SCALPING BOT PROMPT (copy-paste everything below)

```
# Prompt: Build a Production-Ready Crypto **Scalping Bot** (Python 3.11+, Async, Replit-First)

**Role:** You are an expert Python engineer and high-frequency crypto trader.

**Goal:** Produce a complete, runnable **scalping bot** codebase from scratch, using modern async Python, strict order-routing, hybrid sim/live modes, robust logging, and small-balance friendliness. Default to **Simulation** mode, but fully support **Live** trading on Binance Spot. Include a tiny web status server to keep Replit awake. The resulting repo must run on Replit with one click and locally via scripts.

---

## Deliverables — Files to Output (exact names, with real code)
```

README.md
requirements.txt
.env.example
.replit
replit.nix
run.sh
config.py
logging_setup.py
utils.py
price_guard.py
price_normalizer.py
technical_analysis.py
portfolio.py
binance_api.py
coinbase_api.py # optional stub (parity interfaces)
order_router.py

execution_policy.py # simple slippage / taker policy
trade_executor.py # thin, deprecated wrapper to router
ws_price_feed.py
main.py # TraderEngine orchestrator
backtest/simulator.py
backtest/metrics.py
web/app.py # Quart server; starts trading engine as bg task
scripts/run_live.py
scripts/run_paper.py
scripts/run_sim.py
live_console.py
docs/REPLIT_SETUP.md
logs/.gitkeep
data/.gitkeep
scalping_strategy.py
tests/test_signals.py
tests/test_strategy_contract.py

Global Requirements

- **Python 3.11+**, fully **type-annotated** where public, concise docstrings for public classes/methods.
- **Async I/O** everywhere: one shared `aiohttp.ClientSession` with `ClientTimeout(total=Config.REQUEST_TIMEOUT)`. Close it cleanly on shutdown.
- **Retries/backoff & jitter** on network calls; log non-200 with details; respect 429s.
- **Structured Logging:** Use `structlog` for JSON logs (`logging_setup.get_logger`) and a CSV event logger (`logging_setup.get_sim_logger("simulation_events.csv")`). All trade/decision events must include: `ts, symbol, side, qty, price, fees, pnl, reason, mode`.
- **Modes:** Simulation (default), Paper-like (paper orders with live prices), and Live. Toggle with `Config.USE_LIVE_MODE` and `Config.LIVE_PRICES_IN_SIM`.
- **Safety rails:** Central **OrderRouter** enforces: min notional, position caps, **per-symbol dedupe** (inflight TTL), **sell cooldown**, qty/price normalization.

- **Fees & Slippage:** Default taker fee **0.10% per side** (`Config.DEFAULT_TAKER_FEE = 0.001`). ExecutionPolicy provides a simple slippage guard (bps) and taker/maker toggles (maker paths can be a no-op initially).
- **Small-Balance Friendly:** Implement **LOW_BALANCE_TEST_MODE** and **ULTRA_MICRO_MODE** logic. For tiny balances (~$240), compute minimal viable qty from **exchange filters** and fees; skip if below min-notional after rounding; log `min_notional_fail`.
- **Persistence:** Only use `./logs` and `./data` for runtime artifacts. Include `.gitkeep` files.
- **Graceful shutdown:** Handle SIGINT/SIGTERM; close websockets; stop tasks; flush logs; close client session.

Replit Environment (First-Class Support)

.replit
```ini
run = "bash run.sh"
```
replit.nix

```
{ pkgs }: {
  deps = [
    pkgs.python311
    pkgs.python311Packages.pip
    pkgs.glibcLocales
    pkgs.openssl
    pkgs.cacert
  ];
}
```
run.sh

```
#!/usr/bin/env bash
set -euo pipefail
python -V
python - <<'PY'
import os
```

```
print("[bootstrap] working dir:", os.getcwd())
PY
if [ ! -f .venv_created ]; then
  python -m pip install --upgrade pip
  pip install -r requirements.txt
  touch .venv_created
fi
# Launch the status web app which also spawns the trading
loop in background
exec python -m web.app
```

web/app.py requirements

- Use **Quart**.
- On startup, create and start the trading engine (**background task**).
- Serve `GET /` with a tiny status page (Mode, Equity, Cash, Win-rate, open positions).
- Also serve `GET /metrics` returning JSON: equity, cash, realized_pnl, open_positions, wins, losses, win_rate, uptime seconds, mode.

config.py — Keys & Tunables

- Load from env **first** (Replit Secrets), fallback to `.env` (local dev). Missing live keys must **not** crash; instead run in Simulation.
- Provide these classes and defaults:

```
class ApiKeys:
    BINANCE_API_KEY: str | None
    BINANCE_SECRET_KEY: str | None
    COINBASE_API_KEY: str | None
    COINBASE_PRIVATE_KEY: str | None   # PEM for ES256 JWT
    COINBASE_ORG_ID: str | None

class Config:
    USE_LIVE_MODE = False
    LIVE_PRICES_IN_SIM = True
    USE_MOCK_BALANCE = False
    USE_LIVE_BALANCE_IN_SIM = True
```

```
    SIM_STARTING_BALANCE = 10_000.0
    LOW_BALANCE_TEST_MODE = True
    ULTRA_MICRO_MODE = True

    HYPER_AGGRESSION_MODE = True
    DYNAMIC_STRICT_MODE = True
    WIN_RATE_TARGET = 0.70
    STRICT_MODE_AFTER_LOSSES = 2

    RISK_PER_TRADE = 0.02
    MAX_CONCURRENT_POSITIONS = 5
    MAX_DRAWDOWN_PCT = 0.10

    SYMBOLS =
["BTCUSDT","ETHUSDT","SOLUSDT","DOGEUSDT","XRPUSDT"]
    TA_LOOKBACK = 200
    SCAN_INTERVAL_SEC = 0.35

    DEFAULT_TAKER_FEE = 0.001
    EDGE_BUFFER_BPS = 5
    NANO_PROFIT_THRESHOLD = 0.0005   # +0.05%

    REQUEST_TIMEOUT = 10
    LOG_LEVEL = "INFO"
```

logging_setup.py

- `get_logger(name) -> structlog.BoundLogger` (JSON logs).
- `get_sim_logger(csv_filename)` returns an append-safe CSV writer with header: `ts,symbol,side,qty,price,fees,pnl,reason,mode`.

utils.py

- `now_ts()` (seconds), `clean_symbol()` (strip `/`, uppercase).
- Optional: `ExchangeSymbolMapper` (Binance `BTCUSDT` ↔ Coinbase `BTC-USD`).

- PnL helper: `pnl_pct(buy, current, fee_per_side)` = gross – (2*fee_per_side).

price_guard.py

- `ensure_price(price: float|None) -> float` (raise if `None` or ≤ 0).
- `within_bounds(v, lo, hi) -> bool`.

price_normalizer.py

- **Must use `Decimal` with `ROUND_DOWN`** for precision-safe snapping:
  - `normalize_price(price, tick_size) -> float`
  - `normalize_quantity(qty, step_size) -> float`
  - `check_min_notional(qty, price, min_notional) -> bool`

technical_analysis.py (vectorized)

- Indicators: RSI(14), EMA(9/20/50), VWAP, Bollinger(20,2).
- `analyze_symbol(klines_df)` returns dict with: `close, ema9, ema20, ema50, rsi, bb_ma, bb_upper, bb_lower, vwap`.

portfolio.py

- `Position(symbol, qty, buy_price, current_price)` with `update_price()`, `unrealized`, and `pnl_frac` based on `pnl_pct`.
- `Portfolio` tracks `cash`, positions map, `realized_pnl`, `wins/losses`, `win_rate`, `equity`.
- **Book fees to cash & PnL**:
  - On **buy**: `cash -= cost + taker_fee`

    - On **sell**: `cash += proceeds - taker_fee`; realized PnL includes exit fee.

- `can_open()` enforces `MAX_CONCURRENT_POSITIONS`.

binance_api.py (CRITICAL: Live-ready)

Implement a minimal, robust async client:

- `BASE_URL = "https://api.binance.com"` (allow easy override to `"https://testnet.binance.vision"`).
- Maintain a single shared `aiohttp.ClientSession`.
- **Server Time Sync**:
    - `async def sync_time()` calls `GET /api/v3/time` and stores `self._time_offset_ms = serverTime - localMs`.
    - Use `self._now_ms()` to stamp signed calls.
- **Signed Requests**:
    - `_request(method, endpoint, params, *, signed=False)`
    - For `signed=True`: set `recvWindow=5000`, include `timestamp=self._now_ms()`, compute **HMAC SHA256**over **query string** with secret, attach `signature`, set `X-MBX-APIKEY`.
    - **POST** must send **form-encoded** (`data=params`), not JSON. `GET` uses `params=params`.
- **Exchange Filters**:
    - `get_filters(symbol)` caches from `GET /api/v3/exchangeInfo?symbol=...`
    - Extract: `PRICE_FILTER.tickSize`, `LOT_SIZE.stepSize`, `NOTIONAL.minNotional` (fallback `MIN_NOTIONAL`where applicable).
    - In Simulation mode, return sane defaults if not cached.
- **Public Price/klines**:

    - `get_price(symbol)`:
        - SIM: random-walk drift stored per symbol (stable across calls).
        - LIVE: `GET /api/v3/ticker/price?symbol=...`
    - `get_klines(symbol, limit=200, interval="1m")`:
        - SIM: synth candles from drift.
        - LIVE: `GET /api/v3/klines` and map to dicts `{open,high,low,close,volume}` floats.
- **Order Placement**:
    - `place_order(symbol, side, qty, price=None)`:
        - Normalize qty/price using filters (Decimal snapping).
        - Enforce min-notional.
        - MARKET when `price is None`, else LIMIT with `timeInForce="GTC"`.
        - **LIVE**: `POST /api/v3/order` (signed).
        - **SIM/PAPER**: return a simulated immediate FILLED at current price.

coinbase_api.py (optional)

- Provide an interface-compatible stub:
    - `get_price(symbol)` → `RuntimeError` unless implemented.
    - `place_order(...)` → simulated FILLED in paper/sim modes.
    - Include symbol mapping via `ExchangeSymbolMapper` if used.

order_router.py (CRITICAL: central gate)

- CSV logger for every decision; structured JSON logs too.

- **Dedupe**: `_inflight` map (key like `BUY:BTCUSDT`) with TTL=1.5s blocks repeated actions.
- **Sell Cooldown**: `_last_sell_ts` map; block sells within 2s of last sell.
- **Risk Sizing**: `risk_qty = portfolio.equity * RISK_PER_TRADE / price`.
- **Normalization & Min-Notional**: use filters from `binance_api.get_filters`.
- **Public API**:

```
async def market_buy(self, symbol: str, qty: float | None, *, strategy: str) -> dict
```

- `async def market_sell(self, symbol: str, qty: float, *, strategy: str) -> dict`
-

- Log `reason` codes: `momentum_cross`, `nano_take_profit`, `tp1`, `tp2`, `stop_loss_breach`, `dedupe_blocked`, `sell_cooldown`, `min_notional_fail`, `position_cap`, `error:<msg>`.
- Mode label returns `"LIVE"` when `Config.USE_LIVE_MODE` else `"SIM"`.

execution_policy.py

- `ExecutionPolicy(slippage_bps: float = 5.0)` with:
  - `apply_slippage(price, side)` adjusts by ±bps.
  - `allow_taker()` returns `Config.HYPER_AGGRESSION_MODE`.

trade_executor.py

- Back-compat shim that warns it's deprecated and forwards to `OrderRouter`.

ws_price_feed.py

- **Async background loop** that keeps a dict of `last_price[symbol]`.
- In SIM: call `binance_api.get_price` at ~4 Hz; in LIVE a simple REST poller is fine (WS per-symbol streams are optional).
- **stop() must call `self._stop.set()`** and await task, so Replit Stop is graceful.
- `get_last_price(symbol)` returns latest or `None`.

scalping_strategy.py

- **Entry** (scalping momentum w/ safety):
  - `close > EMA9 > EMA20` and `RSI >= 50`.
  - **Dynamic strictness**: if total trades ≥10 and win-rate < `WIN_RATE_TARGET` and recent losses ≥ `STRICT_MODE_AFTER_LOSSES`, block new entries.
  - Only open if `symbol` not already in positions and `portfolio.can_open()`.
- **Exit**:
  - **Nano-profit**: if `pnl >= NANO_PROFIT_THRESHOLD` (+0.05%), close full.
  - Tiered: TP1 +0.15% (sell 50%), TP2 +0.30% (sell 50% of remainder).
  - Hard SL: –0.20% OR structure break (`EMA9 < EMA20` or `RSI < 45`).
  - PnL calc must subtract **2*fee** in edge math.
- **Loop**:
  - `scan_and_trade()` sweeps `Config.SYMBOLS` every `SCAN_INTERVAL_SEC`.
  - Use `ws_price_feed.get_last_price(symbol)` then REST fallback.
  - Use `binance_api.get_klines(...)` for TA.

- All actions go through `OrderRouter`.
- CSV log an event per symbol per sweep (`reason="scan"`).

main.py — TraderEngine

- Wire together:
  - `aiohttp.ClientSession` (shared)
  - `BinanceAPI` then `await api.sync_time()` (LIVE only), and lazy `get_filters` cache per symbol.
  - `Portfolio` (starting balance = `$240` if `LOW_BALANCE_TEST_MODE` else `SIM_STARTING_BALANCE`).
  - `WSPriceFeed` start/stop.
  - `OrderRouter` and `ScalpingStrategy`.
- **Circuit breaker** task:
  - Track equity high; if drawdown ≥ `MAX_DRAWDOWN_PCT`, **stop engine** and log `circuit_breaker_tripped`.
- **Graceful shutdown** on SIGINT/SIGTERM (works on Replit/Unix; tolerate NotImplemented on Windows).
- Provide `async def run_forever()` entrypoint.

backtest/simulator.py & metrics.py

- **Simulator** runs the live components for N minutes in Simulation, then writes `logs/sim_metrics.json` containing: equity, cash, realized_pnl, open_positions, wins, losses, win_rate.
- `metrics.py` has a small `Metrics` dataclass to compute win rate & drawdown (used if needed).

web/app.py

- Quart app; on startup, schedule the trading engine as a background task.
- `GET /` status page: Mode (LIVE/SIM), Equity, Cash, Win-rate, open positions.
- `GET /metrics` returns JSON metrics and uptime.

scripts/

- `scripts/run_sim.py --minutes N` → run Simulation for N minutes.
- `scripts/run_paper.py --minutes N` → paper-like run (still SIM router, but can sample live prices if `LIVE_PRICES_IN_SIM=True`).
- `scripts/run_live.py` → set `USE_LIVE_MODE=true` and run `main.run_forever()`.

tests/

- `tests/test_signals.py`:
  - `test_rsi_midline_behavior` (RSI on uptrend ends > 50)
  - `test_ema_ordering` (EMA9 > EMA20 > EMA50 on monotonic series)
- `tests/test_strategy_contract.py`:
  - Assert `ScalpingStrategy.scan_and_trade` exists and is `async`.

Run with `pytest -q`.

requirements.txt (pin versions)

```
aiohttp==3.10.5
pandas==2.2.2
numpy==1.26.4
python-dotenv==1.0.1
structlog==24.1.0
quart==0.19.6
```

```
websockets==12.0
uvloop==0.19.0; platform_system != "Windows"
pytest==8.3.2
```

README.md Outline

- Overview; risk warning (defaults to Simulation).
- Install and run (local & Replit).
- Env vars (BINANCE keys for live).
- Switching modes; logs & data locations.
- Tuning knobs: `NANO_PROFIT_THRESHOLD`, `SCAN_INTERVAL_SEC`, `HYPER_AGGRESSION_MODE`.
- Self-test commands.

Definition of Done & Self-Test

1. **Replit boots cleanly**: `bash run.sh` starts Quart on `PORT` and runs the engine in background.
2. **No secrets present**: Without keys, app runs in SIM; `/` and `/metrics` work; no crashes.
3. **Simulator works**: `python scripts/run_sim.py --minutes 5` finishes, writes `./logs/sim_metrics.json`.
4. **Paper run works**: `python scripts/run_paper.py --minutes 5` logs orders/decisions CSVs.
5. **Live-ready**:
   - `binance_api.sync_time()` used; `recvWindow=5000`; **POST /api/v3/order** is form-encoded; HMAC signature over query; `X-MBX-APIKEY` sent.
   - `exchangeInfo` filters are cached and used (tick/step/minNotional).
6. **WS resilience**: WS task stops gracefully; price polling fallback works.
7. **Compliance**: All orders normalized and pass min-notional; router blocks otherwise.

8. **Logging quality**: `./logs/` contains JSON and CSV logs with required fields.

9. **Tests pass**: `pytest -q` passes the two tests.

10. **Status page** shows Mode, open positions, win-rate, equity; `/metrics` returns JSON.

IMPORTANT: Default to Simulation; require explicit env toggles for live; highlight trading risk and user responsibility in README.