Public web edition | PDF download included | Balance Labs playbook | Beginner-friendly explanations

Python Backtest Handbook

คู่มือเข้าใจ product flow ของ Balance Labs แบบสอนง่าย ตั้งแต่ prompt, strategy spec, backtest, Monte Carlo, ranking, artifact ไปจนถึง deploy

คู่มือเชิงปฏิบัติที่อธิบายพื้นฐานซึ่งคนใช้ Balance Labs ควรรู้ เพื่อเปลี่ยน prompt ให้กลายเป็น strategy ที่อ่านผลเป็น ตัดสินใจเป็น และพร้อมต่อยอดสู่การ share หรือ deploy ได้จริง

Updated 31 March 2026 | Reading time 45-60 min | Web route /ebook/python-ai-trading-systems

Download PDF | Open web edition

หนังสือเล่มนี้จะช่วยอะไร

- เห็นภาพ product flow ของ Balance Labs ตั้งแต่ prompt -> spec -> execute -> ranking/share -> deploy
- เข้าใจ concept สำคัญอย่าง data hygiene, walk-forward, robustness และ Monte Carlo แบบภาษาคน
- อ่าน scorecard, artifact และ trade evidence แล้วตัดสินใจได้ว่าควร iterate, share หรือหยุด
- ใช้ AI และ Labs Chat เป็นตัวเร่งงาน โดยยังคุม assumptions, cost และ risk ได้

At a glance

CHAPTERS

12

ครอบคลุมตั้งแต่ architecture ถึง go-live checklist

CODE BLOCKS

11

ตัวอย่างโค้ดที่พอให้เอาไปต่อยอดเป็น pipeline จริง

INFOGRAPHICS

12

แผนภาพสรุป logic, architecture และ validation ให้เห็นเป็นภาพรวมเร็วขึ้น

VALIDATION LAYERS

6

walk-forward, cost stress, regime split, Monte Carlo และ decision gates

GOAL

Prompt -> Deploy

ผูกบทเรียนกับ Labs Chat, Execute, Ranking, Share และ Deploy

สารบัญ

Foundation

Chapter 1

นิยาม backtest ที่เชื่อถือได้ใน Balance Labs ต้องเริ่มจากคำถาม ไม่ใช่เริ่มจากโค้ด

Balance Labs ไม่ได้ทำให้โจทย์ที่ฟุ้งกลายเป็นโจทย์ที่ดีโดยอัตโนมัติ backtest ที่ดีไม่ได้ตอบแค่ว่า strategy ทำกำไรหรือไม่ แต่ต้องตอบได้ด้วยว่ามันกำไรภายใต้สมมติฐานอะไร, แพ้ใน regime ไหน, อ่อนไหวกับต้นทุนแค่ไหน และมีเหตุผลพอจะ iterate ต่อหรือหยุดตรงนี้หรือไม่

Figure: Research map before code

เริ่มจากโจทย์วิจัย, market universe, timeframe, execution assumptions และ risk budget ก่อนค่อยลง code เพื่อไม่ให้ backtest กลายเป็นการ optimize คำถามที่ยังนิยามไม่จบ

- ถามให้ชัดว่า strategy ชนะใน regime ไหน
- ล็อก execution model ก่อนดูผลกำไร
- ตั้ง acceptance criteria ให้ตัดสินใจ iterate หรือหยุดได้

งาน backtest ส่วนใหญ่พังเพราะเริ่มจาก indicator หรือ prompt ที่ดูน่าตื่นเต้นเกินไป แล้วค่อยมาเติมกติกาหลังบ้านทีหลัง เช่น ขนาด position, ค่าคอมมิชชั่น, เวลาที่อนุญาตให้เปิดสถานะ หรือข้อจำกัดเรื่อง liquidity วิธีนี้ทำให้ผลลัพธ์ดูดีบนกระดาษ แต่ใช้ตัดสินใจจริงไม่ได้

แปลเป็นภาษาของ Balance Labs

Prompt คือจุดรับโจทย์ ไม่ใช่คำตอบสุดท้าย คุณค่าจริงเกิดเมื่อโจทย์ถูกแปลงเป็น spec, รันผ่าน Execute, อ่าน scorecard, แล้วตัดสินใจต่อว่าจะ refine, share หรือ deploy

ขั้นต่ำที่ต้องนิยามก่อนเริ่ม

- instrument universe: จะเทรดอะไรบ้าง หุ้น, ฟิวเจอร์ส, FX, crypto หรือแค่สินทรัพย์เดียว
- timeframe: intraday, hourly, daily และกฎการ resample ที่แน่นอน
- session assumptions: เปิดตลาดเมื่อไร, หยุดถือข้ามคืนหรือไม่, ปิดสถานะก่อนข่าวหรือไม่
- execution assumptions: เข้าที่ close bar, next open, midpoint หรือ limit fill
- transaction costs: commission, spread, slippage, funding, borrow fee
- risk budget: position sizing, leverage cap, max concurrent exposure, stop logic
- acceptance criteria: metric อะไรที่ต้องผ่าน เช่น profit factor, drawdown, turnover, Sharpe, expectancy

ตัวอย่างกรอบคำถามวิจัย

คำถาม	ตัวอย่าง	เหตุผล
Strategy edge คืออะไร	mean reversion หลัง breakout fail ใน BTC 1H	ถ้าอธิบาย edge ไม่ได้ การ iterate จะกลายเป็นการสุ่ม
จะชนะที่ไหน	regime ผันผวนสูงแต่ไม่มี trend ต่อเนื่อง	ช่วยแยก market condition ที่ strategy ควรเปิดใช้งาน
จะล้มเหลวที่ไหน	trend day ที่ลากต่อโดยไม่มี retrace	ช่วยวางตัวกรองหรือ kill switch
ต้นทุนมีผลแค่ไหน	edge หายเมื่อ slippage เกิน 6 bps	ช่วยประเมินความ deployable ของผลลัพธ์

หลักคิดที่ควรเลิกใช้

คำว่า “รันก่อน เดี๋ยวค่อยดูทีหลัง” ใช้ได้กับ prototype สั้น ๆ แต่ใช้ไม่ได้กับระบบวิจัยที่ต้องอธิบายให้ทีม, ลูกค้า หรือ future-you เข้าใจ ถ้าโจทย์ไม่ชัด ต่อให้ engine เร็วแค่ไหน คุณก็แค่ได้คำตอบเร็วขึ้นสำหรับคำถามที่ผิด

Key takeaway

ก่อนกดให้ AI ช่วยเขียนหรือกด Execute ใน Labs ให้ล็อกโจทย์, market universe, execution assumptions และ acceptance criteria ให้ชัด ไม่อย่างนั้นคุณจะ optimize สิ่งที่ไม่เคยนิยาม

Architecture

Chapter 2

โครงสร้างโปรเจกต์ที่ดีทำให้การวิจัยไม่พังเมื่อ strategy ตัวแรกพอใช้ได้

ในโลกของ Balance Labs คุณอาจเห็นเป็นหน้า Chat, Execute, Ranking และ Deploy แต่ข้างหลังทั้งหมดก็คือการแยกชั้น responsibility เดิม ๆ ให้ชัด ถ้า layer เหล่านี้ปนกัน วันหนึ่งการทดลองที่ดูง่ายจะกลายเป็น maintenance nightmare

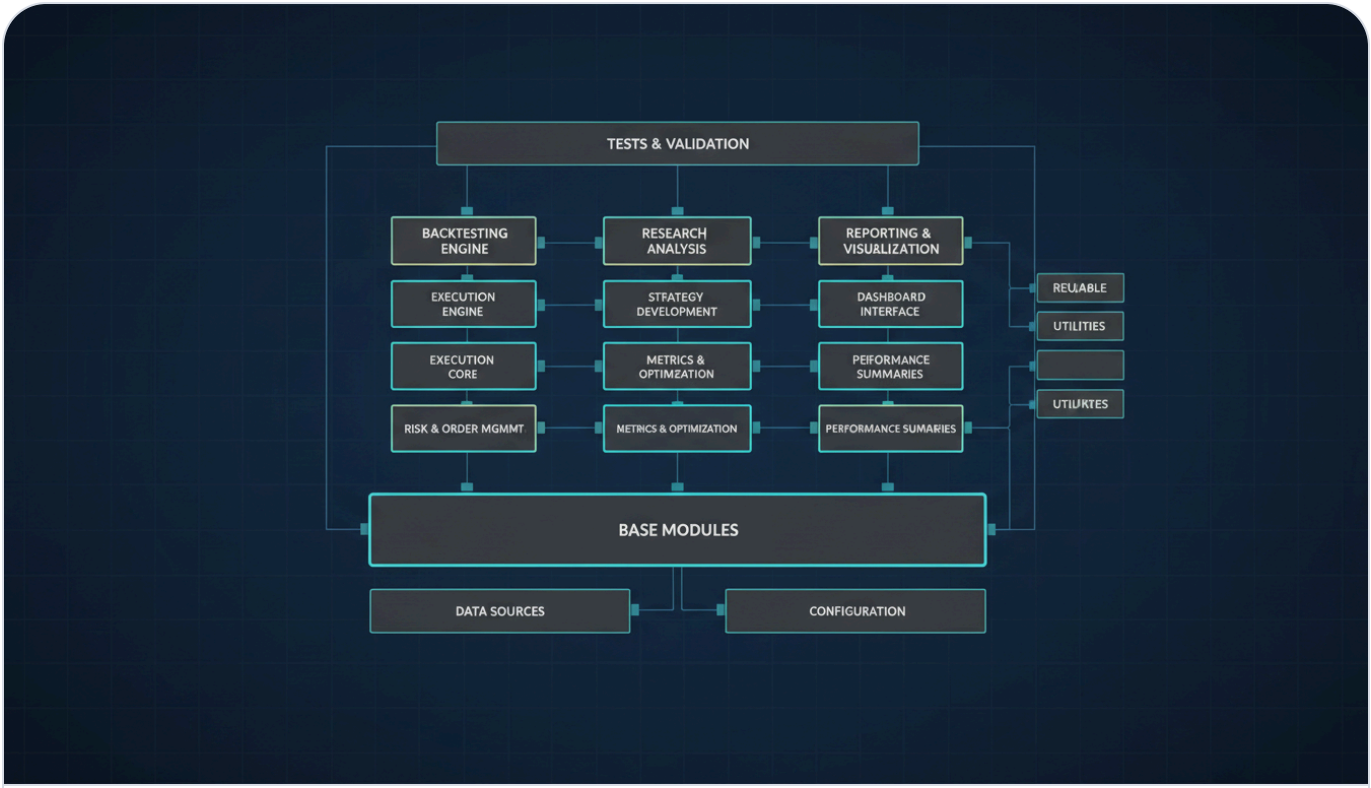

Figure: Project architecture that survives iteration

ภาพนี้สรุป stack ขั้นต่ำของโปรเจกต์ backtest ที่แยก config, data, strategy logic, engine, metrics และ reporting ออกจากกันชัดเจน เพื่อให้เพิ่ม strategy ใหม่โดยไม่ทำให้ codebase พังตาม

- notebook ใช้สำรวจ ไม่ใช่เก็บ logic หลัก
- engine กับ reporting ต้องแยก responsibility
- ทุก run ต้องย้อนกลับไปหา config และ data path ได้

โครงสร้างโปรเจกต์ขั้นต่ำที่แนะนำ project-tree.txt

```
python-backtest/
  pyproject.toml
  README.md
```

```
data/
  raw/
  processed/
configs/
  btc_1h_mean_reversion.yaml
src/
  data_loader.py
  indicators.py
  strategy_rules.py
  engine.py
  execution.py
  metrics.py
  reporting.py
  walk_forward.py
  optimize.py
  types.py
reports/
  runs/
tests/
  test_data_loader.py
  test_engine.py
  test_metrics.py
notebooks/
  exploration_only.ipynb
```

ให้ notebook เป็นที่ทดลองไอเดีย ไม่ใช่ที่เก็บ logic หลักของระบบ

หลักการแบ่งโมดูล

- data_loader.py **รับผิดชอบ** ingest, timezone, column naming, missing-bar checks
- strategy_rules.py **สร้าง** signal **โดยไม่รู้เรื่องเงิน**, position, fees
- execution.py **แปลง** signal **เป็น** fill price, quantity, fees **และ** position state
- engine.py **ร้อยทุกอย่างเข้าด้วยกันและสร้าง** trade ledger
- metrics.py **เปลี่ยน** ledger **เป็น** scorecard **ที่เปรียบเทียบ** run **ต่าง ๆ ได้**
- reporting.py **สร้าง** HTML, CSV, JSON summary, equity curve **และ** chart caption
- walk_forward.py **และ** optimize.py **ต้องเรียก** engine **ผ่าน** interface **เดียวกับ** run **ปกติ**

ถ้าโครงสร้างดี คุณจะสามารถเปลี่ยน strategy **โดยไม่แก้** engine, **เปลี่ยน** execution model **โดยไม่แก้** indicator, **และเพิ่ม** report **โดยไม่ยุ่งกับสัญญาณ ถ้าเปลี่ยนทีหนึ่งแล้วกระทบทุกไฟล์ แปลว่ายังแยก** layer **ไม่ขาด**

map นี้ผูกกับ product อย่างไร

Labs Chat ช่วย intake และ scaffold, Execute คือ engine + evidence layer, Ranking/Share คือ decision layer, ส่วน Deploy คือ production handoff ถ้าเข้าใจ layer เหล่านี้ คุณจะใช้ product เป็นมากกว่าแค่กล่อง prompt

เลือกสแต็กให้เหมาะกับเป้าหมาย

สำหรับงานส่วนใหญ่ pandas + numpy + matplotlib หรือ plotly เพียงพอ ถ้าต้องการ event-driven simulation ที่ซับซ้อนมากค่อยพิจารณา backtrader, vectorbt, zipline-style framework หรือเขียน engine เฉพาะทางเอง อย่าเริ่มจาก framework ใหญ่เพราะมันดูครบ ถ้ายังไม่รู้ว่าจริง ๆ ต้องการอะไร

Key takeaway

แยก responsibility ให้ชัดตั้งแต่วันแรก: intake/prompt, config, data, indicators, strategy rules, engine, metrics, reports และ deployment handoff

Data

Chapter 3

Data hygiene สำคัญกว่า strategy sophistication ในช่วงแรกเกือบทุกครั้ง

ข้อมูลที่มี timezone เพี้ยน, candle หาย, duplicate bars, split adjustment ผิด หรือ indicator ใช้ source price ไม่ตรงกัน จะทำให้คุณเสียเวลา debug logic ทั้งที่ปัญหาอยู่ต้นน้ำ

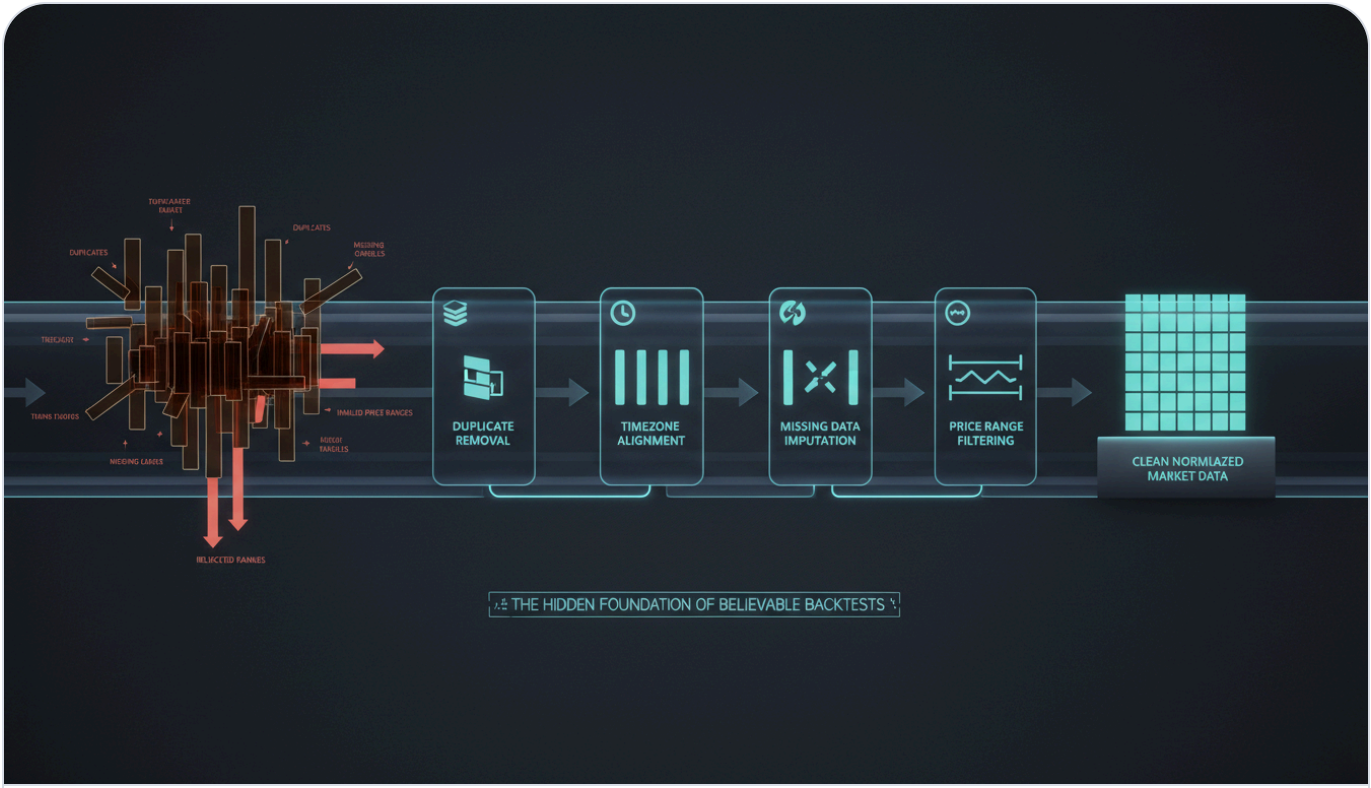

Figure: Data hygiene pipeline

ก่อนคุยเรื่อง edge ต้องทำให้ data path สะอาดก่อนเสมอ ตั้งแต่ ingest, timezone normalization, duplicate removal, OHLC validation ไปจนถึงการ publish dataset ที่ strategy ทุกตัวใช้ร่วมกัน

- timezone และ bar boundary ต้องนิยามครั้งเดียว
- fail ดังเมื่อ data shape ผิดหรือมีช่องโหว่
- strategy ทุกตัวควรใช้ normalized dataset ชุดเดียวกัน

สิ่งที่ data pipeline ต้องล็อกให้ชัด

- timezone เดียวตลอด pipeline
- ชื่อคอลัมน์มาตรฐาน เช่น open/high/low/close/volume
- duplicate index ต้องไม่มี
- missing bar ต้องรู้ว่าเจอเมื่อไรและตัดสินใจอย่างไร

- source ของราคา indicator ต้องชัด เช่น close, hlc3, ohlc4
- corporate actions, contract rolls, funding และ session boundary ต้องสอดคล้องกับตลาดที่ใช้

ตัวอย่างฟังก์ชัน normalize OHLCV src/data_loader.py

```python
from __future__ import annotations

import pandas as pd


REQUIRED_COLUMNS = ["open", "high", "low", "close", "volume"]


def normalize_ohlcv(df: pd.DataFrame, timezone: str = "UTC") -> pd.DataFrame:
    frame = df.copy()
    frame.columns = [str(col).strip().lower() for col in frame.columns]

    missing = [col for col in REQUIRED_COLUMNS if col not in frame.columns]
    if missing:
        raise ValueError(f"missing required columns: {missing}")

    if not isinstance(frame.index, pd.DatetimeIndex):
        raise TypeError("index must be a DatetimeIndex")

    if frame.index.tz is None:
        frame.index = frame.index.tz_localize(timezone)
    else:
        frame.index = frame.index.tz_convert(timezone)

    frame = frame[~frame.index.duplicated(keep="last")].sort_index()

    if frame[REQUIRED_COLUMNS].isna().any().any():
        raise ValueError("price data contains NaN values")

    invalid_range = (frame["high"] < frame["low"]).any()
    invalid_open = ((frame["open"] > frame["high"]) | (frame["open"] < frame["low"])).any()
    invalid_close = ((frame["close"] > frame["high"]) | (frame["close"] < frame["low"])).any()
    if invalid_range or invalid_open or invalid_close:
        raise ValueError("ohlc range validation failed")

    return frame
```

ฟังก์ชันนี้ตั้งใจให้ fail ดัง ๆ เมื่อ data ไม่ผ่าน แทนที่จะปล่อยให้ error ไปโผล่ในผลลัพธ์ปลายทาง

อาการผิดปกติที่พบประจำ

อาการ	สาเหตุที่เป็นไปได้	ผลกระทบต่อ backtest
equity curve กระโดดแปลก ๆ	duplicate bar หรือ split adjustment ผิด	metrics เพี้ยนทั้งชุด
entry ช้าหรือเร็วหนึ่งแท่ง	timezone หรือ resample offset ผิด	strategy เหมือนใช้ข้อมูลอนาคต
win rate ดูดีเกินจริง	mid-price fill แต่ตลาดจริงเข้า at ask/bid	ประเมิน edge สูงเกินจริง
drawdown ต่ำผิดปกติ	ใช้ close-to-close แต่จริงควรโดน stop intrabar	risk understate

แนวทางที่ควรยึด

ใช้ fixture data ชุดเล็ก ๆ ที่เช็กด้วยสายตาได้ แล้วเขียน unit tests กับมัน ก่อนจะปล่อยให้ระบบวิ่งกับข้อมูลหลายแสนแถว การมี test data ชุดเล็กช่วยให้คุณพิสูจน์ได้ว่า entry, exit, stop, position flip และ fee logic ทำงานตรงตามที่ตั้งใจจริง

Key takeaway

สร้างขั้นตอน normalize ข้อมูลครั้งเดียวให้แน่น แล้วบังคับให้ทุก strategy ใช้ path เดียวกัน

Specification | Chapter 4

Strategy specification ควรอยู่ในรูปที่ทดสอบได้ ไม่ใช่คำอธิบายกว้าง ๆ

ปัญหาของหลายทีมไม่ใช่เขียนโค้ดไม่เก่ง แต่เป็นการส่ง requirement ให้ทีมเขียนหรือให้ AI ในรูปที่คลุมเครือเกินไป เช่น “หาแนวโน้มแล้วเข้าเมื่อ momentum ดี” ซึ่งไม่สามารถตรวจสอบได้

Figure: Strategy specification anatomy

strategy ที่ส่งต่อให้ทีมหรือ AI ได้ต้องแตกเป็นชิ้นส่วนที่ทดสอบได้ เช่น regime, setup, trigger, risk, exit และ invalidation ไม่ใช่คำอธิบายกว้าง ๆ ที่ตีความได้หลายแบบ

- แยกส่วนที่ optimize ได้ออกจากสมมติฐานหลัก
- เปลี่ยน trigger โดยไม่พัง risk logic
- ทุกส่วนต้องโยงไปสู่ test case ได้

เมื่อ specification ชัด คุณจะสามารถเปลี่ยนแค่ส่วนที่ต้องการทดลอง เช่น เปลี่ยน trigger โดยไม่ยุ่งกับ sizing หรือเปลี่ยน stop logic โดยไม่ทำให้ regime filter สับสน ที่สำคัญคือทุกคนในทีมจะคุยด้วยภาษาชุดเดียวกัน

ตัวอย่าง config object สำหรับ strategy — src/types.py

```python
from dataclasses import dataclass


@dataclass(frozen=True)
class StrategyConfig:
    symbol: str
    timeframe: str
    fast_window: int
    slow_window: int
    atr_window: int
    atr_stop_multiple: float
    risk_per_trade: float
    max_holding_bars: int
    fee_bps: float
    slippage_bps: float
    allow_short: bool = False
```

เก็บ parameter ที่เปลี่ยนได้ไว้ใน config เพื่อให้ optimize, walk-forward และ reporting เรียกใช้รูปแบบเดียวกัน

แม่แบบการเขียนกติกา strategy

- Regime: ตลาดแบบไหนถึงอนุญาตให้หา setup
- Setup: รูปแบบเชิงโครงสร้างที่ทำให้ strategy สนใจ bar ชุดนั้น
- Trigger: เงื่อนไขเข้าแบบ exact เช่น close cross above fast SMA
- Risk: stop, take profit, trailing logic, timeout, max position
- Exit: ปิดอย่างไรเมื่อ edge หมดหรือ risk event เกิด
- Invalidation: สถานการณ์ไหนบอกว่า signal นี้ต้องไม่ถูกนับหรือไม่ควร deploy

สิ่งที่ไม่ควรส่งให้ AI

อย่าส่งข้อความลอย ๆ เช่น “เขียน strategy แนว breakout ที่กำไรดี” แล้วหวังว่า model จะเดา execution assumptions, cost model และ metrics ให้ถูก สิ่งที่ model ทำได้ดีคือเร่งงานภายใต้ spec ที่วัดผลได้ ไม่ใช่ตีความโจทย์ธุรกิจแทนคุณ

Key takeaway

แยก strategy ออกเป็น regime filter, setup, trigger, risk, exit และ invalidation อย่างชัดเจน

Engine

Chapter 5

แกนหลักของ backtest engine ต้องแยก signal ออกจาก execution และ ledger

หลายคนเริ่มจากการคูณ returns กับ signal ตรง ๆ ซึ่งใช้สำรวจเบื้องต้นได้ แต่พอมี stop, time-based exit, partial fill หรือ reversal logic โมเดลแบบนี้จะเริ่มหลอกคุณทันที

Figure: Signal to execution to ledger flow

engine ที่เชื่อถือได้ต้องแยก signal, execution simulation และ trade ledger ออกจากกัน เพื่อให้ debug ได้ว่า strategy ตั้งใจอะไร, ตลาด fill อย่างไร และผลลัพธ์สุดท้ายเกิดจากอะไรแน่

- signal คือ intent ไม่ใช่ผล fill
- execution layer แปลง intent เป็นราคาและจำนวน
- ledger คือแหล่งจริงของ trade history และ metrics

signal บอกความตั้งใจของ strategy ส่วน execution เป็นตัวแปลงความตั้งใจนั้นให้เป็น fill price และ quantity ภายใต้ข้อจำกัดตลาด จากนั้น ledger ค่อยเก็บว่าเปิดเมื่อไร ปิดเมื่อไร ได้หรือเสียเท่าไร ถ้าสามชั้นนี้ผูกกันแน่นเกินไป การ debug จะยากมาก

ตัวอย่าง engine loop แบบเรียบง่าย src/engine.py

```
from __future__ import annotations

from dataclasses import dataclass


@dataclass
class Trade:
    entry_time: object
    entry_price: float
    qty: float
    side: int
    stop_price: float
    exit_time: object | None = None
    exit_price: float | None = None
    fee_paid: float = 0.0


def run_backtest(frame, signal, config, execution_model):
    cash = 100_000.0
    position = None
    trades = []

    for timestamp, row in frame.iterrows():
        desired_side = int(signal.loc[timestamp])

        if position is None and desired_side != 0:
            fill = execution_model.entry_fill(row["open"], desired_side, config.slippage_bps)
            qty = execution_model.position_size(
                cash=cash,
                price=fill,
                risk_per_trade=config.risk_per_trade,
                stop_multiple=config.atr_stop_multiple,
                atr_value=row["atr"],
            )
            fee = execution_model.fee(fill, qty, config.fee_bps)
            position = Trade(
                entry_time=timestamp,
                entry_price=fill,
                qty=qty,
                side=desired_side,
                stop_price=execution_model.stop_price(fill, desired_side, row["atr"], config.atr_stop_
                fee_paid=fee,
            )
            cash -= fee
            continue
```

```
        if position is not None:
            should_exit = execution_model.should_exit(
                row=row,
                side=position.side,
                stop_price=position.stop_price,
                desired_side=desired_side,
            )
            if should_exit:
                fill = execution_model.exit_fill(row["close"], position.side, config.slippage_bps)
                fee = execution_model.fee(fill, position.qty, config.fee_bps)
                position.exit_time = timestamp
                position.exit_price = fill
                position.fee_paid += fee
                cash += execution_model.realized_pnl(position)
                cash -= fee
                trades.append(position)
                position = None

    return trades, cash
```

ตัวอย่างนี้ตั้งใจให้เห็น separation ของ signal, execution model และ ledger มากกว่าจะเป็น engine ที่ครบทุก edge case

trade lifecycle ที่ต้องนิยามให้ครบ

- entry timing: ใช้ราคาแท่งถัดไปหรือแท่งปัจจุบัน
- re-entry: อนุญาตกลับเข้าใน bar เดียวกันหรือไม่
- flip logic: long เป็น short ทันทีหรือบังคับ flat ก่อน
- partial exit: รองรับหรือไม่
- stop precedence: ถ้า bar เดียวชนทั้ง stop และ target จะตัดสินใจอย่างไร
- end-of-sample handling: ปิดสถานะค้างที่ bar สุดท้ายหรือไม่นับ

vectorized backtest ยังมีประโยชน์

สำหรับ exploration แรก ๆ การคำนวณแบบ vectorized ยังเร็วและมีประโยชน์มาก แต่เมื่อกลยุทธ์เริ่มมีสถานะภายในระบบมากขึ้น เช่น cooldown, pyramiding, timed exit, intrabar stop หรือ partial fill ให้เตรียมย้ายสู่ stateful engine เพื่อหลีกเลี่ยงการโกงตัวเองด้วยสมมติฐานแฝง

Key takeaway

engine ที่ใช้งานจริงควรมีอย่างน้อยสามชั้น: signal series, position state machine และ trade ledger

Execution

Chapter 6

ค่าคอมมิชชั่น, slippage, spread และ sizing คือที่ซ่อนของความจริง

strategy ที่ดูดีมากก่อนหักต้นทุนมักไม่มีค่ามากนักในโลกจริง ยิ่งเทรดถี่ ต้นทุนยิ่งไม่ใช่ footnote แต่เป็นส่วนหนึ่งของ edge

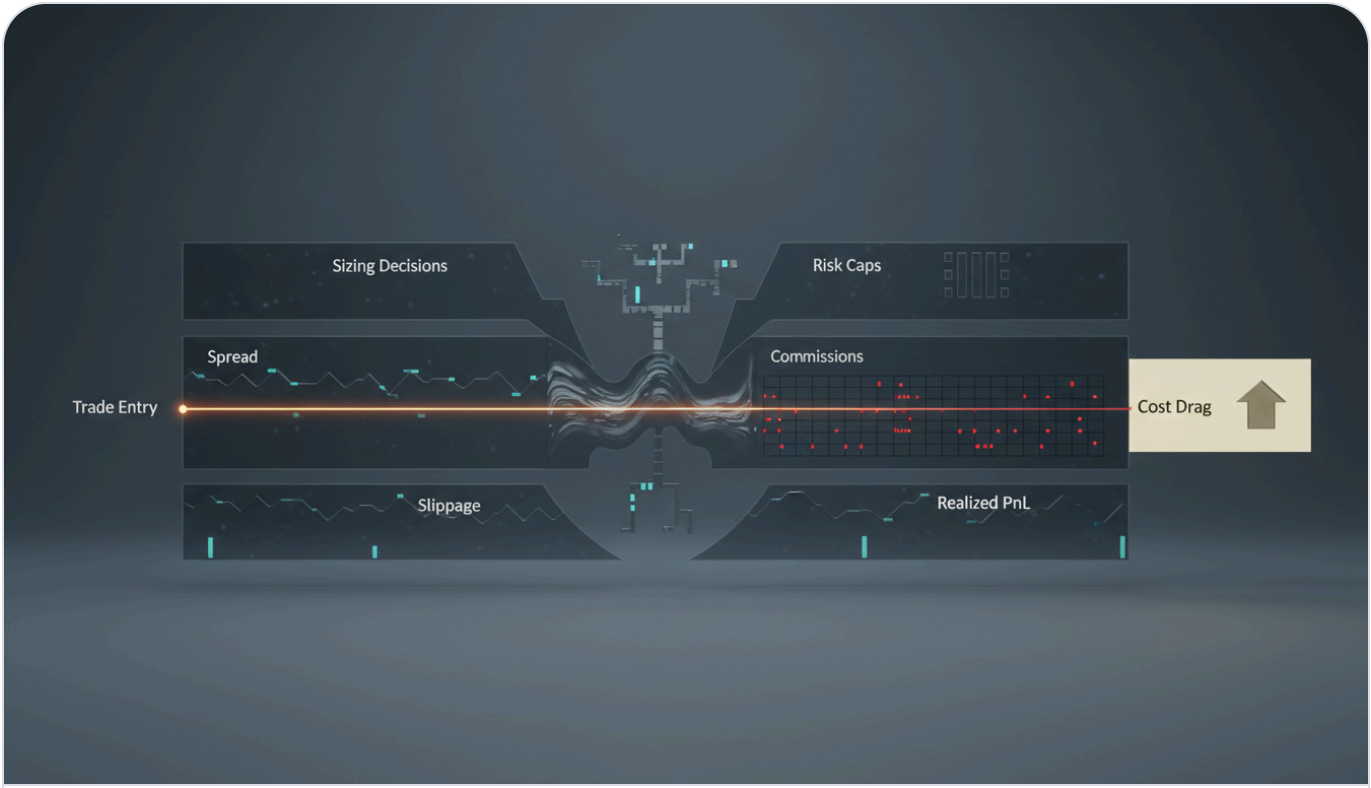

Figure: Cost stack and sizing model

รูปนี้รวมต้นทุนที่มักถูกลืมไว้ในบรรทัดเดียว ทั้ง commission, spread, slippage, funding และ sizing model เพื่อเตือนว่ากำไรสุทธิไม่ได้มาจาก signal อย่างเดียว แต่เกิดจากการอยู่รอดหลังหัก friction แล้ว

- gross PnL ไม่มีความหมายถ้า net พังเมื่อหัก cost
- sizing เปลี่ยนภาพ drawdown มากกว่าที่หลายทีมคิด
- cost stress เป็นเงื่อนไขขั้นต่ำก่อนพูดเรื่อง deploy

องค์ประกอบของ execution model ที่ควรมี

องค์ประกอบ	ตัวอย่าง	ทำไมสำคัญ
Commission	0.04% ต่อข้าง	กระทบ expectancy โดยตรง

องค์ประกอบ	ตัวอย่าง	ทำไมสำคัญ
Slippage	2-10 bps ตาม volatility	ทำให้ edge ในตลาดเร็วหายง่ายมาก
Spread	เข้า ask ออก bid	จำเป็นมากสำหรับ FX และ crypto
Funding / borrow	ถือ short ข้ามวัน	กระทบ swing และ stat arb
Sizing	fixed fractional / ATR risk	ควบคุม drawdown และ comparability

ตัวอย่าง helper สำหรับ fill และต้นทุน src/execution.py

```
def apply_slippage(price: float, side: int, slippage_bps: float) -> float:
    multiplier = 1 + (slippage_bps / 10_000.0) * side
    return price * multiplier


def commission(notional: float, fee_bps: float) -> float:
    return notional * (fee_bps / 10_000.0)


def fixed_fractional_qty(cash: float, risk_per_trade: float, entry: float, stop: float) -> float:
    risk_budget = cash * risk_per_trade
    per_unit_risk = abs(entry - stop)
    if per_unit_risk <= 0:
        return 0.0
    return risk_budget / per_unit_risk
```

อย่าคิด quantity จากเงินทั้งหมดเสมอไป ถ้า strategy อิง stop distance การ sizing ตามความเสี่ยงต่อดีลจะให้ภาพ drawdown ที่ตรงกว่า

คำถามที่ต้องตอบก่อนเชื่อผลลัพธ์

- slippage คงที่พอหรือควรผูกกับ volume/ATR/spread
- ถ้าส่งคำสั่งขนาดใหญ่ market impact จะเริ่มเมื่อไร
- position sizing ใช้ equity ปัจจุบันหรือ initial capital
- รับ leverage สูงสุดเท่าไรต่อ symbol และรวมทั้งพอร์ต
- การคำนวณต้นทุนใช้ per-trade, per-share, per-contract หรือ notional bps

สัญญาณเตือนว่าต้นทุนยังไม่ realistic

เมื่อ gross PnL สูง แต่ net PnL ยุบเหลือใกล้ศูนย์, average trade เล็กมาก, turnover สูงเกินไป หรือ sensitivity ต่อ slippage เพิ่มขึ้นแค่นิดเดียวแล้ว edge หาย แปลว่า strategy นี้ยังไม่พร้อมสำหรับการคุยเรื่อง deploy

Key takeaway

ไม่มี backtest ไหน deployable ถ้ายังไม่ผ่าน cost model และ position sizing model ที่อิงความจริงพอสมควร

Metrics | Chapter 7

Metrics ที่ดีต้องทำให้คุณกล้าปฏิเสธ strategy ได้เร็ว ไม่ใช่แค่หาข้ออ้างให้มันอยู่ต่อ

รายงานที่มีแค่ total return หรือ win rate ไม่พอ เพราะ strategy ชนะ 80% แต่กำไรเฉลี่ยเล็กและแพ้หนักครั้งเดียวก็อาจแย่กว่ากลยุทธ์ที่ win rate ต่ำกว่าอย่างมาก

Figure: Scorecard for fast decision making

scorecard ที่ดีต้องช่วยคัด strategy ออกให้เร็ว ไม่ใช่ช่วยหาเหตุผลให้มันอยู่ต่อ ภาพนี้สรุป metric หลักที่ควรถูกอ่านพร้อมกันเพื่อเห็นทั้งผลตอบแทน ความเสี่ยง และคุณภาพของ sample

- ดู avg trade after costs คู่กับ trade count เสมอ
- drawdown และ exposure บอกขนาดความเจ็บจริง
- distribution view ช่วยแยก outlier ออกจาก behavior ปกติ

scorecard ขั้นต่ำที่ควรมีทุก run

Metric	ใช้ดูอะไร	ข้อควรระวัง
Net PnL / Return	ภาพรวมกำไรสุทธิ	อย่าดูโดยไม่หักต้นทุน

Metric	ใช้ดูอะไร	ข้อควรระวัง
Max drawdown	ความลึกของการถอยทุน	ควรดูทั้ง absolute และ relative
Profit factor	สัดส่วน gross win ต่อ gross loss	ใช้เดี่ยว ๆ ไม่พอ
Expectancy per trade	ค่าเฉลี่ยความคุ้มค่าต่อดีล	สำคัญมากกับระบบเทรดถี่
Trade count	ขนาด sample	metric สวยแต่มีเทรดน้อยเชื่อยาก
Time in market	ประสิทธิภาพการใช้ทุน	ช่วยเทียบระบบที่ถือสถานะต่างกันมาก
Turnover	การหมุนพอร์ต	สูงเกินไปมักแพ้ต้นทุน

ตัวอย่างสรุปผลเป็น scorecard src/metrics.py

```
import pandas as pd


def build_performance_report(trades: pd.DataFrame, equity_curve: pd.Series) -> dict:
    gross_profit = trades.loc[trades["pnl"] > 0, "pnl"].sum()
    gross_loss = trades.loc[trades["pnl"] < 0, "pnl"].sum()
    max_drawdown = (equity_curve / equity_curve.cummax() - 1.0).min()

    return {
        "trade_count": int(len(trades)),
        "net_pnl": float(trades["pnl"].sum()),
        "win_rate": float((trades["pnl"] > 0).mean()) if len(trades) else 0.0,
        "avg_trade": float(trades["pnl"].mean()) if len(trades) else 0.0,
        "profit_factor": float(gross_profit / abs(gross_loss)) if gross_loss < 0 else None,
        "max_drawdown": float(max_drawdown),
        "best_trade": float(trades["pnl"].max()) if len(trades) else 0.0,
        "worst_trade": float(trades["pnl"].min()) if len(trades) else 0.0,
    }
```

ผลลัพธ์ควร serialize ออกมาได้เป็น dict หรือ JSON เสมอ เพื่อให้ ranking, dashboard และ report generator ใช้ต่อได้

ถ้าแปลเป็นภาษาของ product scorecard คือภาษากลางของ ranking, compare view, shared pages และ operator review ยิ่งทุก run ส่งออก schema เดียวกันได้ คุณก็ยิ่งรู้เร็วขึ้นว่ากลยุทธ์ไหนควรได้เวลารอบถัดไป

นอกจาก scorecard หลัก ควรมี distribution view ด้วย เช่น histogram ของ trade PnL, monthly return heatmap, underwater curve, hold time distribution และ breakdown ตาม market regime เพราะมันช่วยให้เห็นว่ากำไรเกิดจาก pattern เดิมซ้ำ ๆ หรือเกิดจาก outlier ไม่กี่ดีล

metric ที่ช่วยประหยัดเวลามากที่สุด

average trade after costs, max drawdown, trade count, exposure และ profit factor เป็นห้าค่าที่คัด strategy ทิ้งได้เร็วมาก ถ้าค่าชุดนี้ไม่ผ่าน อย่าเพิ่งไปสน Sharpe ที่ละเอียดกว่า

Key takeaway

สร้าง scorecard มาตรฐานชุดเดียวแล้วใช้มันกับทุก run เพื่อให้เทียบ strategy ข้ามเวลาและข้ามทีมได้

Optimization

Chapter 8

Parameter search และ walk-forward ต้องออกแบบเพื่อหาความเสถียร ไม่ใช่ออกแบบเพื่อหาค่าที่สวยที่สุดในอดีต

การ grid search ที่ดีไม่ได้มีเป้าหมายเพื่อหาตัวเลขชุดเดียวที่ชนะทุกคน แต่เพื่อดูว่าบริเวณ parameter ไหนให้ผลคงที่พอ และเมื่อเวลาเดินต่อไปยังรักษาพฤติกรรมเดิมได้หรือไม่

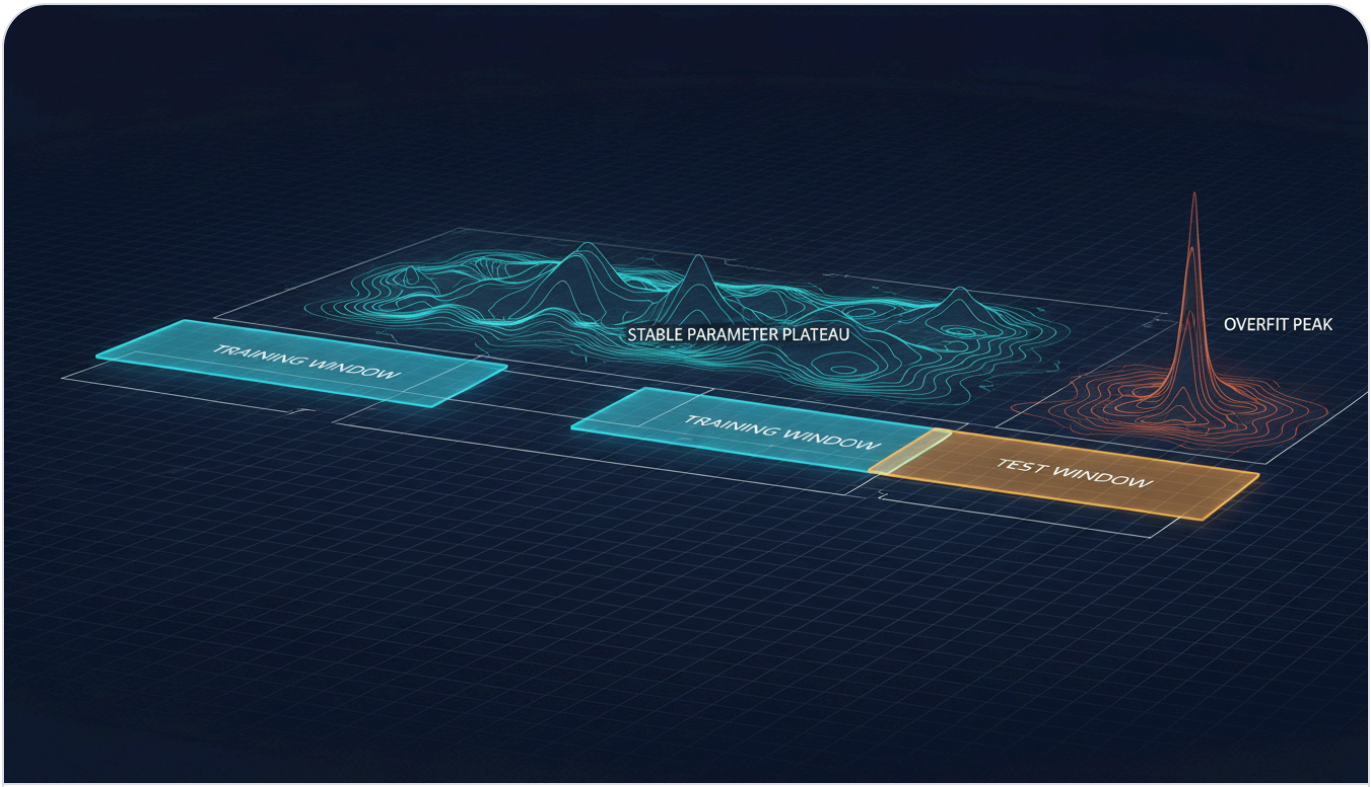

Figure: Walk-forward and parameter plateau

แทนที่จะหาค่า parameter ที่สวยที่สุดในอดีต ให้มองหาบริเวณที่ผลยังพอใช้ได้แม้เลื่อนค่าออกจากจุดดีที่สุด และดูว่าความสม่ำเสมอใน walk-forward ยังพอรับได้หรือไม่เมื่อเวลาเดินต่อไป

- ยอดแหลมมัก overfit มากกว่าพื้นที่ราบ
- บันทึก best config ของทุก window ไว้เสมอ
- สนใจ degradation profile มากกว่าคะแนนสูงสุดครั้งเดียว

หลักการ optimize ที่ควรยึด

- เริ่มจากช่วง parameter ที่มีเหตุผลเชิงโครงสร้าง ไม่ใช่ช่วงกว้างจนสุ่มหาโชค
- เก็บทุก run เป็นตาราง ranking แทนการดูเฉพาะ best row
- ดูความชันของผลลัพธ์รอบ ๆ parameter ที่ดีที่สุด ถ้าแหลมเกินไปมัก overfit

- แยก in-sample และ out-of-sample อย่างชัดเจน
- ใช้ walk-forward เพื่อดูว่าต้อง re-fit บ่อยแค่ไหนและ performance decay เท่าไร

Walk-forward คืออะไรแบบง่าย ๆ

มันคือการจำลองสถานการณ์ว่าในแต่ละช่วงเวลาเรามีข้อมูลอดีตเท่าที่เห็นจริง แล้วเลือก parameter จากอดีตนั้นก่อน เดินไปทดสอบกับอนาคตช่วงถัดไป ถ้าผลพอใช้ได้หลายหน้าต่าง แปลว่าระบบมีวินัยมากกว่าการบังเอิญเจอเลขสวยครั้งเดียว

ตัวอย่าง walk-forward loop — src/walk_forward.py

```
def walk_forward_validate(frame, train_bars, test_bars, candidate_configs, run_once):
    windows = []
    start = 0

    while start + train_bars + test_bars <= len(frame):
        train = frame.iloc[start : start + train_bars]
        test = frame.iloc[start + train_bars : start + train_bars + test_bars]

        ranked = []
        for config in candidate_configs:
            train_report = run_once(train, config)
            ranked.append((config, train_report["net_pnl"], train_report["max_drawdown"]))

        ranked.sort(key=lambda item: (item[1], item[2]), reverse=True)
        best_config = ranked[0][0]
        test_report = run_once(test, best_config)

        windows.append(
            {
                "train_start": train.index[0],
                "train_end": train.index[-1],
                "test_start": test.index[0],
                "test_end": test.index[-1],
                "best_config": best_config,
                "test_report": test_report,
            }
        )
        start += test_bars

    return windows
```

walk-forward ที่ดีต้องบันทึก best config ของแต่ละช่วงไว้ด้วย เพื่อให้วิเคราะห์ได้ว่า parameter drift มากผิดปกติหรือไม่

หลุมพรางที่พบบ่อย

ถ้า strategy ดูดีเฉพาะจุดเมื่อ fast=17, slow=143, stop=2.7 และพอเลื่อนเลขนิดเดียวแล้วพังหมด แปลว่าคุณกำลังถือ curve-fit มากกว่ากำลังถือ system design ที่เสถียร

Key takeaway

มองหา parameter plateau, out-of-sample consistency และ degradation profile มากกว่าค่า best-in-sample ตัวเดียว

Robustness

Chapter 9

ผลลัพธ์ที่ดีต้องถูกทรมานก่อนถึงจะคุยเรื่องความเชื่อมั่นได้

หลังจากได้ strategy ที่ดูมีทรง ขั้นตอนต่อไปไม่ใช่รีบ deploy แต่คือทำให้มันเจอกับสถานการณ์เลวร้ายพอ เพื่อวัดว่าพฤติกรรมมันแตกหักง่ายแค่ไหน

Figure: Robustness torture suite

robustness test ไม่ได้มีไว้ทำให้ report ดูหนา แต่มีไว้ตอบว่า edge เปราะบางแค่ไหนเมื่อโดน cost stress, regime split, timing delay และ data perturbation ในระดับที่โลกจริงอาจเจอได้

- ดูว่าพังเพราะอะไร ไม่ใช่แค่พังหรือไม่พัง
- strategy ที่ดีควรเสื่อมแบบค่อยเป็นค่อยไป
- ผลที่อยู่รอดได้ภายใต้ความแย่เล็กน้อยจึงคุยเรื่อง deploy ต่อได้

ชุดทดสอบ robustness ที่ควรมี

- cost stress: เพิ่ม commission/slippage ทีละขั้นแล้วดูว่า edge อยู่หรือไม่
- regime split: แยกตลาด trend, range, high-vol, low-vol
- trade shuffle / bootstrap: ดู distribution ของ path outcome
- parameter perturbation: ขยับค่ารอบจุดที่ดีที่สุด

- delay entry/exit 1 bar: ดูว่า edge ไวต่อ timing เกินไปหรือไม่
- data perturbation: bar shift เล็กน้อยหรือ random drop เพื่อวัดความไวต่อข้อมูล

ตัวอย่างสัญญาณ pass / fail

การทดสอบ	สัญญาณว่ายังพอใช้	สัญญาณว่าอันตราย
Cost stress	net PnL ลดลงแต่ยังบวก	เพิ่มต้นทุนเล็กน้อยแล้ว strategy ติดลบ
Regime split	รู้ชัดว่าชนะใน regime ใด	กำไรทั้งหมดมาจากช่วงสั้นมากช่วงเดียว
Parameter perturbation	ผลลัพธ์ยังใกล้เคียงกัน	เลื่อนนิดเดียวแล้วพังหมด
1-bar delay	edge ลดลงแบบสมเหตุสมผล	entry ช้าหนึ่งแท่งแล้วกลายเป็นคนละระบบ

ถ้าอธิบายแบบง่าย Monte Carlo คือการสุ่มเรียงลำดับผลลัพธ์เดิม หรือสุ่มตัวอย่างชุดดีลซ้ำหลายครั้ง เพื่อดูว่า equity curve และ drawdown จะเหวี่ยงได้แค่ไหนถ้า luck ไม่เรียงแบบเดิม มันช่วยให้คุณเห็น path risk ก่อนตัดสินใจเอา run ไป share หรือ deploy

ในเชิงปฏิบัติ ผมแนะนำให้ robustness report เป็นส่วนบังคับของทุก strategy ที่คิดจะ deploy เพราะมันช่วยเปลี่ยนการตัดสินใจจากความรู้สึกเป็น evidence เช่น “strategy นี้ยังรอดที่ slippage 4 bps แต่ไม่รอดที่ 8 bps” ซึ่งใช้คุยกับ execution team ได้ทันที

Monte Carlo บอกอะไร และไม่บอกอะไร

มันช่วยตอบว่า “ถ้าลำดับชนะ/แพ้ไม่เหมือนเดิม ระบบยังรับ drawdown ไหวไหม” แต่ไม่ได้แก้ปัญหา data, look-ahead bias หรือ execution model ที่ผิด เพราะฉะนั้น Monte Carlo เป็นชั้น stress test เพิ่ม ไม่ใช่ตราประทับว่ากลยุทธ์ดีแล้ว

Key takeaway

robustness test คือเครื่องมือวัดความเปราะบางของ edge ไม่ใช่ขั้นตอนเสริมที่ทำเมื่อมีเวลา

Reporting

Chapter 10

ผล backtest ที่ดีต้องอ่านจบภายในไม่กี่นาทีแล้วตัดสินใจต่อได้

report ที่ดีไม่ควรบังคับให้คนอ่านกลับไปเปิด notebook เพื่อหาว่า strategy นี้เข้าออกอย่างไร เทรดทั้งหมดกี่ครั้ง หรือแพ้หนักเพราะอะไร รายงานควรบรรจุบริบท, scorecard, chart, trade samples และ next action ในที่เดียว

Figure: Readable research artifact

งาน backtest จะเริ่ม scale ได้เมื่อทุก run ออกมาเป็น artifact ที่อ่านและส่งต่อได้ ภาพนี้สรุปชิ้นส่วนหลักของหนึ่ง research package ตั้งแต่ summary, charts, trade ledger, metadata ไปจนถึง next action

- คนที่ไม่ได้เขียน strategy ควรอ่านแล้วเข้าใจได้
- metadata คือสิ่งที่ทำให้ผลย้อนกลับและ audit ได้
- artifact ที่ดีเปลี่ยนงานวิจัยจาก activity เป็น inventory

artifact ที่ควรออกจากทุก run

- summary.json สำหรับ dashboard และ ranking
- trades.csv สำหรับ audit รายดีล
- equity_curve.csv หรือ parquet สำหรับวิเคราะห์ต่อ
- report.html ที่มี scorecard + charts + commentary

- config snapshot เพื่อให้ reproducible
- run metadata เช่น git SHA, data range, source, timestamp

ตัวอย่าง metadata ที่ควรบันทึก reports/runs/run_metadata.json

```
{
  "run_id": "btc_1h_mr_20260330_014500",
  "symbol": "BTCUSDT",
  "timeframe": "1h",
  "data_start": "2022-01-01T00:00:00Z",
  "data_end": "2026-03-01T00:00:00Z",
  "config_path": "configs/btc_1h_mean_reversion.yaml",
  "git_sha": "abc1234",
  "fee_bps": 4.0,
  "slippage_bps": 3.0,
  "engine_version": "1.2.0"
}
```

metadata ทำให้ run ย้อนกลับได้ว่าผลนี้มาจากโค้ดและข้อมูลชุดไหน ไม่ต้องอาศัยความจำ

เมื่อมี artifact ครบ คุณจะเปลี่ยนงานวิจัยจากกิจกรรมชั่วคราวเป็น inventory ที่ค้นหาและนำกลับมาใช้ใหม่ได้ และตรงนี้เองที่ทีมเริ่ม scale ได้ เพราะไม่ต้องถามคนเดิมทุกครั้งว่า run นี้เคยทำอะไรไว้

artifact ในภาษาของ Balance Labs คืออะไร

มันคือสิ่งที่ระบบเอาไปแสดงใน ranking, shared strategy page, my strategies และ audit history ถ้า run หนึ่งไม่มี artifact ที่ย้อนกลับได้ มันก็ยากจะกลายเป็นสินทรัพย์ของ product

rule ง่าย ๆ สำหรับ report

ถ้าคนที่ไม่ได้เขียน strategy อ่าน report แล้วไม่สามารถตอบได้ว่า edge คืออะไร, แพ้เมื่อไร, และควร iterate ตรงไหน แปลว่า report ยังไม่ดีพอ แม้ตัว engine จะถูกต้องก็ตาม

Key takeaway

ทุก run ควรได้ artifact ที่แชร์ต่อ, ขึ้น ranking หรือ audit ได้ เช่น HTML report, JSON summary, CSV ledger และ chart image

AI Chapter 11

ใช้ AI เป็นตัวเร่งงาน backtest ได้ แต่ต้องมี contract, guardrails และ verification loop

ใน Balance Labs AI อยู่ที่ชั้น intake และ acceleration มันช่วยย่นเวลาทำ boilerplate, สร้าง test case, refactor report text และแตก strategy idea เป็น config ได้ดี แต่ถ้าปล่อยให้ AI ตัดสิน assumptions แทนคุณ ระบบจะได้โค้ดสวยแต่ logic มั่ว

Figure: AI guardrails and verification loop

ให้ AI เร่งงานเฉพาะส่วนที่ตรวจซ้ำได้ แล้วบังคับทุก output ผ่าน contract, deterministic checks, tests และ evidence review วงจรนี้คือสิ่งที่ทำให้ AI เป็น accelerator แทนที่จะเป็นแหล่ง bias เพิ่ม

- AI ควรเร่ง boilerplate ไม่ควรเดา assumptions แทนเรา
- prompt ที่ดีต้องระบุ non-negotiable constraints
- output ทุกชิ้นควรถูกตรวจด้วยกลไกเดิมกับโค้ดคนเขียน

งานที่ AI ควรทำ

- scaffold โครงสร้างไฟล์และ dataclass
- แปลง strategy spec เป็น skeleton code
- เขียน unit tests จาก edge cases ที่เรานิยามไว้แล้ว

- สรุปผลลัพธ์จาก summary.json ให้เป็นภาษาที่ทีมธุรกิจอ่านเข้าใจ
- ช่วยสร้าง comparison matrix ระหว่างหลาย run

งานที่ AI ไม่ควรทำแทนคุณโดยไม่มีการตรวจ

- เดา execution assumptions
- เลือก cost model เอง
- สรุปว่ากลยุทธ์ deploy ได้โดยไม่มี robustness report
- สร้าง indicator logic ที่ใช้ข้อมูลอนาคตโดยไม่มี test จับ

ถ้าใช้ Labs Chat ให้คิดว่า AI ช่วย draft spec, generate code, refactor และสรุปผลเบื้องต้น ส่วนคำตัดสินยังต้องมาจาก Execute report, trade log, robustness และ deployment gates ไม่ใช่มาจากน้ำเสียงที่มั่นใจของโมเดล

ตัวอย่าง prompt contract ที่ดีขึ้น — prompt-contract.txt

```
Task:
Implement a long-only Python backtest strategy.

Non-negotiable assumptions:
- Use next-bar open for entry and exit.
- Charge 4 bps commission and 3 bps slippage per side.
- No look-ahead bias.
- Use ATR-based stop loss.
- Return trades ledger, summary dict, and equity curve.

Required tests:
- No duplicate entries while a position is open.
- Exit is triggered when stop is breached.
- Strategy does not read future bars.
- Summary keys follow the agreed schema.
```

ยิ่ง contract ชัด AI ยิ่งเป็นเครื่องทุ่นแรงที่มีประโยชน์ และยิ่งลดรอบการแก้โค้ดไร้สาระ

อย่าใช้ AI เป็นแหล่งความมั่นใจปลอม

โค้ดที่ดูสะอาดและคำอธิบายที่มั่นใจไม่ได้แปลว่าถูกต้อง สิ่งที่ทำให้ output ของ AI มีค่าไม่ใช่น้ำเสียง แต่คือการที่มันผ่าน test, data validation และ evidence review ชุดเดียวกับโค้ดที่คนเขียนเอง

Key takeaway

ให้ AI เร่งงานที่ตรวจซ้ำได้ และบังคับทุก output ผ่าน deterministic checks ก่อนเสมอ

Deployment

Chapter 12

ก่อนย้ายจาก research ไป production ต้องมี handoff ที่ชัด ไม่อย่างนั้นความเสี่ยงจะย้ายจาก alpha ไปอยู่ที่ operation

ใน product flow ของ Balance Labs จุดนี้คือช่วงที่ผลจาก Labs Execute ถูกส่งต่อไปยัง ranking, share page หรือ deploy rail อย่าง Botmoon กลยุทธ์ที่ backtest ดีอาจยังไม่พร้อมใช้งานจริง ถ้ายังไม่มีการกำหนด kill switch, monitoring, expected live deviation และการจัดการเหตุการณ์ผิดปกติ เช่น data feed ขาด, latency สูง หรือ broker reject order

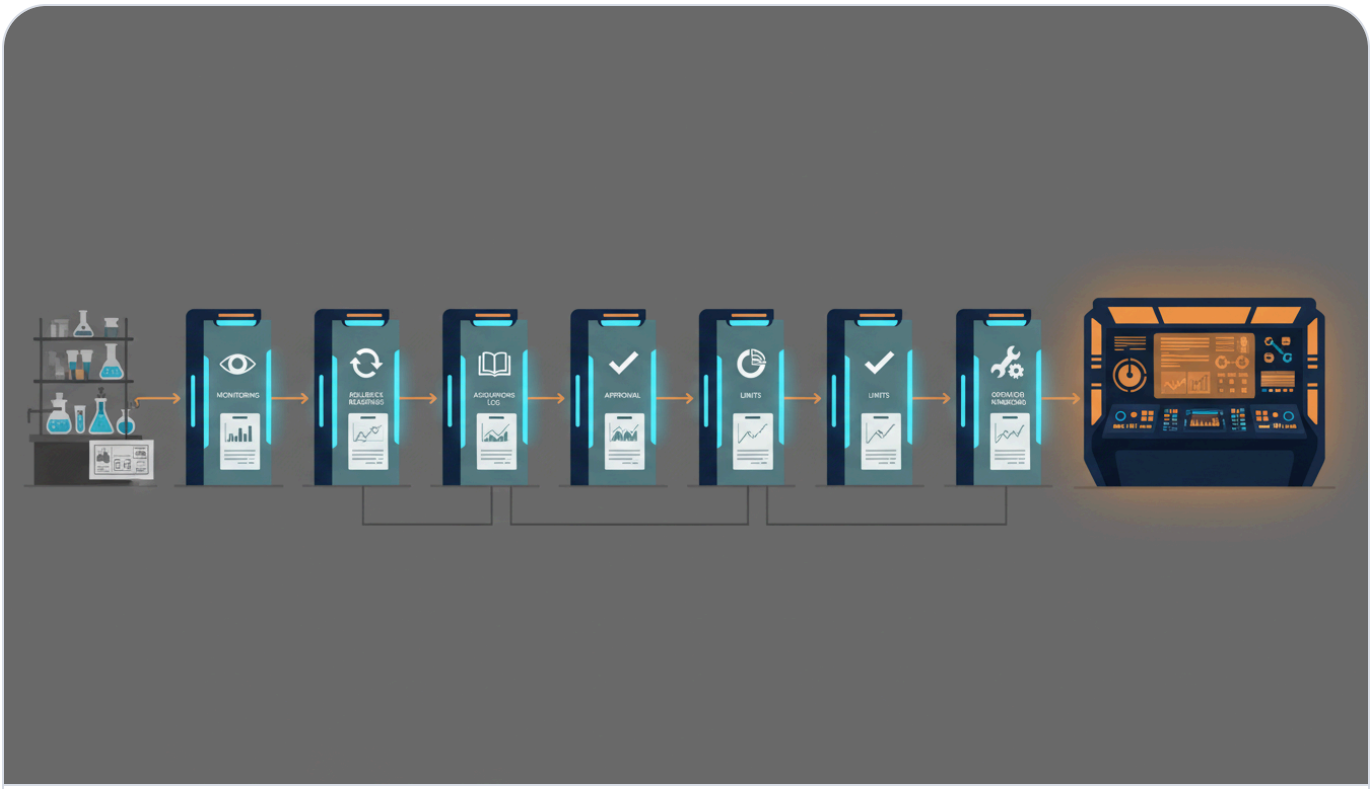

Figure: Production handoff and go-live gates

ช่วงเปลี่ยนผ่านจาก research ไป production คือจุดที่หลายทีมเสียของ ภาพนี้รวบรวม gate สำคัญตั้งแต่ validation, monitoring, alerting, kill switch และ rollback path เพื่อให้ระบบ deploy ได้อย่างรับผิดชอบ

- go-live ต้องมี owner ของ monitoring และ incident path
- expected live deviation ต้องถูกนิยามก่อนยิงจริง
- handoff ที่ดีลด operational risk ได้มากกว่าเพิ่ม alpha

ถ้าต้องส่งระบบจาก Balance Labs ไปให้ operator ใช้งานจริง หนึ่งในรูปแบบที่ practical มากคือ deploy ไปยัง rail อย่าง Botmoon หรือสร้าง Python GUI ที่ user แค่ใส่ Binance API key, secret, symbol และกด Run ได้เลย แต่ชั้นนี้ควรเป็นเพียง shell สำหรับเรียก strategy ที่ผ่านการอนุมัติแล้ว ไม่ใช่ที่รวม logic ทุกอย่างไว้แบบไร้โครงสร้าง

Prompt template: ให้ AI สร้าง Binance trading GUI gui-trader-prompt.txt

```
Task:
Create a Python desktop GUI that runs an approved Binance trading system.

Context:
- Strategy logic already exists as pure Python functions.
- End user should only paste Binance API key and Binance API secret, choose symbol, then press Run.
- The app must start in sandbox / paper mode by default.
- Separate GUI, trading service, exchange client, and risk checks into different files.

Must build:
- A Tkinter GUI with fields for API key, API secret, symbol, timeframe, and risk per trade.
- Buttons for Validate keys, Run, Stop, and Dry run.
- A read-only log panel plus clear status labels.
- A Binance client wrapper with reconnect handling and rate-limit aware calls.
- A service layer that imports signals from the approved strategy module instead of rebuilding the str
- Setup instructions for pip install and python app.py.

Non-negotiable safeguards:
- Do not hardcode credentials.
- Keep API keys in memory only for the current session.
- Default to sandbox / testnet mode.
- Refuse to place orders when config validation fails.
- Validate position size, symbol filters, and max exposure before any order.

Return:
- app.py
- trader_service.py
- exchange_client.py
- risk_checks.py
- README.md
- tests for config validation and order sizing
```

prompt แบบนี้ช่วยให้ AI scaffold ระบบ operator GUI ที่แยกชั้นชัดเจน และยังบังคับ guardrails เรื่อง credential, risk checks และ sandbox mode ตั้งแต่ต้น

องค์ประกอบขั้นต่ำของ GUI สำหรับรันระบบเทรด

- ช่องกรอก Binance API key และ API secret แบบ session only ไม่บันทึกลง source code
- ปุ่ม Validate, Run, Stop และสถานะว่ากำลัง sandbox หรือ live
- ช่องเลือก symbol, timeframe, risk per trade และ max exposure
- log panel แบบ read-only เพื่อให้ operator รู้ว่าระบบทำอะไรอยู่
- risk checks ก่อนยิง order ทุกครั้ง เช่น symbol filters, position size และ exposure limit

ตัวอย่าง Python GUI runner สำหรับ Binance app.py

```
from __future__ import annotations

import threading
import time
import tkinter as tk
from dataclasses import dataclass
from tkinter import ttk

import ccxt


@dataclass
class GuiConfig:
    api_key: str
    api_secret: str
    symbol: str
    sandbox: bool = True


class BinanceRunner:
    def __init__(self, config: GuiConfig, log):
        self.config = config
        self.log = log
        self.running = False
        self.exchange = ccxt.binance(
            {
                "apiKey": config.api_key,
                "secret": config.api_secret,
                "enableRateLimit": True,
                "options": {"defaultType": "spot"},
            }
        )
        if config.sandbox:
            self.exchange.set_sandbox_mode(True)

    def validate(self) -> None:
        if not self.config.api_key or not self.config.api_secret:
            raise ValueError("API key and secret are required")
```

```
        self.exchange.load_markets()
        market = self.exchange.market(self.config.symbol)
        if not market["active"]:
            raise ValueError("Symbol is not active on Binance")

    def start(self) -> None:
        self.running = True
        while self.running:
            ticker = self.exchange.fetch_ticker(self.config.symbol)
            last_price = float(ticker["last"])
            self.log(f"{self.config.symbol} last price: {last_price:.2f}")

            # Replace this stub with the approved strategy + risk checks.
            signal = "hold"
            if signal == "buy":
                self.log("buy signal generated; place order only after risk validation")

            time.sleep(3)

    def stop(self) -> None:
        self.running = False


class TradingGui(tk.Tk):
    def __init__(self) -> None:
        super().__init__()
        self.title("Binance Strategy Runner")
        self.geometry("760x520")

        self.api_key = tk.StringVar()
        self.api_secret = tk.StringVar()
        self.symbol = tk.StringVar(value="BTC/USDT")
        self.sandbox = tk.BooleanVar(value=True)
        self.status = tk.StringVar(value="idle")
        self.runner: BinanceRunner | None = None

        form = ttk.Frame(self, padding=16)
        form.pack(fill="both", expand=True)

        ttk.Label(form, text="Binance API Key").grid(row=0, column=0, sticky="w", pady=6)
        ttk.Entry(form, textvariable=self.api_key, width=56).grid(row=0, column=1, sticky="ew", pady=6

        ttk.Label(form, text="Binance API Secret").grid(row=1, column=0, sticky="w", pady=6)
        ttk.Entry(form, textvariable=self.api_secret, show="*", width=56).grid(row=1, column=1, sticky

        ttk.Label(form, text="Symbol").grid(row=2, column=0, sticky="w", pady=6)
        ttk.Entry(form, textvariable=self.symbol, width=24).grid(row=2, column=1, sticky="w", pady=6)
```

```
        ttk.Checkbutton(form, text="Use sandbox / testnet", variable=self.sandbox).grid(
            row=3, column=1, sticky="w", pady=6
        )

        controls = ttk.Frame(form)
        controls.grid(row=4, column=1, sticky="w", pady=10)
        ttk.Button(controls, text="Run", command=self.handle_run).pack(side="left", padx=(0, 8))
        ttk.Button(controls, text="Stop", command=self.handle_stop).pack(side="left")

        ttk.Label(form, textvariable=self.status).grid(row=5, column=1, sticky="w")

        self.log_box = tk.Text(form, height=14, state="disabled")
        self.log_box.grid(row=6, column=0, columnspan=2, sticky="nsew", pady=(12, 0))

        form.columnconfigure(1, weight=1)
        form.rowconfigure(6, weight=1)

    def write_log(self, message: str) -> None:
        self.log_box.configure(state="normal")
        self.log_box.insert("end", message + "\n")
        self.log_box.see("end")
        self.log_box.configure(state="disabled")

    def handle_run(self) -> None:
        config = GuiConfig(
            api_key=self.api_key.get().strip(),
            api_secret=self.api_secret.get().strip(),
            symbol=self.symbol.get().strip().upper(),
            sandbox=self.sandbox.get(),
        )
        self.runner = BinanceRunner(config, self.write_log)
        thread = threading.Thread(target=self._run_worker, daemon=True)
        thread.start()

    def _run_worker(self) -> None:
        try:
            self.status.set("validating")
            assert self.runner is not None
            self.runner.validate()
            self.status.set("running")
            self.runner.start()
        except Exception as exc:
            self.status.set("error")
            self.write_log(str(exc))

    def handle_stop(self) -> None:
        if self.runner:
            self.runner.stop()
```

```
            self.status.set("stopped")


if __name__ == "__main__":
    TradingGui().mainloop()
```

โค้ดนี้เป็น operator shell ขั้นต้น: user เอา key/secret ไปวางแล้วรันได้ทันที แต่ส่วน signal และ order placement ควรเชื่อมกับ strategy module ที่ผ่านการตรวจแล้ว และควรเริ่มจาก sandbox / paper mode ก่อนเสมอ

เริ่มจาก sandbox ก่อน live เสมอ

การมี GUI ที่กด Run ได้ง่าย ไม่ได้แปลว่าควรปล่อยให้ยิงเงินจริงทันที ค่า default ควรเป็น sandbox หรือ dry-run, key ไม่ควรถูกเก็บถาวรในตัวแอป, และทุก order ต้องผ่าน position sizing กับ exposure checks ก่อนเสมอ

go-live gates ที่ควรผ่าน

Gate	คำถามที่ต้องตอบ	ตัวอย่างเกณฑ์
Research	ผ่าน out-of-sample หรือยัง	walk-forward net positive และ drawdown ยังรับได้
Execution	ต้นทุนจริงใกล้กับที่จำลองหรือไม่	live slippage ไม่เกิน tolerance
Risk	มี hard stop ระดับระบบหรือไม่	max daily loss, max position, disconnect behavior
Monitoring	รู้ไหมว่าต้อง alert เมื่อไร	order reject, feed lag, pnl drift, exposure breach
Rollback	ถ้าพังจะหยุดอย่างไร	manual kill switch + automated disable flag

สิ่งที่ต้องส่งต่อให้ทีม execution หรือ future operator

- strategy thesis แบบหนึ่งหน้า
- config production และค่าที่ไม่ควรแก้เอง
- expected trade frequency และ expected holding time
- live-vs-backtest drift thresholds
- failure playbook เมื่อ feed, broker หรือ signal service ผิดปกติ
- link ไปยัง report ล่าสุดและ historical validation artifact

เมื่อคุณทำ handoff ดี คุณจะลดความเสี่ยงที่ระบบล้มเพราะ operation มากกว่าล้มเพราะ alpha เสีย ซึ่งเป็นสิ่งที่เกิดขึ้นบ่อยอย่างน่าประหลาด โดยเฉพาะในทีมที่ research กับ execution แยกกันคนละชุด

ปุ่ม Deploy ไม่ได้แปลว่าข้าม validation ได้

ยิ่ง workflow ยิ่งได้ง่ายเท่าไร guardrails ต้องยิ่งชัด ปุ่ม Deploy ควรถูกมองว่าเป็นการเปลี่ยนโหมดความรับผิดชอบจาก evidence review ไปสู่ operational control ไม่ใช่ปุ่มลัดข้ามงานตรวจ

นิยาม deployable system แบบเรียบง่าย

ระบบ deployable คือระบบที่มี edge พอประมาณ, assumptions ชัด, ต้นทุน realistic, evidence อ่านง่าย, และมีวิธีปิดความเสี่ยงเมื่อโลกจริงไม่เดินตาม backtest มันไม่จำเป็นต้องดูสมบูรณ์แบบ แต่มันต้องรับผิดชอบได้

Key takeaway

go-live ที่ดีไม่ใช่แค่กด Deploy ได้ แต่คือรู้ล่วงหน้าว่าอะไรถือว่าผิดปกติ ใครต้องรับผิดชอบ และต้องหยุดเมื่อไร

FAQ

ควรเริ่มจาก library สำเร็จรูปหรือเขียน engine เอง

ถ้ายังนิยามกติกา execution ไม่ซับซ้อน pandas-based engine ที่เขียนเองจะทำให้เข้าใจระบบลึกกว่า แต่ถ้าทีมมี requirement ซับซ้อนและต้องการ ecosystem สำเร็จรูป อาจใช้ framework ที่มีอยู่แล้วโดยยังคงหลักแยก signal, execution และ report ให้ชัด

ถ้ามีแค่ notebook อยู่แล้ว ต้องรื้อใหม่ทั้งหมดไหม

ไม่จำเป็น ให้เริ่มจากย้าย logic ที่สำคัญที่สุดออกจาก notebook ก่อน เช่น data normalization, signal generation และ metrics จากนั้นค่อยทำ report และ config ให้ reproducible การ refactor ทีละชั้นปลอดภัยกว่าการรื้อทีเดียว

backtest ต้องใช้ tick data เสมอหรือไม่

ไม่เสมอ ถ้า strategy ของคุณตัดสินใจบน daily หรือ hourly bars ข้อมูลแท่งอาจเพียงพอ แต่ต้องยอมรับข้อจำกัดเรื่อง intrabar execution และ stop behavior ถ้า edge อาศัย microstructure จริง tick data จะสำคัญมากขึ้น

Sharpe ratio ยังจำเป็นไหม

ยังมีประโยชน์ แต่ไม่ควรเป็น metric ตัวแรกที่คุณใช้คัดกลยุทธ์ สำหรับงานคัดกรองเบื้องต้น avg trade after costs, trade count, drawdown และ profit factor มักให้สัญญาณที่ใช้ได้เร็วกว่า

Monte Carlo คืออะไรแบบง่าย ๆ

ให้นึกว่ามันคือการเอาผลลัพธ์เดิมมาสลับลำดับหรือสุ่มตัวอย่างใหม่หลายรอบ เพื่อดูว่าเส้นทุนและ drawdown จะเหวี่ยงได้แค่ไหนถ้าโชคไม่เรียงแบบเดิม มันใช้ดู path risk ไม่ใช่ใช้ยืนยันว่า edge มีจริง

AI ช่วยเขียน strategy ทั้งหมดให้จบได้ไหม

ช่วย scaffold และเร่งงานได้มาก แต่ไม่ควรปล่อยให้ AI เลือก assumptions, data path หรือ validation criteria เอง คุณยังต้องเป็นคนกำกับ spec และเป็นเจ้าของ evidence อยู่ดี

ถ้าใช้ Balance Labs อยู่แล้ว ยังต้องเข้าใจ Python ลึกไหม

ไม่จำเป็นต้องเขียน engine เองเพื่อเริ่มต้น แต่ควรเข้าใจพออ่าน assumptions, trade log, cost model และรู้ว่า strategy logic อยู่ตรงไหน เพราะ product ช่วยย่นงานให้ แต่ไม่ได้ลบความรับผิดชอบเรื่องการตีความผลลัพธ์

อะไรคือสัญญาณว่า strategy นี้ควรหยุดพัฒนา

เมื่อ edge หายทันทีหลังใส่ต้นทุน realistic, ผลลัพธ์ดีจาก outlier ไม่กี่ดีล, parameter แหลมเกินไป, หรือ robustness tests ส่วนใหญ่ไม่ผ่าน การหยุดเร็วช่วยประหยัดเวลากว่าการแก้ต่อแบบไร้กรอบ

Final Checklist

- นิยาม market universe, timeframe, execution assumptions และ acceptance criteria แล้ว

- normalize data ผ่านขั้นตอนเดียวกันทุก run และมี validation ชัดเจน
- strategy spec แยกเป็น regime, setup, trigger, risk, exit และ invalidation แล้ว
- engine แยก signal, execution model และ trade ledger ออกจากกันแล้ว
- ใส่ commission, spread, slippage และ position sizing แบบ realistic แล้ว
- มี scorecard มาตรฐานและ distribution view สำหรับทุก run
- run parameter search แบบเก็บทุกผลลัพธ์ ไม่ดูแค่ best row
- ผ่าน walk-forward และ cost stress อย่างน้อยหนึ่งรอบ
- เข้าใจว่า Monte Carlo ใช้ดู path risk ไม่ใช่ใช้กลบข้อผิดพลาดของ data/execution
- มี HTML/JSON/CSV artifact ที่แชร์ต่อ, ขึ้น ranking หรือ audit ได้
- AI output ทุกชิ้นผ่าน tests และ deterministic checks ก่อนใช้งาน
- มี go-live gates, monitoring, kill switch และ rollback path ชัดเจน
- มี prompt สำหรับ scaffold operator GUI ที่กำหนด sandbox, risk checks และ session-only credential handling ชัดเจน
- GUI runner รับ Binance API key / secret แบบ session only และเริ่มใน sandbox หรือ paper mode โดย default

Web edition + PDF edition

หน้า route หลักถูกทำไว้สำหรับการอ่านบนเว็บ ส่วนไฟล์ PDF ใช้สำหรับดาวน์โหลด, พิมพ์, หรือเก็บใน research library ภายในทีม ทั้งสองเวอร์ชันใช้ source เดียวกันเพื่อไม่ให้เนื้อหาเพี้ยนกันในภายหลัง