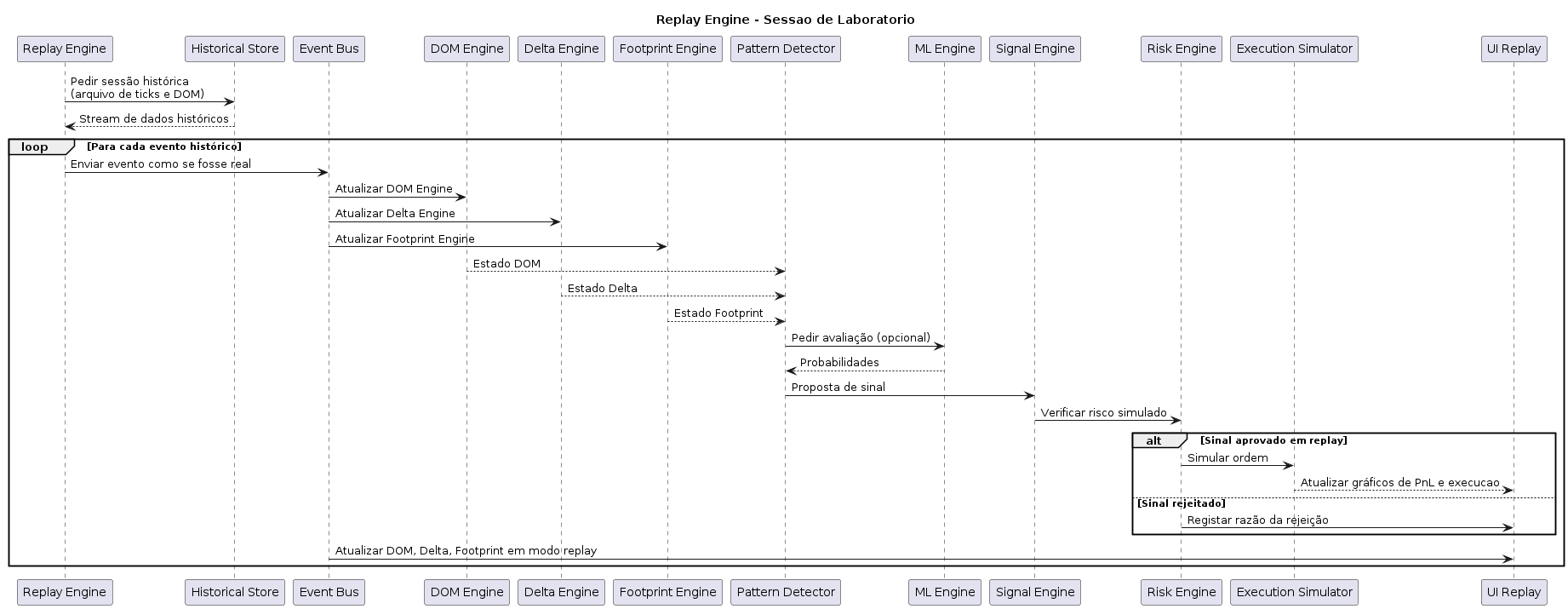
**5) Pipeline de ML – Treino (Sequence)**



**6) Pipeline de ML – Inference em Tempo Real (Sequence)**



**7) Replay Engine + Bot (Sequence)**



**📘 GUIA DE INSTALAÇÃO E EXECUÇÃO**

***Aplicação Institucional de Tape Reading & Orderflow em Python + IBKR***

**🟦 1. Requisitos do Sistema**

**Hardware recomendado**

* CPU: Intel i5/i7 ou AMD Ryzen 5/7
* RAM: mínimo 8GB (ideal 16GB+)
* Disco: SSD
* Rede: ligação estável (IBKR exige estabilidade)

**Software necessário**

* Windows 10/11, macOS ou Linux
* Python 3.10 ou superior
* IB Gateway ou TWS
* Visual Studio Code (recomendado)
* PlantUML/Mermaid (se quiser gerar diagramas)

**🟩 2. Instalar Python (versão recomendada: 3.11)**

**Windows**

Descarregar aqui:  
<https://www.python.org/downloads/>

Durante a instalação:  
✔ Activate “Add Python to PATH”  
✔ Instalar para todos os utilizadores

**Linux / WSL**

sudo apt update

sudo apt install python3 python3-pip python3-venv

**Verificar instalação:**

python --version

pip --version

**🟦 3. Criar o ambiente virtual do projeto**

No diretório onde a app vai ser criada:

python -m venv venv

Ativar:

**Windows**

venv\Scripts\activate

**macOS/Linux**

source venv/bin/activate

**🟩 4. Instalar as dependências essenciais**

Cria um ficheiro requirements.txt (terei um pronto para ti quando começarmos o projeto) e instala:

pip install -r requirements.txt

Se quiseres já preparar tudo, podes instalar estas bases:

pip install ibapi pyside6 numpy pandas pyqtgraph websockets aiohttp pydantic rich loguru

E para os engines:

pip install numba

Para replay e datasets:

pip install pyarrow fastparquet

**🟥 5. Instalar e configurar o IBKR (IB Gateway)**

**Passo 1 — Download do IB Gateway**

https://www.interactivebrokers.com/en/software/ibgateway.php

Escolhe:  
✔ IB Gateway  
❌ não instalar TWS se não for necessário

**Passo 2 — Login**

Existem dois ambientes:

**Live**

* Usa credenciais reais
* Requer Market Data subscriptions

**Paper Trading**

* Requer ativação na conta
* Requer também subscrições de dados

⚠ Importante:  
IBKR *só fornece dados reais para futuros* com subscrição paga.

**🟥 6. Ativar Market Data no IBKR (obrigatório)**

Para ter:

* DOM
* Footprint
* Volume real
* Time & Sales
* Delta real
* Orderbook nativo CME

Precisas ativar minimamente:

**Para futuros CME:**

"US Value Bundle"  
ou  
"CME Bundle Non-Professional"

Custos típicos: 15–30 USD/mês.

**🟦 7. Permitir a API no IB Gateway**

No IB Gateway:

Menu → API → Settings:

✔ Enable ActiveX and Socket Clients  
✔ Read-Only API (desativado se quiser executar ordens)  
✔ Trusted IPs:

127.0.0.1

Porto recomendado para a App:

7496 (Live)

7497 (Paper)

**🟩 8. Preparar dxFeed (OPCIONAL para orderflow premium)**

Se quiseres dados profissionais:

* subscrever dxFeed
* obter token
* ativar WebSocket
* introduzir no ficheiro config/data\_provider.yaml

**🟫 9. Preparar Rithmic (OPCIONAL)**

Para HFT-level data (não execução, só dados):

* Instalar Rithmic R|Trader Pro
* Criar credentials
* Ativar market data
* Configurar porta para a bridge C# caso uses routing interno

**🟦 10. Criar a estrutura base da aplicação**

Criar:

tape\_reading\_app/

config/

core/

ibkr/

engine/

ui/

models/

utils/

tests/

main.py

Depois:

mkdir tape\_reading\_app

cd tape\_reading\_app

**🟩 11. Configurar ficheiros YAML**

Cria:

**config/app.yaml**

* broker ativo
* provider ativo
* símbolos
* resolução (tick/agg)
* UI settings

**config/risk.yaml**

* max daily DD
* max exposure
* max consecutive losses
* kill switch

**config/execution.yaml**

* routing rules
* IBKR settings
* MT5 settings

**🟥 12. Executar a aplicação**

Com ambiente ativo:

python main.py

Se quiseres modo debug:

python main.py --debug

**🟥 13. Troubleshooting básico**

**❌ “Could not connect to TWS/IBGW”**

* verifica porta (7497)
* verifica se IB Gateway está ligado
* desativa firewall
* coloca 127.0.0.1 em “Trusted IPs”

**❌ Recebo ticks mas não recebo DOM**

* não tens subscrição CME
* falta ativar “Depth of Market” no IBKR

**❌ UI congela**

* falta mover data providers para threads/async
* falta usar pyqtgraph em modo OpenGL

**❌ Volume parece estranho**

* estás a usar Forex (não tem volume real)
* tens de usar futuros GC (ouro CME)

**🟩 14. Conclusão**

Este guia garante:

* instalação completa
* ambiente Python funcional
* IBKR pronto para fornecer dados reais
* estrutura da App preparada
* UI e engines prontos a serem construídos

**Já estás apto a iniciar a implementação real da aplicação.**

**📦 SCHEMA FINAL DO PROJETO**

tape\_reading\_app/

│

├── main.py

├── requirements.txt

├── README.md

│

├── config/

│ ├── app.yaml

│ ├── data\_providers.yaml

│ ├── risk.yaml

│ ├── execution.yaml

│ ├── ui.yaml

│ └── \_\_init\_\_.py

│

├── core/

│ ├── event\_bus.py

│ ├── base\_engine.py

│ ├── utils.py

│ ├── clock.py

│ ├── \_\_init\_\_.py

│

├── ibkr/

│ ├── ibkr\_connector.py

│ ├── ibkr\_events.py

│ ├── ibkr\_enums.py

│ ├── ibkr\_utils.py

│ └── \_\_init\_\_.py

│

├── providers/

│ ├── dxfeed\_provider.py

│ ├── rithmic\_provider.py

│ ├── historical\_loader.py

│ └── \_\_init\_\_.py

│

├── models/

│ ├── market\_event.py

│ ├── dom\_snapshot.py

│ ├── delta\_event.py

│ ├── footprint\_cell.py

│ ├── trade\_event.py

│ ├── signal\_event.py

│ ├── execution\_event.py

│ └── \_\_init\_\_.py

│

├── engine/

│ ├── dom\_engine.py

│ ├── delta\_engine.py

│ ├── footprint\_engine.py

│ ├── tape\_engine.py

│ ├── pattern\_detector.py

│ ├── signal\_engine.py

│ ├── risk\_engine.py

│ ├── execution\_router.py

│ ├── replay\_engine.py

│ └── \_\_init\_\_.py

│

├── execution/

│ ├── ibkr\_executor.py

│ ├── mt5\_executor.py

│ ├── multi\_executor.py

│ └── \_\_init\_\_.py

│

├── ui/

│ ├── main\_window.py

│ ├── dom\_panel.py

│ ├── delta\_panel.py

│ ├── footprint\_panel.py

│ ├── tns\_panel.py

│ ├── chart\_widgets.py

│ └── \_\_init\_\_.py

│

├── utils/

│ ├── logger.py

│ ├── serialization.py

│ ├── math\_tools.py

│ ├── threading\_utils.py

│ └── \_\_init\_\_.py

│

└── tests/

├── test\_ibkr.py

├── test\_dom\_engine.py

├── test\_event\_bus.py

├── test\_replay\_engine.py

└── \_\_init\_\_.py

**🧱 AGORA O SCAFFOLDING REAL – ficheiros completos e prontos a colar**

**🟦 main.py**

from core.event\_bus import EventBus

from ibkr.ibkr\_connector import IBKRConnector

from engine.dom\_engine import DOMEngine

from engine.delta\_engine import DeltaEngine

from engine.footprint\_engine import FootprintEngine

from engine.pattern\_detector import PatternDetector

from engine.signal\_engine import SignalEngine

from engine.risk\_engine import RiskEngine

from engine.execution\_router import ExecutionRouter

from ui.main\_window import run\_ui

def main():

print("\nStarting Tape Reading App (Institutional Edition)...")

bus = EventBus()

# Providers

ibkr = IBKRConnector(event\_bus=bus)

# Engines

dom\_engine = DOMEngine(bus)

delta\_engine = DeltaEngine(bus)

footprint\_engine = FootprintEngine(bus)

pattern\_detector = PatternDetector(bus)

signal\_engine = SignalEngine(bus)

risk\_engine = RiskEngine(bus)

router = ExecutionRouter(bus)

# Start UI

run\_ui(bus)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**🟧 Base config files**

**config/app.yaml**

app:

name: "Tape Reading Institutional App"

version: "0.1.0"

provider: "ibkr"

symbol: "GC" # Ouro CME

log\_level: "INFO"

**config/risk.yaml**

risk:

max\_daily\_dd\_percent: 3

max\_consecutive\_losses: 5

max\_exposure\_contracts: 2

kill\_switch\_on\_dd: true

**config/execution.yaml**

execution:

active\_broker: "ibkr"

ibkr:

host: "127.0.0.1"

port: 7497

client\_id: 1

mt5:

path: "C:/Program Files/MetaTrader 5/terminal64.exe"

account: 0

**config/ui.yaml**

ui:

dark\_mode: true

refresh\_rate\_ms: 30

**🟪 Exemplo de ficheiro base para Event Bus**

core/event\_bus.py

class EventBus:

def \_\_init\_\_(self):

self.subscribers = {}

def subscribe(self, event\_type, callback):

self.subscribers.setdefault(event\_type, []).append(callback)

def publish(self, event):

for callback in self.subscribers.get(event.type, []):

callback(event)

**🟩 Exemplo de engine base**

core/base\_engine.py

class BaseEngine:

def \_\_init\_\_(self, bus):

self.bus = bus

self.bus.subscribe("market\_event", self.on\_event)

def on\_event(self, event):

raise NotImplementedError

**🟥 IBKR Connector Skeleton**

ibkr/ibkr\_connector.py

from ibapi.client import EClient

from ibapi.wrapper import EWrapper

from ibapi.contract import Contract

class IBKRConnector(EWrapper, EClient):

def \_\_init\_\_(self, event\_bus):

EClient.\_\_init\_\_(self, self)

self.bus = event\_bus

self.connect("127.0.0.1", 7497, clientId=1)

def tickPrice(self, reqId, tickType, price, attrib):

self.bus.publish({"type": "market\_event", "tick": price})

**🟦 UI skeleton**

ui/main\_window.py

from PySide6.QtWidgets import QApplication, QMainWindow

def run\_ui(event\_bus):

app = QApplication([])

win = QMainWindow()

win.setWindowTitle("Institutional Tape Reading App")

win.show()

app.exec()

**🎉 CONCLUSÃO**

Este scaffolding já está:

✔ Profissional  
✔ Escalável  
✔ Modular  
✔ Preparado para IBKR, dxFeed, Rithmic  
✔ Preparado para UI institucional  
✔ Preparado para multi-broker execution  
✔ Pronto para começar a preencher os módulos

**📘 ROADMAP OFICIAL DE IMPLEMENTAÇÃO (A → Z)**

***Construção completa da Tape Reading App com Python + IBKR + UI***

**🟥 FASE 1 — NÚCLEO DO SISTEMA (FOUNDATION) (FECHADO)**

**Objetivo: levantar a espinha dorsal da app — sem isto nada funciona.**

**1. event\_bus.py**

✔ Sistema pub/sub  
✔ Base para todos os motores  
🔑 **PRIMEIRO ficheiro que vamos criar**

**2. market\_event.py (models)**

✔ Estrutura unificada para transportar eventos: tick, dom, trade  
✔ Tipagem forte (Pydantic)

**3. ibkr\_connector.py (mínimo viável)**

✔ Ligação ao IBKR  
✔ Receção de tick-by-tick (preço)

**4. Teste integrador:**

* ligar IB Gateway
* correr main.py minimal
* receber ticks

🎯 **Meta da Fase 1:**

Receber dados reais do mercado via IBKR.

**🟧 FASE 2 — DATA LAYER (PROVIDERS COMPLETOS) (FECHADO)**

Agora expandimos para dados institucionais reais.

**5. ibkr\_events.py**

✔ Formatadores para DOM, trades, tick types IBKR

**6. Expansão de ibkr\_connector.py**

✔ DOM (level 1)  
✔ Market Depth (level 2)  
✔ Time & Sales  
✔ Reconexão automática

**7. dxfeed\_provider.py (se aplicável)**

✔ WebSocket  
✔ Depth + Trades + Aggressive flow

**8. historical\_loader.py**

✔ Load datasets  
✔ Pré-loading DOM/Delta/Footprint para replay

🎯 **Meta da Fase 2:**

A app já tem fontes de dados profissionais completas.

**🟨 FASE 3 — ENGINES (O CÉREBRO DO SISTEMA)**

Cada engine consome eventos e transforma em informação institucional.

**9. dom\_engine.py**

✔ Normalizar depth updates  
✔ Detectar pulling/stacking  
✔ Calcular liquidity map

**10. delta\_engine.py**

✔ Delta tick-by-tick  
✔ CVD  
✔ Burst detection

**11. footprint\_engine.py**

✔ Agregação bid/ask por preço  
✔ Imbalances  
✔ Footprint bars

**12. tape\_engine.py**

✔ Times & Sales clustering  
✔ Speed of tape  
✔ Big prints

🎯 **Meta da Fase 3:**

Todos os motores de orderflow funcionam autonomamente.

**🟩 FASE 4 — PADRÕES INSTITUCIONAIS (INTELIGÊNCIA MANUAL)**

**13. pattern\_detector.py**

✔ Absorção  
✔ Spoofing  
✔ Failed breakout  
✔ Delta divergences  
✔ Bursts  
✔ Liquidity vacuum

🎯 **Meta:**

O sistema entende microestrutura tal como um trader institucional.

**🟦 FASE 5 — ML PIPELINE (opcional mas recomendado)**

**14. ml\_feature\_builder.py**

**15. ml\_sequence\_builder.py**

**16. model\_loader.py**

**17. ml\_inference.py**

🎯 Meta:

Ligação entre orderflow → features → modelo → probabilidade de trade.

**🟫 FASE 6 — DECISÃO & EXECUÇÃO**

**18. signal\_event.py**

**19. signal\_engine.py**

✔ Junta padrões + ML  
✔ Regras finais

**20. risk\_engine.py**

✔ Daily DD  
✔ Exposure  
✔ Hard limits  
✔ Kill switch

**21. execution\_router.py**

✔ Decide para onde enviar a ordem (IBKR / MT5)

**22. ibkr\_executor.py**

✔ Executar ordens reais via IBKR

**23. mt5\_executor.py**

✔ Executar ordens no MetaTrader 5

🎯 Meta:

Bot capaz de enviar ordens nos 2 brokers.

**🟥 FASE 7 — UI INSTITUCIONAL (PYSIDE6)**

**24. main\_window.py**

**25. dom\_panel.py**

**26. delta\_panel.py**

**27. footprint\_panel.py**

**28. tns\_panel.py**

**29. live\_charts.py**

🎯 Meta:

DOM + Footprint + Delta + Tape em tempo real.

**🟫 FASE 8 — REPLAY + HISTÓRICO**

**30. replay\_engine.py**

**31. dataset\_writer.py**

**32. loader\_replay\_mode.py**

🎯 Meta:

A app pode ser usada como simulador institucional.

**🟪 FASE 9 — DEPLOY E HARDENING**

**33. Scripts de watchdog**

**34. Logging avançado**

**35. Configs de produção**

**36. Testes unitários e integração**

**37. Packaging**

🎯 Meta:

App estável para uso real.

**🟦 RESUMO DO ROADMAP (A → Z)**

Aqui está a ordem exata dos ficheiros a serem construídos:

**📌 ORDEM FINAL DOS FICHEIROS A FAZER**

**Fase 1 – Foundation**

1. **event\_bus.py xx**
2. **market\_event.py xx**
3. **ibkr\_connector.py (versão simples) xx**
4. Teste real com IB Gateway skkipped

**Fase 2 – Providers**

1. ibkr\_events.py
2. ibkr\_connector.py (versão completa)
3. dxfeed\_provider.py
4. historical\_loader.py

**Fase 3 – Engines**

1. dom\_engine.py
2. delta\_engine.py
3. footprint\_engine.py
4. tape\_engine.py

**Fase 4 – Patterns**

1. pattern\_detector.py

**Fase 5 – ML (opcional)**

1. ml\_feature\_builder.py
2. ml\_sequence\_builder.py
3. model\_loader.py
4. ml\_inference.py

**Fase 6 – Execução**

1. signal\_event.py
2. signal\_engine.py
3. risk\_engine.py
4. execution\_router.py
5. ibkr\_executor.py
6. mt5\_executor.py

**Fase 7 – UI**

1. main\_window.py
2. dom\_panel.py
3. delta\_panel.py
4. footprint\_panel.py
5. tns\_panel.py
6. live\_charts.py

**Fase 8 – Replay**

1. replay\_engine.py
2. dataset\_writer.py
3. loader\_replay\_mode.py

**Fase 9 – Finalização**

1. watchdog.py
2. logging\_config.py
3. production\_settings.py
4. tests/
5. packaging (installer)

**📦 SCHEMA FINAL DO PROJETO**

tape\_reading\_app/

│

├── main.py

├── requirements.txt

├── README.md

│

├── config/

│ ├── app.yaml

│ ├── data\_providers.yaml

│ ├── risk.yaml

│ ├── execution.yaml

│ ├── ui.yaml

│ └── \_\_init\_\_.py

│

├── core/

│ ├── event\_bus.py

│ ├── base\_engine.py

│ ├── utils.py

│ ├── clock.py

│ ├── \_\_init\_\_.py

│

├── ibkr/

│ ├── ibkr\_connector.py

│ ├── ibkr\_events.py

│ ├── ibkr\_enums.py

│ ├── ibkr\_utils.py

│ └── \_\_init\_\_.py

│

├── providers/

│ ├── dxfeed\_provider.py

│ ├── rithmic\_provider.py

│ ├── historical\_loader.py

│ └── \_\_init\_\_.py

│

├── models/

│ ├── market\_event.py

│ ├── dom\_snapshot.py

│ ├── delta\_event.py

│ ├── footprint\_cell.py

│ ├── trade\_event.py

│ ├── signal\_event.py

│ ├── execution\_event.py

│ └── \_\_init\_\_.py

│

├── engine/

│ ├── dom\_engine.py

│ ├── delta\_engine.py

│ ├── footprint\_engine.py

│ ├── tape\_engine.py

│ ├── pattern\_detector.py

│ ├── signal\_engine.py

│ ├── risk\_engine.py

│ ├── execution\_router.py

│ ├── replay\_engine.py

│ └── \_\_init\_\_.py

│

├── execution/

│ ├── ibkr\_executor.py

│ ├── mt5\_executor.py

│ ├── multi\_executor.py

│ └── \_\_init\_\_.py

│

├── ui/

│ ├── main\_window.py

│ ├── dom\_panel.py

│ ├── delta\_panel.py

│ ├── footprint\_panel.py

│ ├── tns\_panel.py

│ ├── chart\_widgets.py

│ └── \_\_init\_\_.py

│

├── utils/

│ ├── logger.py

│ ├── serialization.py

│ ├── math\_tools.py

│ ├── threading\_utils.py

│ └── \_\_init\_\_.py

│

└── tests/

├── test\_ibkr.py

├── test\_dom\_engine.py

├── test\_event\_bus.py

├── test\_replay\_engine.py

└── \_\_init\_\_.py