CASE STUDY 03

Automated Cryptocurrency Trading Bot

Python development | API integration | Backtesting | Signal automation | Cloud concepts

Context

A personal learning project that automates a repeated, rules-based decision process.

My role

Directed requirements and strategy logic, built modules with AI-assisted programming, tested and debugged.

Important

Not financial advice. No claim of guaranteed or optimised trading performance.

Turning a manual routine into a controlled workflow

Business problem

Manual trading needs constant chart monitoring, repeated indicator review, rapid order entry and emotional discipline. Decisions become inconsistent and hard to scale.

Project objective

Build a rules-based system that retrieves market data, calculates indicators, generates buy and sell signals, backtests and executes approved spot trades.

Solution approach

A modular Python platform with separate components for market data, indicators, signal logic, execution, logging and backtesting.

Outcome

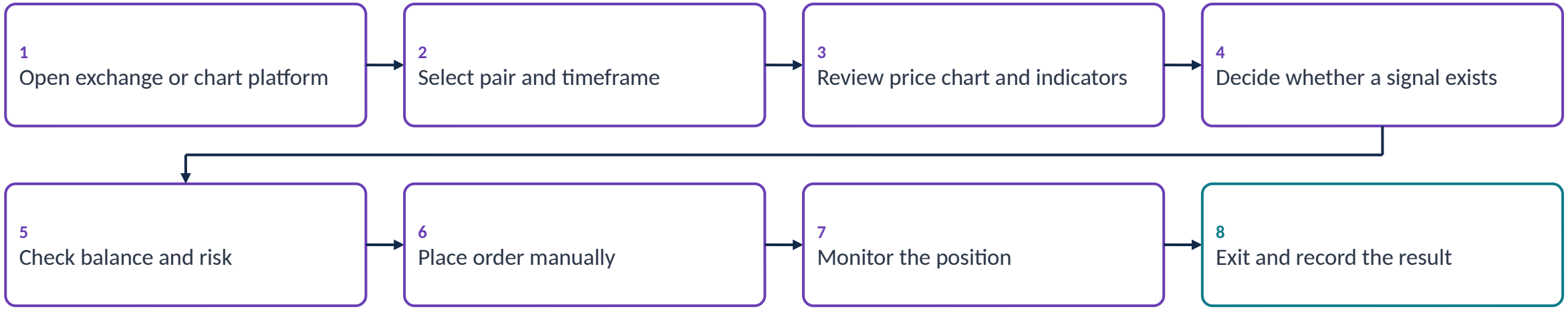
A working personal project showing automation, API integration, troubleshooting, an exchange migration and cloud-deployment concepts.

The pattern: one controlled workflow



Transferable: the same break-down mindset applies to business systems, operations and process-improvement work.

As-is: the manual trading process



Repeat continuously as markets change: difficult to scale, test or monitor consistently

Key limitations

- Time intensive
- Constant monitoring
- Slow reaction to signals
- Manual order-entry errors

Decision-quality risks

- Emotional bias
- Inconsistent criteria
- Limited repeatability
- Hard to compare scenarios

Process opportunity

- Convert chart review into rules
- Automate calculations
- Backtest before use
- Log actions and exceptions

Requirements and design objectives

Trading scope

- Spot trading only
- Long-only strategy
- No leverage
- Buy and sell by rule

Data and indicators

- Retrieve candles
- Support 5m, 1h, 2h
- Calculate indicators
- Handle exchange formats

Strategy engine

- Deterministic signals
- Fee allowance
- Balance validation
- Duplicate-signal protection

Operations

- Logging
- Error handling
- API keys in environment variables
- Cloud-ready runtime

Design principle: separate exchange-specific code, indicator calculations, strategy decisions and execution so each can be tested and changed independently when APIs or assumptions change.

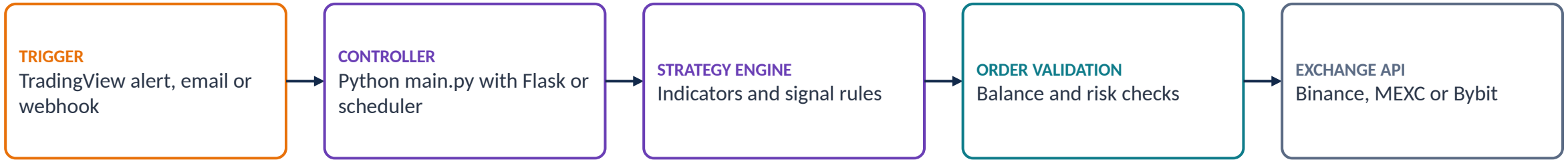
Modular
separated responsibilities

Spot only
no leverage

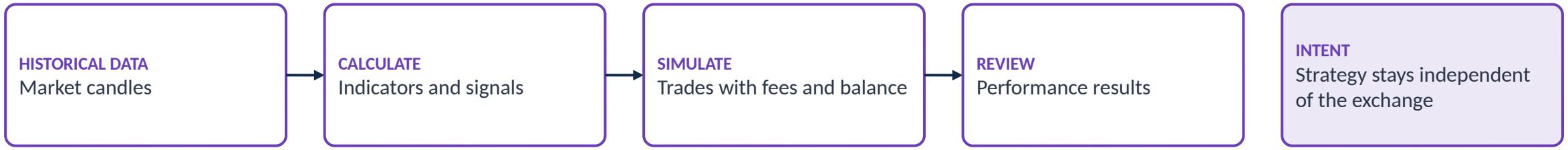
Backtest
before live use

Env vars
credential control

Solution architecture



Backtesting workflow



Important note: this was built as a personal project and learning system. Any real-money use would need stronger controls, monitoring, credential protection, paper-trading validation and risk limits.

Modular Python design

Module	Purpose	Capability demonstrated
binance_data_fetcher.py	Retrieve historical Binance candlestick data	API use and data extraction
MEXC data module	Retrieve exchange data after migration	Exchange adaptation
heikin_ashi.py	Convert standard candles into Heikin Ashi values	Data transformation
chandelier_exit.py	Calculate Chandelier Exit logic	Indicator implementation
zlsma.py	Calculate Zero-Lag LSMA	Indicator implementation
check_buy_sell_signals.py	Evaluate combined strategy rules	Decision logic
execute_trade.py	Submit buy and sell orders through the API	Execution automation
backtest.py	Simulate historical trades and fees	Testing and validation
main.py / main_rsi.py	Coordinate the workflow and RSI variation	System orchestration

Design value: market data, indicators, strategy, execution and testing can change independently rather than living in one large script.

Trading strategy logic

Heikin Ashi

Reduces short-term noise and smooths candle behaviour

Chandelier Exit

ATR-based exit logic. Period tested around 22, multiplier around 3

ZLSMA

Trend filter, commonly tested around a 50-period setting

RSI

Momentum confirmation. 14-period window, thresholds tested around 50–55

Rules-based decision flow



Tested assumptions

- Spot, long-only. 100% simulated capital per trade
- Full position sold on exit signal. Fee allowance ~0.1%

Positioning: these are tested settings and design choices, not guaranteed or optimised trading parameters.

Backtesting process



8. Refine strategy, then repeat

Backtest assumptions

- Starting balance 1,000 USDT
- Fee rate 0.001
- Full allocation per simulated trade
- Full sell on exit signal

What it measures

- Trade history
- Balance change
- Behaviour by timeframe
- Impact of fees and signals

What it does not prove

- Future profitability
- Market stability
- Perfect execution
- Exchange availability

Business-analysis lesson: backtesting turns assumptions into measurable test scenarios, which supports better decisions than intuition alone.

Development challenges and resolutions

Challenge	Investigation and response	Capability demonstrated
Exchange data formats differed	Revised dataframe mapping and parsing rules	Problem diagnosis
MEXC returned 8 columns, not 12	Adjusted column expectations and validation	Data handling
APIs needed epoch timestamps	Converted dates to exchange-compatible values	Technical troubleshooting
Libraries had different methods	Compared clients and exchange-specific APIs	Research and adaptation
Cloud start-up failures	Checked the Flask health endpoint and deployment logs	Runtime debugging
API keys missing in cloud runtime	Moved credentials to environment variables	Security awareness
Email alert filtering	Processed unread alerts with specific subjects	Automation design
Exchange migration changed rules	Separated exchange logic from strategy logic	Modular design

Why it is included: real projects are rarely perfect. This shows troubleshooting, structured learning and persistence through ambiguity.

Cloud and automation architecture

External trigger
TradingView alert, email alert or webhook

↓

Cloud service
Google Cloud Run with a Python Flask runtime

↓

Endpoints and monitoring
/health, /trigger and logs

↓

Bot workflow
Signal handling, validation, execution decision

↓

Exchange connection
CCXT or exchange client with secure credentials

Deployment objective

Let the bot run without a home PC staying on. The cloud service receives triggers, runs validation and records logs.

Minimum production controls

- Managed environment variables
- Logging, alerts and duplicate-signal protection
- API keys without withdrawal rights

Not production-grade without

- Monitoring and recovery rules
- Paper-trading validation
- Error escalation, position and exposure limits

Risk, security and governance

Risk area	Example risks	Practical controls
Market risk	Losses, volatility, strategy overfitting	Spot only, paper trading, conservative limits
Credential risk	API key exposure or misuse	Environment variables, no withdrawal permission
Execution risk	Wrong quantity, duplicate signal, partial fill	Validation, logs, duplicate checks
Exchange risk	Outage, rate limit, changed API rules	Error handling, retry logic, monitoring
Data risk	Bad candles, missing values, precision errors	Input validation, exchange-specific parsing
Operational risk	Cloud failure, missed alerts, unstable runtime	Health checks, logs and alerting
Decision risk	Backtest not representative of the future	Test across timeframes, avoid profit claims

Governance: automation should reduce repetitive work, but it must include validation, monitoring and human oversight.

What this project shows a hiring manager

What I did in the project	Business analyst skill it demonstrates
Broke a manual routine into explicit rules	Process decomposition and requirements definition
Separated exchange logic from strategy logic	Designing for change and reducing rework
Backtested assumptions before any execution	Evidence-based decisions and test scenarios
Diagnosed data-format, timestamp and deployment faults	Root cause analysis and structured troubleshooting
Kept credentials out of code, no withdrawal rights	Security and risk awareness
Documented architecture, limits and risks plainly	Communicating complex systems to non-specialists

Honest scope: a personal learning project with AI-assisted coding. I directed the design and testing and can explain every component.

What this case study demonstrates

Systems analysis

- Process decomposition
- Requirements thinking
- Rule-based design
- Exception handling

Technical learning

- Python and APIs
- CCXT and dataframes
- Flask and Cloud Run concepts
- AI-assisted development

Testing

- Historical backtesting
- Parameter testing
- Error diagnosis
- Iterative refinement

Governance

- Environment variables
- API-key controls
- Risk identification
- Human oversight

I can learn unfamiliar technology, design modular automation, troubleshoot real technical issues and explain a complex system in a structured business-analysis format.

Personal learning project. Not financial advice. No guaranteed trading performance.