Trading Bot – Codebase Explanation (Beginner■Friendly)

This document explains every file you've created so far, what it's responsible for, how it interacts with the rest of the system, typical inputs/outputs, and common pitfalls. Keep it as a reference while you build features.

1) Big■Picture Overview

Your bot is organized as small modules with clear jobs:

• **config/** — reading environment variables and runtime settings.
• **core/** — cross■cutting utilities like logging.
• **exchange/** — a clean wrapper around the Bybit API (via pybit).
• **data/** — fetches and cleans market candles; computes indicators.
• **risk/** — turns "risk 1% per trade" into exact size and SL/TP levels.
• **orders/** — builds and submits bracket orders (entry + TP/SL), handles rounding to exchange steps.
• **strategy/** — strategies that output LONG/SHORT/FLAT decisions from data.
• **backtest/** — simulates strategy performance on historical candles.
• **analytics/** — summary metrics (win rate, Sharpe, drawdown, etc.).
• **cli/** — small scripts to run checks and demos quickly (no UI needed).

2) config/settings.py — Configuration Loader

Purpose: Single source of truth for settings. Reads secrets and flags from the .env file and validates them so other modules don't have to worry about it.

Main elements:
• Settings dataclass — typed container holding api_key, api_secret, testnet, log_level, runtime_mode, and db_path.
• load_env() — reads environment variables, converts them to correct types, and performs friendly checks (e.g., missing key → clear error).

Used by: Any module that needs keys/flags (exchange client, CLI scripts).

Typical output: A Settings object. Example: testnet=True, runtime_mode='paper'.

Common pitfalls:
• If .env isn't present or keys are wrong, load_env() raises an error.
• Always keep .env in .gitignore to avoid leaking secrets.

3) core/logger.py — Logger Factory

Purpose: Centralized logger setup so all parts of the bot produce consistent, timestamped logs to console and a rotating log file.

Main elements:
• get_logger(name, level) — returns a configured logging.Logger. Adds both console and file handlers (logs/app.log). Uses rotation to keep files small.

Used by: Almost every script/module (Bybit client, backtests, CLI).

Why it matters: Logs are the bot's "black box recorder." If an order fails or a crash happens, check logs to see exactly when and why.

4) exchange/bybit_client.py — Bybit API Wrapper

Purpose: One clean interface to Bybit so the rest of your code doesn't depend on the vendor SDK details.

Main capabilities:
• Session creation with API keys and testnet flag.
• ping(), server_time() to check connectivity.
• Market data: get_symbols(), get_ticker(), get_klines().
• Account data: get_balance(), get_positions().
• Trading: place_order(), cancel_order(), get_fills().
• Instrument rules: get_symbol_info(), get_min_qty(), get_tick_size().
• Helpers: _round_step(), place_postonly_limit().

Used by: Data fetching (candles), order executor (place/cancel orders), sanity scripts.

Inputs/Outputs: Python dicts matching pybit's v5 JSON structure.

Pitfalls:
• Wrong symbol/category leads to empty results.
• Post■Only orders auto■cancel if your price would cross the book; that's expected safety.

5) data/market_data.py — Candles & Indicators

Purpose: Fetch Bybit klines and convert them into a clean pandas DataFrame (OHLCV). Compute common indicators used by strategies.

Main functions:
• fetch_ohlcv(symbol, interval, limit, category) → DataFrame with index=datetime (UTC) and columns: open, high, low, close, volume. It reverses Bybit's newest■first list to oldest■first.
• add_sma(), add_ema() — moving averages on price.
• add_rsi() — Relative Strength Index (0–100).
• add_atr() — Average True Range (volatility).
• add_basic_indicators() — convenience to append SMA/EMA/RSI/ATR columns.

Used by: Strategy checks, backtests, and any analytics.

Pitfalls:
• Indicators need enough bars to warm up; early rows may be NaN.
• If fetch_ohlcv returns zero rows, check symbol/interval and network.

6) risk/manager.py — Position Sizing & Levels

Purpose: Turn human risk ideas into numbers your bot can trade: order size, and SL/TP levels.

Main functions:
• position_size(equity, risk_pct, stop_distance, contract_value=1.0) → quantity that risks exactly equity * risk_pct if SL is hit.
• propose_levels(entry, atr, atr_mult_sl, atr_mult_tp, side) → suggested SL/TP from ATR multipliers.
• validate_order(qty, min_qty, max_leverage=10) → safety checks before submitting.

Used by: Order executor and backtests.

Typical inputs: equity = 2000, risk_pct = 0.01, stop_distance = 1×ATR.
Typical outputs: qty (e.g., 0.1 contracts), levels like SL at entry – ATR, TP at entry + 2×ATR.

7) orders/executor.py — Build & Submit Bracket Orders

Purpose: Convert “Buy with 1% risk, SL=1×ATR, TP=2×ATR” into a concrete, exchange■valid order.

Main elements:
• BracketConfig — risk % and ATR multipliers.
• OrderExecutor — wraps the Bybit client and risk math to:
– read instrument rules (min qty, qty step, tick size)
– round qty and price correctly
– build a Limit order with tpslMode=Full, takeProfit, stopLoss
– submit, cancel (and a placeholder for amend).

Used by: Demo scripts and future live runner.

Pitfalls:
• If risk is tiny or ATR is big, rounded qty may drop below min; handle with a friendly message.
• Post■Only ensures you don’t accidentally cross the book; may auto■cancel if price is too aggressive.

8) strategy/base.py — Strategy Interface

Purpose: A small contract every strategy must implement so the rest of the system can call it generically.

Main elements:
• Strategy.warmup() — bars required before a valid signal.
• Strategy.generate_signal(df) — returns dict with signal (LONG/SHORT/FLAT), reason, and optional meta (e.g., ATR).

Used by: Backtests, paper/live runners.

9) strategy/sma_cross.py — SMA(20/50) Crossover

Purpose: Example strategy: when the fast SMA crosses above the slow SMA → LONG; when it crosses below → SHORT; otherwise FLAT. Also computes ATR for risk sizing.

Logic:
• Compute SMA(fast) and SMA(slow) on the last two bars.
• If yesterday fast ≤ slow and today fast > slow → bull cross (LONG).
• If yesterday fast ≥ slow and today fast < slow → bear cross (SHORT).
• Else → FLAT.
• Includes atr14 in meta for sizing.

Used by: Strategy check CLI, backtests.

10) backtest/engine.py — Historical Simulator

Purpose: Evaluate a strategy on past candles with realistic assumptions.

How it works:
• For each bar (after warmup), call generate_signal.
• If FLAT and you get LONG/SHORT, enter on the **next bar's open**.
• If IN POSITION, check the current bar's **high/low** for SL/TP hits (intrabar).
• Apply fees and slippage (basis■points model) on entry and exit.
• Only one position at a time (simplifies logic).
• Record each trade (entry/exit, side, PnL) and the equity curve over time.

Inputs: Clean OHLCV DataFrame, Strategy instance, BTConfig (risk %, ATR multipliers, costs).
Outputs: { trades: [...], equity_curve: pd.Series, final_equity: float }.

Assumptions & caveats:
• Entry at next open avoids look■ahead bias.
• If both SL and TP are within the same bar, SL wins (conservative).
• This is a single■position model — no pyramiding yet.

11) analytics/metrics.py — Performance Summary

Purpose: Turn raw trades and equity into understandable statistics.

Main outputs:
• **trades** (count), **final_equity** (ending account value).
• **winrate** (%) and **profit factor** (gross wins ÷ gross losses).
• **max_dd** (maximum drawdown as currency), **sharpe** (risk■adjusted return).
• Averages: **avg_win**, **avg_loss**.

Notes: Sharpe here is a simple approximation using equity % changes; good for quick comparisons.

12) cli/ — Utility & Demo Scripts

General rule: Run from the project root using python -m cli.script_name.

• **init_check.py** — Loads config and logger; prints mode/testnet. Quick sanity test.
• **bybit_check.py** — Pings Bybit, prints server time, symbols, ticker, a few candles, and wallet balance.
• **order_sanity.py** — Places a Post■Only limit order far from market and cancels it (safe lifecycle test).
• **indicators_check.py** — Fetches candles and writes a CSV with SMA/EMA/RSI/ATR to reports/.
• **risk_check.py** — Computes position size and SL/TP from mock numbers.
• **order_bracket_demo.py** — Builds a bracket (entry + TP/SL) using ATR and submits/cancels it.
• **strategy_check.py** — Runs SMA Cross on recent candles and prints LONG/SHORT/FLAT + ATR.
• **backtest_sma.py** — Runs a backtest and saves trades/equity CSVs to reports/.

13) How Modules Talk to Each Other

Typical end■to■end flow:

1) **data/market_data.py** fetches candles → DataFrame with indicators.
2) **strategy/*** looks at the latest bars → returns LONG/SHORT/FLAT + ATR.
3) **risk/manager.py** turns ATR and risk % into size + SL/TP distances.
4) **orders/executor.py** rounds qty/price to exchange steps → submits bracket order via **exchange/bybit_client.py**.
5) **backtest/engine.py** simulates this process historically; **analytics/metrics.py** summarizes results.
6) **cli/** scripts orchestrate calls for quick testing.

14) What’s Next (Roadmap)

• **paper/** — A loop that runs live with simulated fills: read the latest candle, generate a signal, size it, and log hypothetical trades (no real orders).
• **exec/live_runner.py** — The coordinator for real testnet orders (robust retries, heartbeats).
• **storage/db.py** — SQLite to log bars, orders, fills, trades for auditing and restart safety.
• **monitor/alerts.py** — Console alerts first, later Telegram/Slack for errors and daily PnL.