Cahier des charges détaillé — Application Python type “AI Signals” pour trading de memecoins (Solana)

1. Objectif du produit

Concevoir une application web et bot temps réel qui:

- Agrège en continu les nouveaux tokens/memecoins (ex. launchpads Pump.fun), la liquidité, le MC, les holders, les achats/ventes.
- Détecte et score des “Smart Signals” basés sur l’activité de portefeuilles suivis, de KOLs et de patterns on-chain.
- Permet d’acheter/vendre via wallet non-custodial, avec gestion du risk, take-profit/stop-loss et auto-sell.
- Fournit tableau de bord, listes d’adresses suivies, “Smart Activity”, force du signal, et narratifs AI.
- Expose une API et un bot (Telegram/Discord) pour alertes et exécutions automatiques.

Portée initiale: écosystème Solana; architecture extensible à d’autres chaînes EVM ultérieurement.

2. Personas et cas d’usage

- Trader avancé: veut capter early pumps, suivre “smart money”, exécuter en un clic ou auto-trader via règles.
- Analyste quant: backtest des stratégies, exports de données.
- Créateur KOL: publication de signaux à la communauté.

Cas d’usage clés:

- Veille en temps réel des mints Pump.fun et paires Raydium/Jupiter.
- Alertes quand X adresses suivies achètent Y montant dans Z minutes.
- Scoring d’un token (force 0–3x) selon métriques on-chain et ordre de marché.
- Achat instantané depuis la carte signal; auto-sell aux paliers TP/SL.
- Gestion d’une “watchlist” de 50–5,000 adresses, import/export CSV.
- Vue “Smart Activity” listant les derniers fills des top wallets avec prix, MC, quantité, valeur SOL/USDC.

- Rédaction d’un court narratif AI du token (optionnel, sans conseils financiers).

3. Fonctionnalités

3.1 Ingestion de données on-chain

- Sources:
  - Websocket RPC Solana (v1.18+), Geyser plugin (si dispo) pour transactions en temps réel.
  - Indexeurs: Helius, Triton, SolanaFM, Flipside (pour snapshots/archives).
  - DEX/AMM: Raydium, Orca; aggregator Jupiter API pour quotes/route.
  - Launchpad: Pump.fun API/events.
  - Market data: Pyth/Switchboard pour SOL/USDC.
- Événements captés:
  - Création de mint, première LP, ajout/retrait de liquidité, première cote sur DEX.
  - Swaps, transfers, holders uniques, top holders, renonciation mint authority, freeze authority, burn LP.
- Dédoublonnage, parsing, normalisation (schemas Avro/Parquet).

3.2 Moteur de signaux

- Règles:
  - “Push Time” (intensité des achats nets sur fenêtre 1–15 min).
  - “Smart Buy Burst”: N wallets suivis achètent ≥M SOL dans T minutes.
  - “Top10/No Mint/No Freeze/LP Burnt” filtres booléens.
  - Momentum prix/MC, profondeur LP, slippage estimé, route qualité.
- Score “Strength” (0.0–3.0):
  - pondération: activité smart money (40%), momentum (25%), qualité tokenomics/risques (15%), liquidité/impact (20%).
- Détection de risques:
  - Mint authority active, freeze authority active, taxes > X%, blacklist patterns, top holder > Y%, honeypot (simulate buy/sell).
- Narratif AI:
  - Génération courte (<= 400 chars) basée sur métadonnées on-chain + réseaux sociaux (optionnel), avec garde-fous de conformité.

3.3 Portefeuilles suivis (“Smart Money”)

- CRUD d’adresses:
  - Ajout manuel, import CSV/JSON, import depuis handle social (optionnel), alias, tags.
- Métriques:
  - Solde SOL/USDC, PnL agrégé, hit-rate 7/30/90j, taille moyenne, temps de détention médian.
- Classements:
  - Featured, Community, My Address; “Monitor Count”.
- Détection automatique de clusters (heuristiques graphes, ex. co-achat, funding).

3.4 Trading et exécution

- Wallet non-custodial:
  - Intégration Phantom/Solflare (web), clé locale chiffrée (bot/server-side signer optionnel avec policy).
- Trading:
  - Buy/Sell en un clic via Jupiter; gestion slippage, priorité fees, retry.
  - OCO: TP/SL multiples, trailing stop, DCA in/out.
  - Auto-trade rules: si Strength≥S et SmartBuy≥N alors buy Q SOL; auto-sell à TP/SL.
- Sécurité:
  - Pre-trade simulation, honeypot check, “dry-run” via RPC simulate.
  - Rate-limit, anti-sandwich (priority fee adaptatif, MEV-avoid si dispo).

3.5 Frontend Web

- Pages:
  - Dashboard “Public Signals”, “My Signals”.
  - Carte signal: graphique prix/MC, badges risques, force, derniers achats smart, liquidité, 24h volume.
  - Modale “Smart Activity” avec Time, Smart Money, Price, MC, Amount, Value.
  - Gestion des adresses suivies avec tri par balance, monitor count.
- UX:
  - Thème sombre, graphes lissés, micro-interactions.
  - Filtre global: chaîne, timeframe, critères de risque, tri par Strength.
- Internationalisation: EN/FR.

3.6 Bot Telegram/Discord

- Commandes:
  - /signal <token>, /buy, /sell, /watch <address>, /list, /rules.
- Alertes push: nouveaux signaux, fills des smart wallets, TP/SL exécutés.
- Mode “auto-trade” avec confirmation ou exécution directe selon policy.

3.7 Backtest et export

- Backtest sur historiques 30–180 jours:
  - Entrées: règles de signal, fenêtres, tailles.
  - Sorties: CAGR, max drawdown, hit-rate, Sharpe approximatif.
- Export CSV/Parquet des ticks, signaux, trades.

3.8 Rôles et permissions

- Anonyme (lecture publique limitée).
- Utilisateur enregistré: portefeuilles privés, règles, API key.
- Admin: gestion featured wallets, listes no-go, paramètres scoring.

4. Architecture technique

4.1 Vue d’ensemble

- Frontend: Next.js/React + TypeScript, WebSocket pour temps réel.
- Backend API: Python FastAPI.
- Workers streaming: Python (aiohttp, websockets), Rust optionnel pour débit élevé.
- Message bus: Redis Streams ou NATS; Kafka si >50k events/s.
- Stockage:
  - OLTP: PostgreSQL (TimescaleDB extension) pour séries temporelles.
  - Cache: Redis.
  - Data lake: S3/MinIO pour Parquet (backtests/archives).
- File d’exécution trades: worker “trader” isolé.
- Observabilité: Prometheus + Grafana, OpenTelemetry traces, Sentry.

4.2 Modules backend Python

- connectors/
  - solana_ws.py (subscriptions logs, program IDs Raydium/Token/AssociatedToken)
  - pumpfun.py (events/REST)

    - jupiter.py (quotes, swap)
    - raydium.py (pools, LP)
    - price_oracles.py (Pyth/Switchboard)
- parsers/
    - tx_decoder.py (anchor/idl si dispo, borsh)
- signals/
    - rules.py (définition de règles)
    - scorer.py (calcule Strength)
    - risk.py (honeypot, authorities, holder concentration)
- smartmoney/
    - registry.py (CRUD, tags, stats)
    - detector.py (burst detection)
- trading/
    - wallet.py (key mgmt, ed25519, bip44)
    - executor.py (route Jupiter, simulate, submit, retry)
    - risk_controls.py (limits, TP/SL, OCO)
- storage/
    - repositories.py (Timescale queries)
- api/
    - main.py (FastAPI routes, auth JWT/OAuth)
    - websockets.py (topics: signals, fills, prices)
- bots/
    - telegram_bot.py, discord_bot.py
- backtest/
    - engine.py (replay, metrics)
- ml/
    - narrative_llm.py (Appel modèle local/hosté, prompt safety)

4.3 Schéma de données (simplifié)

- tokens(id, mint, symbol, created_at, lp_created_at, lp_burnt, mint_authority, freeze_authority, metadata_uri)
- pools(id, token_mint, dex, base, quote, lp_amount, price, mc, liquidity_usd, vol_24h)
- ticks(token_id, ts, price, mc, liquidity, buys, sells, netflow_smart, holders)
- wallets(id, address, alias, tags, stats_json)
- smart_trades(id, wallet_id, token_id, side, amount, price, tx, ts)

- signals(id, token_id, score, reason_json, risk_level, window, ts)
- rules(id, user_id, json_def, enabled, created_at)
- trades(id, user_id, token_id, side, amount, avg_price, status, tp_sl_json, tx, ts)

Indexation sur ts, token_id, wallet_id; hypertables Timescale pour ticks/signals.

5. Algorithmes clés

5.1 Strength score (exemple)

Strength = 0.4*SmartBurst + 0.25*Momentum + 0.2*LiquidityQuality + 0.15*RiskScore

- SmartBurst: min(3, log10(total_SOL_smart_5m + 1) + k*unique_smart_buyers)
- Momentum: z-score du retour 1m et 5m lissé + accélération volume.
- LiquidityQuality: f(LP_USD, price_impact_1_SOL, depth@1%).
- RiskScore: 3 - pénalités (mint authority active, freeze active, top_holder>30%, tx_tax>5%, dev_wallet_sells).

Normalisation 0–3.

5.2 Détection “Smart Activity”

- Fenêtre glissante T=10m: agrège achats par wallet_tag in (“KOL”, “smartmoney”).
- Sort par ts desc; calcule prix moyen, MC au fill, quantité, valeur SOL.
- Déclencheur alerte si somme valeur ≥ V_min.

5.3 Honeypot/sellability check

- Simulation buy 0.1 SOL puis simulate sell 0.1 SOL via Jupiter/route la plus courte; si fail ou taxe > X%, marque “risky/untradeable”.

6. Sécurité et conformité

- Clés privées: chiffrement AES-256-GCM, KMS (HashiCorp Vault/AWS KMS), déverrouillage session.
- Permissions bot: trading désactivé par défaut; liste blanche tokens; limites quotidiennes.
- Rate limiting, WAF, protection DDoS.
- Logs d’audit immuables (WORM S3).
- Disclaimer clair; pas de conseils financiers; filtrage contenu généré (modération).

7. Performance et SLA

- Latence cible signal: < 800 ms du commit tx à notification.
- Débit cible: 5k events/s soutenu, 20k pics.
- Disponibilité: 99.9% web/API; reprise auto après RPC drop.
- Temps d'exécution swap: p50 < 2.0 s, p95 < 5.0 s.

8. Roadmap et MVP

MVP (6–8 semaines)

- Ingestion Pump.fun + swaps Raydium via Helius.
- Détection SmartBurst sur liste d'adresses importée.
- Scoring v0, cartes signaux, "Smart Activity", filtres risques basiques.
- Connexion wallet Phantom, buy/sell via Jupiter, TP/SL simple.
- Bot Telegram alertes.
- Stockage Timescale, exports CSV.

v1.1 (8–12 semaines)

- Backtests, auto-trade rules, trailing stops, OCO.
- Honeypot robuste, top holders parsing.
- Tableau "Addresses" avec monitor count, classement.
- Narratif AI basique et i18n FR/EN.

v1.2+

- Clustering de wallets, ML pour prédiction pumping probability.
- Discord bot, rôles/abonnements, multi-chain EVM.

9. Stack technique recommandée

- Python 3.11+
  - FastAPI, Pydantic v2
  - asyncio, aiohttp, websockets
  - solana-py, anchorpy
  - numpy, pandas, scikit-learn (backtest/ML léger)
  - psycopg, SQLAlchemy
  - redis, tenacity, orjson, uvloop
- Frontend: Next.js/React, Zustand/Redux, Recharts/ECharts, Tailwind/Chakra.

- Base de données: PostgreSQL 15 + TimescaleDB, Redis 7.
- DevOps: Docker, Docker Compose, k8s (prod), GitHub Actions, Helm.
- Observabilité: Prometheus, Grafana, Loki, OpenTelemetry, Sentry.
- Sécurité: Vault/KMS, JWT, 2FA, secrets scanning.

10. Interfaces API (exemples)

- GET /signals?min_score=1.5&filters=no_mint,no_freeze&limit=50
- GET /tokens/{mint}/timeseries?fields=price,mc,liquidity&window=1h
- GET /smart-activity?window=10m&tag=kol
- POST /trade/buy { mint, amount_sol, slippage_bps, tp_sl: {...} }
- POST /rules { definition_json }
- POST /wallets/import { csv_file }
- WS /stream: topics=signals,smart_activity,trades

Schémas Pydantic fournis au dev, réponses paginées, timestamps en ms, idempotency-key sur trades.

11. Maquettes UX (référence)

- “Public Signals” en cartes: graphique mini sparkline, strength badge (1.1x–2.8x), boutons Buy/Sell.
- Modale “Smart Activity”: tableau Time, Smart Money (tag cœur KOL), Price, MC, Amount, Value.
- Page “Addresses”: colonnes Address/Alias, Balance, Monitor Count, actions Monitor/Stop.

12. Tests et qualité

- Tests unitaires 80% sur modules parsing, scoring, risk, executor.
- Tests d’intégration contre RPC mock + sandbox Jupiter.
- Tests charge: k6/Gatling pour WS/API; chaos tests RPC drop.
- Canaries en prod, feature flags.

13. Déploiement

- Environnements: dev, staging, prod.
- CI/CD: lint (ruff, mypy), tests, build images, migrations Alembic, déploiement Helm.
- Blue/green ou canary, rollback auto.

14. Livrables

- Repo monorepo ou polyrepo (frontend, backend, workers).
- Documentation technique (OpenAPI, schémas DB, runbooks).
- Playbooks incident, tableaux Grafana, dashboards SLO.
- Scripts d'initialisation (seed wallets, règles démo).
- Jeu de données de test anonymisé.

15. Estimation effort (ordre de grandeur)

- MVP: 10–14 semaines/homme pour 3 profils (Backend Python, Frontend, DevOps).
- Coûts infra MVP/mois:
  - RPC dédié Solana: moyen/élevé selon débit.
  - DB m6g.large équivalent + Redis: moyen.
  - Stockage S3: faible.
  - Monitoring/Logs: faible à moyen.

16. Pseudocode Python critique

Ex. pipeline WebSocket → signal:

```
async def run_pipeline():
    async for tx in solana_ws.subscribe(program_ids=RAYDIUM_IDS+PUMPFUN_IDS):
        event = parse_tx(tx)
        await bus.publish("events.raw", event)

async def signal_worker():
    async for ev in bus.consume("events.raw"):
        update_state(ev)  # ticks, wallets, pools
        sig = maybe_emit_signal(ev)
        if sig:
            score = compute_strength(sig.context)
            if pass_risk(sig.token):
                await repo.signals.insert(sig.with_score(score))
                await ws.broadcast("signals", sig)
```

Ex. exécution trade:

```
async def execute_buy(mint, amount_sol, max_slippage_bps):
    route = await jupiter.quote_buy(mint, amount_sol)
    sim_ok = await simulate(route)
    if not sim_ok or route.slippage_bps > max_slippage_bps:
        raise TradeBlocked("Risk/Slippage")
    tx = await jupiter.swap(route, priority_fee=adaptive_fee())
```

```
    await repo.trades.insert({...,"tx": tx.signature})
    return tx.signature
```

17. Risques projet et mitigations

- Qualité RPC/latence: multi-fournisseurs + failover, backpressure.
- Honeypots/malware tokens: simulate avant trade, blocklists communautaires.
- Volatilité extrême: limites de taille, auto-sell, circuit-breakers.
- Conformité: disclaimers, logs, géo-restrictions si nécessaire.

18. Annexes

- Liste initiale de program IDs (Token Program, Associated Token Account, Raydium AMM/CP, Jupiter API endpoints).
- Barème par défaut du Strength, seuils No Mint/No Freeze/LP Burnt.
- Formats d'import CSV d'adresses: address,alias,tags.

Ce cahier des charges peut servir de base directe au développement Python (FastAPI + workers async), en couvrant les vues illustrées: cartes de signaux, activité smart money et gestion d'adresses, ainsi que la chaîne complète ingestion → scoring → alerting → exécution.

⁂

Meilleur "websocket" pour une app Solana type AI Signals

Pour ce cas d'usage (événements on-chain très rapides, détection de mints/LP/swaps et latence minimale), la hiérarchie suivante fonctionne le mieux en pratique.

Recommandation courte

- Priorité 1: WebSocket RPC dédié de Helius avec "Enhanced Transactions" + (option) Geyser stream.
- Priorité 2: Triton RPC WebSocket dédié.
- Priorité 3: Solana Labs/Anza RPC propre avec plugin Geyser (si tu gères l'infra).
- Fallback: QuickNode/Alchemy/NodesGuru WebSocket RPC.

Pourquoi

- Débit et latence: les flux WS des RPC "grand public" throttlent vite. Un plan dédié (Helius/Triton) avec WS + endpoints "transactions" réduisent la latence et les drops.
- Couverture d'événements: tu as besoin des logs de programmes (Token, Raydium, Pump.fun, Jupiter route programs), des signatures confirmées/finalisées, et idéalement des

comptes (pools LP). Les offres ci-dessus exposent subscribeLogs, signatureSub, accountSub avec bonne stabilité.

- Résilience: multi-providers + reconnection backoff, et possibilité d'upgrader vers Geyser (flux gRPC/WS direct depuis un validateur) si tu vises < 500 ms bout-à-bout.

Setup concret (prod)

- Provider principal: Helius
  - WS pour: logsSubscribe sur Program IDs (Token, Raydium AMM/CP/CLMM, Pump.fun), accountSubscribe sur comptes de pools/LP.
  - HTTP pour: getSignaturesForAddress, getTransaction (fallback/rehydrate), simulateTransaction.
- Provider secondaire: Triton (mêmes subscriptions) en hot-standby.
- Auto-failover: si taux d'erreurs WS > seuil ou lag > X s, bascule.
- Buffer: Redis Streams/NATS pour lisser les bursts 5–20k ev/s.
- Si tu autohéberges: validateur RPC avec Geyser + plugins (Kafka/Redis) et un pont WS interne pour tes workers Python.

Bonnes pratiques WS côté Python

- Client: websockets ou aiohttp + uvloop; pydantic/orjson pour parsing.
- Multiplexe: une seule connexion WS par provider, plusieurs subscriptions.
- Backoff: expo jitter; resubscribe automatique après reconnect.
- Idempotence: dédoublonne par signature/slot.
- Mesures: heartbeat ping/pong, métriques lag slot, events/s, drops.

Raccourci opérationnel

- Démarre avec Helius WS dédié. Ajoute Triton en backup. Quand le volume explose, migre certains flux critiques (ex. logs Raydium) vers un Geyser privé pour minimiser la latence et la variabilité.

⁂

Plan de livraison du code Python selon le cahier des charges

Ci-dessous, un squelette complet et prêt à étendre pour un MVP. Il couvre:

- API FastAPI (REST + WebSocket broadcast)
- Ingestion temps réel Solana (logs/pump.fun) via asyncio
- Bus d'événements (Redis Streams) pour découpler
- Moteur de signaux (burst d'achats "smart money", scoring, risques)

- Exécution d’ordres via Jupiter (quote + swap + simulation)
- Stockage PostgreSQL/Timescale avec SQLAlchemy
- Bot Telegram basique pour alertes

Le code est organisé en modules et tient en plusieurs fichiers. Chaque bloc est minimal mais fonctionnel; remplace les TODO par tes clés/IDs et complète les heuristiques.

Note: pour garder la réponse lisible, j’inclus uniquement l’essentiel. Dis-moi si tu veux un repo zip ou chaque fichier séparé.

1) pyproject.toml

```
[project]
name = "solai-signals"
version = "0.1.0"
requires-python = ">=3.11"
dependencies = [
  "fastapi==0.115.0",
  "uvicorn[standard]==0.30.6",
  "pydantic==2.9.2",
  "pydantic-settings==2.5.2",
  "orjson==3.10.7",
  "asyncpg==0.29.0",
  "SQLAlchemy==2.0.34",
  "alembic==1.13.2",
  "redis==5.0.8",
  "websockets==12.0",
  "aiohttp==3.10.3",
  "tenacity==9.0.0",
  "python-telegram-bot==21.6",
  "cryptography==43.0.1",
  "solana==0.34.3",
  "anchorpy==0.20.1",
  "numpy==2.1.1",
  "pandas==2.2.2",
  "httpx==0.27.2",
  "prometheus-client==0.20.0",
  "structlog==24.4.0",
]
```

2) config.py

```
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    # Infra
    POSTGRES_DSN: str = "postgresql+asyncpg://solai:solai@localhost:5432/solai"
    REDIS_URL: str = "redis://localhost:6379/0"

    # Providers
```

```
    HELIUS_WS: str = "wss://mainnet.helius-rpc.com/?api-key=YOUR_KEY"
    HELIUS_HTTP: str = "https://mainnet.helius-rpc.com/?api-key=YOUR_KEY"
    TRITON_WS: str | None = None  # optionnel

    # Jupiter
    JUPITER_BASE: str = "https://quote-api.jup.ag"
    JUPITER_SWAP: str = "https://quote-api.jup.ag/v6/swap"

    # Telegram
    TG_TOKEN: str | None = None
    TG_CHAT_ID: str | None = None

    # Runtime
    NETWORK: str = "mainnet"
    LOG_LEVEL: str = "INFO"

    class Config:
        env_file = ".env"

settings = Settings()
```

3) storage/db.py (SQLAlchemy + migrations prêtes)

```
from sqlalchemy.ext.asyncio import create_async_engine, async_sessionmaker, AsyncAttrs
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column
from sqlalchemy import String, BigInteger, JSON, Float, Integer, Index
from config import settings

engine = create_async_engine(settings.POSTGRES_DSN, pool_pre_ping=True)
SessionLocal = async_sessionmaker(engine, expire_on_commit=False)

class Base(AsyncAttrs, DeclarativeBase):
    pass

class Token(Base):
    __tablename__ = "tokens"
    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    mint: Mapped[str] = mapped_column(String(64), unique=True, index=True)
    symbol: Mapped[str | None] = mapped_column(String(32))
    metadata_uri: Mapped[str | None] = mapped_column(String(256))
    flags: Mapped[dict] = mapped_column(JSON, default={})  # no_mint, no_freeze, lp_burnt

class Tick(Base):
    __tablename__ = "ticks"
    id: Mapped[int] = mapped_column(primary_key=True)
    token_id: Mapped[int] = mapped_column(index=True)
    ts_ms: Mapped[int] = mapped_column(BigInteger, index=True)
    price: Mapped[float] = mapped_column(Float)
    mc: Mapped[float] = mapped_column(Float)
    liquidity: Mapped[float] = mapped_column(Float)
    buys: Mapped[int] = mapped_column(Integer, default=0)
    sells: Mapped[int] = mapped_column(Integer, default=0)

class Signal(Base):
```

```
    __tablename__ = "signals"
    id: Mapped[int] = mapped_column(primary_key=True)
    token_id: Mapped[int] = mapped_column(index=True)
    ts_ms: Mapped[int] = mapped_column(BigInteger, index=True)
    score: Mapped[float] = mapped_column(Float)
    reason: Mapped[dict] = mapped_column(JSON)

class SmartTrade(Base):
    __tablename__ = "smart_trades"
    id: Mapped[int] = mapped_column(primary_key=True)
    wallet: Mapped[str] = mapped_column(String(64), index=True)
    token_id: Mapped[int] = mapped_column(index=True)
    side: Mapped[str] = mapped_column(String(4))  # BUY/SELL
    amount: Mapped[float] = mapped_column(Float)
    price: Mapped[float] = mapped_column(Float)
    ts_ms: Mapped[int] = mapped_column(BigInteger, index=True)

Index("ticks_t_token_ts", Tick.token_id, Tick.ts_ms)

async def init_models():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)
```

4) bus/events.py (Redis Streams)

```
import json, time
from typing import AsyncIterator
import redis.asyncio as redis
from config import settings

STREAM_RAW = "events.raw"
STREAM_SIGNALS = "signals.out"
STREAM_SMART = "smart.activity"

class EventBus:
    def __init__(self):
        self.r = redis.from_url(settings.REDIS_URL, decode_responses=True)

    async def publish(self, stream: str, payload: dict):
        await self.r.xadd(stream, {"d": json.dumps(payload, separators=(",", ":"))})

    async def consume(self, stream: str, group: str, consumer: str) -> AsyncIterator[dict
        try:
            await self.r.xgroup_create(stream, group, id="$", mkstream=True)
        except Exception:
            pass
        last_id = ">"
        while True:
            resp = await self.r.xreadgroup(group, consumer, {stream: last_id}, count=100,
            if not resp:
                continue
            for _, msgs in resp:
                for msg_id, data in msgs:
                    yield msg_id, json.loads(data["d"])
```

```
                    await self.r.xack(stream, group, msg_id)

bus = EventBus()
now_ms = lambda: int(time.time()*1000)
```

5) connectors/solana_ws.py (abonnement logs)

```
import asyncio, json, websockets, structlog
from tenacity import retry, wait_exponential_jitter, stop_after_attempt
from config import settings

log = structlog.get_logger("solana.ws")

SUB_TEMPLATE = {
  "jsonrpc": "2.0",
  "id": 1,
  "method": "logsSubscribe",
  "params": [{"mentions": []}, {"commitment": "confirmed"}]
}

PUMP_FUN_PROGRAMS = [
    "C2xn1J…",  # TODO: vrais Program IDs
]
RAYDIUM_PROGRAMS = [
    "4Rh7y…",
]

@retry(wait=wait_exponential_jitter(1, 10), stop=stop_after_attempt(10))
async def stream_logs(on_event):
    async with websockets.connect(settings.HELIUS_WS, ping_interval=20, ping_timeout=20,
        # subscribe multiple programs
        sub_ids = []
        for progs in (PUMP_FUN_PROGRAMS, RAYDIUM_PROGRAMS):
            req = SUB_TEMPLATE | {"id": len(sub_ids)+1, "params": [{"mentions": progs}, {
            await ws.send(json.dumps(req))
            ack = await ws.recv()
            sub_ids.append(ack)

        log.info("subscriptions_ready", count=len(sub_ids))
        while True:
            raw = await ws.recv()
            evt = json.loads(raw)
            if evt.get("method") == "logsNotification":
                await on_event(evt["params"]["result"])
```

6) signals/rules.py (détection + scoring)

```
from dataclasses import dataclass
from typing import Dict, Any
import math, time
```

```
@dataclass
class Context:
    token_id: int
    buys_smart_sol_5m: float
    unique_smart_buyers_5m: int
    ret_1m: float
    ret_5m: float
    liquidity_usd: float
    price_impact_1sol_bps: float
    risk_penalties: float  # 0..2

def strength(ctx: Context) -> tuple[float, Dict[str, Any]]:
    smart = min(3.0, math.log10(ctx.buys_smart_sol_5m + 1.0) + 0.2*ctx.unique_smart_buyer
    momentum = max(0.0, 0.8*ctx.ret_1m + 0.2*ctx.ret_5m)  # très simple, à affiner
    liquidity_q = min(3.0, 0.002 * ctx.liquidity_usd - 0.001 * ctx.price_impact_1sol_bps)
    risk = max(0.0, 3.0 - ctx.risk_penalties)

    score = 0.4*smart + 0.25*momentum + 0.2*liquidity_q + 0.15*risk
    score = max(0.0, min(3.0, score))

    reason = {
        "smart": smart, "momentum": momentum, "liq": liquidity_q, "risk": risk,
        "inputs": ctx.__dict__
    }
    return score, reason
```

7) signals/engine.py (worker principal)

```
import asyncio, structlog
from bus.events import bus, STREAM_RAW, STREAM_SIGNALS, now_ms
from storage.db import SessionLocal, Signal
from signals.rules import Context, strength

log = structlog.get_logger("signals.engine")

# Mémoire courte des agrégats (à remplacer par Timescale requêtes)
state = {
    # token_id: { "smart_buy_sol": deque[(ts, amount)], ... }
}

def get_context_from_event(ev) -> Context | None:
    # TODO: convertir un raw log/tx en features agrégées
    # Exemple basique:
    token_id = ev.get("token_id")
    if not token_id:
        return None
    buys_smart = ev.get("smart_buy_sol_5m", 0.0)
    uniq = ev.get("uniq_smart_5m", 0)
    return Context(
        token_id=token_id,
        buys_smart_sol_5m=buys_smart,
        unique_smart_buyers_5m=uniq,
        ret_1m=ev.get("ret_1m", 0.0),
        ret_5m=ev.get("ret_5m", 0.0),
```

```
        liquidity_usd=ev.get("liquidity_usd", 0.0),
        price_impact_1sol_bps=ev.get("impact_bps", 50.0),
        risk_penalties=ev.get("risk_pen", 1.0),
    )

async def run():
    async for _, ev in bus.consume(STREAM_RAW, "signals", "signals-1"):
        ctx = get_context_from_event(ev)
        if not ctx:
            continue
        score, reason = strength(ctx)
        if score < 1.0:
            continue
        async with SessionLocal() as s:
            s.add(Signal(token_id=ctx.token_id, ts_ms=now_ms(), score=score, reason=reasc
            await s.commit()
        await bus.publish(STREAM_SIGNALS, {"token_id": ctx.token_id, "score": score, "rea
        log.info("signal", token_id=ctx.token_id, score=round(score,2))
```

8) trading/jupiter.py (quote + swap)

```
import httpx, structlog, base64, json, asyncio
from config import settings

log = structlog.get_logger("jupiter")

async def get_quote_buy(mint: str, amount_sol: float, slippage_bps: int = 150):
    amt = int(amount_sol * 1_000_000_000)  # lamports
    params = {
        "inputMint": "So11111111111111111111111111111111111111112",
        "outputMint": mint,
        "amount": amt,
        "slippageBps": slippage_bps,
        "onlyDirectRoutes": False
    }
    async with httpx.AsyncClient(timeout=10) as cx:
        r = await cx.get(f"{settings.JUPITER_BASE}/v6/quote", params=params)
        r.raise_for_status()
        return r.json()

async def swap(route: dict, user_pubkey: str, payer_signer: callable):
    # Build transaction
    async with httpx.AsyncClient(timeout=20) as cx:
        r = await cx.post(settings.JUPITER_SWAP, json={
            "quoteResponse": route,
            "userPublicKey": user_pubkey,
            "wrapAndUnwrapSol": True,
            "useSharedAccounts": True,
            "dynamicComputeUnitLimit": True,
            "prioritizationFeeLamports": "auto",
        })
        r.raise_for_status()
        j = r.json()
        tx_b64 = j["swapTransaction"]
```

```
        tx_bytes = base64.b64decode(tx_b64)
        # signer fourni par wallet.py
        signed = await payer_signer(tx_bytes)
        # envoyer via RPC HTTP (Helius)
        submit = await cx.post(settings.HELIUS_HTTP, json={
            "jsonrpc": "2.0", "id": 1, "method": "sendRawTransaction",
            "params": [base64.b64encode(signed).decode(), {"skipPreflight": False}]
        })
        submit.raise_for_status()
        sig = submit.json()["result"]
        log.info("swap_submitted", sig=sig)
        return sig
```

9) trading/wallet.py (signer Ed25519 simple)

```
from solana.rpc.async_api import AsyncClient
from solana.keypair import Keypair
from nacl.signing import SigningKey
from nacl.encoding import RawEncoder

class LocalWallet:
    def __init__(self, secret: bytes):
        self._kp = Keypair.from_secret_key(secret)

    @property
    def pubkey(self) -> str:
        return str(self._kp.public_key)

    async def sign_tx(self, tx_bytes: bytes) -> bytes:
        # Le payload Jupiter est déjà une tx v0; on signe avec la clé
        from solana.transaction import VersionedTransaction
        tx = VersionedTransaction.deserialize(tx_bytes)
        tx.sign([self._kp])
        return bytes(tx.serialize())
```

10) api/main.py (FastAPI + broadcast)

```
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
from fastapi.responses import ORJSONResponse
from storage.db import init_models, SessionLocal, Signal
from bus.events import bus, STREAM_SIGNALS
import asyncio, json

app = FastAPI(default_response_class=ORJSONResponse)
subscribers: set[WebSocket] = set()

@app.on_event("startup")
async def _startup():
    await init_models()
    asyncio.create_task(_forward_signals())
```

```
async def _forward_signals():
    async for _, ev in bus.consume(STREAM_SIGNALS, "ws", "ws-1"):
        dead = []
        for ws in subscribers:
            try:
                await ws.send_json(ev)
            except Exception:
                dead.append(ws)
        for ws in dead:
            subscribers.discard(ws)

@app.get("/health")
async def health():
    return {"ok": True}

@app.get("/signals")
async def list_signals(limit: int = 50):
    async with SessionLocal() as s:
        rows = (await s.execute(
            Signal.__table__.select().order_by(Signal.ts_ms.desc()).limit(limit)
        )).mappings().all()
        return [dict(r) for r in rows]

@app.websocket("/ws")
async def ws_endpoint(ws: WebSocket):
    await ws.accept()
    subscribers.add(ws)
    try:
        while True:
            await ws.receive_text()  # keepalive
    except WebSocketDisconnect:
        subscribers.discard(ws)
```

11) bots/telegram_bot.py

```
import asyncio
from bus.events import bus, STREAM_SIGNALS
from config import settings
from telegram import Bot

async def run_bot():
    if not settings.TG_TOKEN or not settings.TG_CHAT_ID:
        return
    bot = Bot(token=settings.TG_TOKEN)
    async for _, ev in bus.consume(STREAM_SIGNALS, "tg", "tg-1"):
        msg = f"Signal: mint={ev['reason']['inputs'].get('mint','?')} score={ev['score']:
        await bot.send_message(chat_id=settings.TG_CHAT_ID, text=msg)
```

12) workers/ingestor.py (branche les logs au bus)

```
import asyncio, structlog, time
from connectors.solana_ws import stream_logs
from bus.events import bus, STREAM_RAW, now_ms

log = structlog.get_logger("ingestor")

async def on_event(evt):
    # Transforme l'événement log en message brut simplifié
    # TODO: parser les logs Pump.fun/Raydium pour extraire mint, montant, wallet, etc.
    payload = {
        "token_id": evt.get("value", {}).get("signature", 0) % 1000000,  # placeholder
        "smart_buy_sol_5m": 3.2,  # TODO
        "uniq_smart_5m": 2,
        "ret_1m": 0.05,
        "ret_5m": 0.12,
        "liquidity_usd": 80000.0,
        "impact_bps": 60.0,
        "risk_pen": 0.5,
        "ts_ms": now_ms(),
    }
    await bus.publish(STREAM_RAW, payload)

async def run():
    await stream_logs(on_event)
```

13) app.py (lanceurs)

```
import asyncio, uvicorn
from workers.ingestor import run as run_ingestor
from signals.engine import run as run_signals
from bots.telegram_bot import run_bot

async def main():
    # Lancer API
    config = uvicorn.Config("api.main:app", host="0.0.0.0", port=8000, log_level="info")
    server = uvicorn.Server(config)
    # Lancer workers concurrents
    await asyncio.gather(
        server.serve(),
        run_ingestor(),
        run_signals(),
        run_bot(),
    )

if __name__ == "__main__":
    asyncio.run(main())
```

14) Commandes de démarrage

- Services requis: PostgreSQL + Timescale (optionnel), Redis.
- Variables: renseigner .env avec HELIUS keys, TG token, DSN DB.
- Lancer:

```
uvicorn api.main:app --reload  # pour dev API seule
python app.py                   # API + workers + bot
```

15) Étapes suivantes pour passer en “réel”

- Parser exact des Program IDs:
  - Pump.fun: décoder instructions pour capter les nouveaux mints et achats.
  - Raydium/Orca: décoder swaps, LP events, prix et MC; ou consommer un indexeur (Helius Enhanced Tx).
- Registry “smart money”:
  - Maintenir une table wallets(tag, alias) et incrémenter les features buys_smart_sol_5m par fenêtre glissante.
- Risk engine:
  - Lecture des comptes Mint (authority, freeze), top holders via indexeur, honeypot simulate (simulateTransaction + mini swap Jupiter dry-run).
- Exécution:
  - Intégrer Phantom/Solflare côté frontend pour signer; garder LocalWallet pour tests.
- Backtests:
  - Enregistrer ticks/quotes et rejouer les règles sur historique.
- Observabilité:
  - Ajouter endpoints /metrics Prometheus et dashboards.

Ce squelette respecte la chaîne ingestion → bus → moteur de signaux → stockage → diffusion → exécution, et il est optimisé pour être étendu rapidement vers une version production conforme au cahier des charges.

⁂

Code prêt à brancher — “Étapes suivantes pour passer en réel”

Objectif: remplacer les placeholders du MVP par des implémentations réelles pour Solana/Pump.fun/Raydium/Jupiter, le registre Smart Money, le risk engine (authorities, top holders, honeypot), et l’exécution côté client (Phantom) avec un backend sécurisé.

Le code ci-dessous ajoute/actualise uniquement les fichiers nécessaires. Il s'appuie sur l'ossature déjà fournie. Copie-les dans le même repo.

0) Prérequis et variables d'environnement

- Helius API key (HTTP + WS)
- Flipside/Helius Enhanced pour top holders (optionnel, sinon parsing direct)
- Redis, PostgreSQL opérationnels

.env

```
HELIUS_HTTP=https://mainnet.helius-rpc.com/?api-key=YOUR_KEY
HELIUS_WS=wss://mainnet.helius-rpc.com/?api-key=YOUR_KEY
PYTH_PRICE_FEED_SOL=J83GarJ...   # feed SOL/USD (optionnel)
```

1) connectors/pumpfun.py — détection mints et buys Pump.fun

```
# connectors/pumpfun.py
import json
from typing import Optional, Tuple

PUMPFUN_PROGRAM = "GPvJk...REPLACE"  # ProgramId réel pump.fun

# Minimal decoder: repère init mint / buy events depuis logs
def parse_pumpfun_logs(result: dict) -> Optional[dict]:
    # result: {"value": {"signature": "...", "logs": [...], "context": {"slot": ...}, "er
    val = result.get("value", {})
    logs = val.get("logs") or []
    sig = val.get("signature")
    slot = result.get("context", {}).get("slot")

    # Heuristique: les logs contiennent souvent "Initialize", "Create", "Buy"
    kind = None
    for l in logs:
        if "Initialize" in l or "create" in l.lower():
            kind = "MINT_INIT"
            break
        if "Buy" in l or "swap" in l.lower():
            kind = "BUY"
            # on laisse continuer pour extraire infos
    if not kind:
        return None

    # Extraction adresse mint depuis un log b58 (si présent)
    mint = None
    for l in logs:
        if "mint:" in l.lower():
            mint = l.split()[-1].strip()
            break
```

```
    return {
        "source": "pumpfun",
        "signature": sig,
        "slot": slot,
        "kind": kind,
        "mint": mint
    }
```

2) connectors/raydium.py — détection swaps Raydium et pool/LP

```
# connectors/raydium.py
from typing import Optional

RAYDIUM_CP_PROGRAM = "EoTcj6..."  # AMM v4/v5
RAYDIUM_CLMM_PROGRAM = "CAMMCg..."  # CLMM
TOKEN_PROGRAM = "TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA"  # SPL token

def parse_raydium_logs(result: dict) -> Optional[dict]:
    val = result.get("value", {})
    logs = val.get("logs") or []
    sig = val.get("signature")
    slot = result.get("slot") or result.get("context", {}).get("slot")

    # Identify swap by log message
    is_swap = any("swap" in l.lower() for l in logs)
    if not is_swap:
        return None

    # Heuristique: extraire mint via "mint" mot-clé (sera amélioré avec getTransaction)
    mint = None
    for l in logs:
        if "mint:" in l.lower():
            mint = l.split()[-1].strip()
            break

    return {
        "source": "raydium",
        "signature": sig,
        "slot": slot,
        "kind": "SWAP",
        "mint": mint
    }
```

3) connectors/helius_tx.py — enrichissement via getTransaction (Enhanced)

```
# connectors/helius_tx.py
import httpx
from config import settings

async def get_tx(signature: str) -> dict | None:
    # Utilise Helius Enhanced getTransaction
```

```
    payload = {
        "jsonrpc": "2.0", "id": 1, "method": "getTransaction",
        "params": [signature, {"maxSupportedTransactionVersion": 0}]
    }
    async with httpx.AsyncClient(timeout=15) as cx:
        r = await cx.post(settings.HELIUS_HTTP, json=payload)
        r.raise_for_status()
        j = r.json()
        return j.get("result")

async def get_account_info(account: str) -> dict | None:
    payload = {"jsonrpc":"2.0","id":1,"method":"getAccountInfo",
               "params":[account,{"encoding":"jsonParsed"}]}
    async with httpx.AsyncClient(timeout=15) as cx:
        r = await cx.post(settings.HELIUS_HTTP, json=payload)
        r.raise_for_status()
        return r.json().get("result")
```

4) smartmoney/registry.py — registre, import CSV, stats rapides

```
# smartmoney/registry.py
import csv
from typing import Iterable

SMART_TAGS = {"KOL","SMART","MM"}

class SmartRegistry:
    def __init__(self):
        self._set = set()         # addresses lower
        self.alias = {}           # addr -> alias

    def add(self, addr: str, alias: str | None = None):
        a = addr.lower()
        self._set.add(a)
        if alias:
            self.alias[a] = alias

    def load_csv(self, path: str):
        with open(path, newline="") as f:
            for row in csv.DictReader(f):
                self.add(row["address"], row.get("alias"))

    def contains(self, addr: str) -> bool:
        return addr.lower() in self._set

registry = SmartRegistry()
```

5) risk/engine.py — authorities, top holders, honeypot

```
# risk/engine.py
import httpx, base64
from typing import Tuple
from connectors.helius_tx import get_account_info
from config import settings

TOKEN_PROGRAM = "TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA"

async def spl_mint_risk(mint: str) -> dict:
    info = await get_account_info(mint)
    parsed = (info or {}).get("value", {}).get("data", {}).get("parsed", {})
    mint_info = parsed.get("info", {})
    freeze = mint_info.get("freezeAuthority")
    mint_auth = mint_info.get("mintAuthority")
    supply = int(mint_info.get("supply", "0"))
    decimals = int(mint_info.get("decimals", 0))
    return {
        "freeze_active": freeze is not None,
        "mint_active": mint_auth is not None,
        "supply": supply,
        "decimals": decimals,
    }

async def top_holders_heuristic(mint: str) -> dict:
    # Simple, à remplacer par un indexeur: on ne fait qu'un placeholder champ
    # Idéalement: Flipside query: top holders %, dev wallet concentration
    return {"top_holder_pct": 0.28, "top5_pct": 0.63}

async def honeypot_simulation(jupiter_quote: dict) -> Tuple[bool, float]:
    # On valide que la route existe et slippage estimé raisonnable
    routes = jupiter_quote.get("data") or []
    if not routes:
        return False, 10000
    best = routes[^4_0]
    est_bps = best.get("priceImpactPct", 0) * 10000
    return True, est_bps
```

6) workers/ingestor_real.py — pipeline réel WS → enrichissement → features

```
# workers/ingestor_real.py
import asyncio, json, websockets, structlog, time
from tenacity import retry, wait_exponential_jitter
from config import settings
from connectors.pumpfun import PUMPFUN_PROGRAM, parse_pumpfun_logs
from connectors.raydium import RAYDIUM_CP_PROGRAM, RAYDIUM_CLMM_PROGRAM, parse_raydium_lo
from connectors.helius_tx import get_tx
from smartmoney.registry import registry
from bus.events import bus, STREAM_RAW, now_ms

log = structlog.get_logger("ingestor.real")
```

```
def _sub_req(method: str, params: dict, idn: int):
    return {"jsonrpc":"2.0","method":method,"id":idn,"params":[params,{"commitment":"conf

@retry(wait=wait_exponential_jitter(1, 10))
async def run():
    programs = [PUMPFUN_PROGRAM, RAYDIUM_CP_PROGRAM, RAYDIUM_CLMM_PROGRAM]
    async with websockets.connect(settings.HELIUS_WS, ping_interval=20, ping_timeout=20,
        # subscribe logs for programs
        for i, pg in enumerate(programs, start=1):
            req = {"jsonrpc":"2.0","id":i,"method":"logsSubscribe",
                   "params":[{"mentions":[pg]}, {"commitment":"confirmed"}]}
            await ws.send(json.dumps(req))
            await ws.recv()  # ack

        log.info("ws_ready", subs=len(programs))
        while True:
            msg = json.loads(await ws.recv())
            if msg.get("method") != "logsNotification":
                continue
            res = msg["params"]["result"]
            # parse both
            evt = parse_pumpfun_logs(res) or parse_raydium_logs(res)
            if not evt:
                continue

            sig = evt["signature"]
            tx = await get_tx(sig)  # enrichissement précis
            if not tx:
                continue

            # Features: détecter acheteurs smart money (addresses dans tx)
            accts = (tx.get("transaction", {}) or {}).get("message", {}).get("accountKeys
            accts_lower = [a.get("pubkey","").lower() if isinstance(a, dict) else str(a).
            uniq_smart = len({a for a in accts_lower if registry.contains(a)})

            payload = {
                "token_mint": evt.get("mint"),
                "signature": sig,
                "slot": evt.get("slot"),
                "source": evt["source"],
                "kind": evt["kind"],
                "uniq_smart_5m": uniq_smart,
                # placeholders supplémentaires; le moteur de signaux agrègera
                "amount_sol": 0.0,
                "ts_ms": now_ms(),
            }
            await bus.publish(STREAM_RAW, payload)
```

7) signals/aggregator.py — fenêtres glissantes et features consolidées

```
# signals/aggregator.py
from collections import deque, defaultdict
import time

WINDOWS = {
    "1m": 60_000,
    "5m": 300_000
}

class FeatureState:
    def __init__(self):
        self.events = deque()  # (ts, amount_sol, uniq_smart)
        self.last_price = None
        self.last_price_1m = None
        self.last_price_5m = None

class Aggregator:
    def __init__(self):
        self.by_mint: dict[str, FeatureState] = defaultdict(FeatureState)

    def add(self, mint: str, ts_ms: int, amount_sol: float, uniq_smart: int, price: float
        st = self.by_mint[mint]
        st.events.append((ts_ms, amount_sol, uniq_smart))
        # purge
        now = ts_ms
        while st.events and now - st.events[^4_0] > WINDOWS["5m"]:
            st.events.popleft()
        if price is not None:
            st.last_price = price

    def features(self, mint: str, now_ms: int):
        st = self.by_mint[mint]
        # calc soma sur 5m
        buys_sol_5m = sum(a for (_, a, _) in st.events)
        uniq_5m = max((u for (_, _, u) in st.events), default=0)
        # momentum fictif (sera remplacé par prix réel)
        ret_1m = 0.0
        ret_5m = 0.0
        return {
            "buys_smart_sol_5m": buys_sol_5m,
            "uniq_smart_5m": uniq_5m,
            "ret_1m": ret_1m,
            "ret_5m": ret_5m
        }

agg = Aggregator()
```

8) signals/engine_real.py — moteur qui appelle le risk engine et publie des signaux

```
# signals/engine_real.py
import structlog, asyncio
from bus.events import bus, STREAM_RAW, STREAM_SIGNALS, now_ms
from storage.db import SessionLocal, Signal, Token
from signals.rules import Context, strength
from signals.aggregator import agg
from risk.engine import spl_mint_risk, top_holders_heuristic, honeypot_simulation
from trading.jupiter import get_quote_buy

log = structlog.get_logger("signals.real")

async def run():
    async for _, ev in bus.consume(STREAM_RAW, "signals-real", "sig-r1"):
        mint = ev.get("token_mint")
        if not mint:
            continue

        # Update feature aggregator (TODO: derive amount_sol from tx parsing)
        agg.add(mint, ev["ts_ms"], ev.get("amount_sol", 0.0), ev.get("uniq_smart_5m", 0))

        feats = agg.features(mint, ev["ts_ms"])

        # Risk checks (authorities, holders)
        r_info = await spl_mint_risk(mint)
        holders = await top_holders_heuristic(mint)

        risk_pen = 0.0
        if r_info["mint_active"]:
            risk_pen += 0.8
        if r_info["freeze_active"]:
            risk_pen += 0.8
        if holders["top_holder_pct"] > 0.35:
            risk_pen += 0.6

        # Quote Jupiter pour price impact (honeypot light)
        j_quote = await get_quote_buy(mint, 0.2, slippage_bps=200)
        tradable, impact_bps = await honeypot_simulation(j_quote)

        ctx = Context(
            token_id=0,  # remplacé par id DB après upsert
            buys_smart_sol_5m=feats["buys_smart_sol_5m"],
            unique_smart_buyers_5m=feats["uniq_smart_5m"],
            ret_1m=feats["ret_1m"],
            ret_5m=feats["ret_5m"],
            liquidity_usd= j_quote.get("data", [{}])[^4_0].get("outAmount", 0)/1e6 if j_q
            price_impact_1sol_bps=impact_bps or 0.0,
            risk_penalties=risk_pen
        )

        score, reason = strength(ctx)
        if score < 1.0 or not tradable:
            continue

        async with SessionLocal() as s:
```

```
            # upsert token
            q = await s.execute(Token.__table__.select().where(Token.mint==mint))
            row = q.mappings().first()
            if row:
                token_id = row["id"]
            else:
                res = await s.execute(Token.__table__.insert().values(mint=mint, flags={}
                token_id = res.scalar_one()
            s.add(Signal(token_id=token_id, ts_ms=now_ms(), score=score, reason=reason |
            await s.commit()

        await bus.publish(STREAM_SIGNALS, {"token_mint": mint, "score": score, "reason":
        log.info("signal_real", mint=mint, score=round(score,2))
```

9) api/wallet_bridge.py — pont de signature côté client (Phantom)

Backend n’exécute plus d’échange avec clé serveur. Il génère une transaction non signée via Jupiter et la renvoie au client pour signature Phantom, puis la soumet.

```
# api/wallet_bridge.py
from fastapi import APIRouter, HTTPException
import httpx, base64
from config import settings

router = APIRouter(prefix="/trade", tags=["trade"])

@router.post("/prepare-buy")
async def prepare_buy(mint: str, user_pubkey: str, amount_sol: float, slippage_bps: int =
    # Quote
    amt = int(amount_sol * 1_000_000_000)
    async with httpx.AsyncClient(timeout=15) as cx:
        q = await cx.get(f"{settings.JUPITER_BASE}/v6/quote", params={
            "inputMint": "So11111111111111111111111111111111111111112",
            "outputMint": mint,
            "amount": amt,
            "slippageBps": slippage_bps,
            "onlyDirectRoutes": False
        })
        q.raise_for_status()
        route = q.json()
        sw = await cx.post(settings.JUPITER_SWAP, json={
            "quoteResponse": route,
            "userPublicKey": user_pubkey,
            "wrapAndUnwrapSol": True,
            "useSharedAccounts": True,
            "dynamicComputeUnitLimit": True,
            "prioritizationFeeLamports": "auto",
            "asLegacyTransaction": False
        })
        sw.raise_for_status()
        j = sw.json()
        return {"swapTransaction": j["swapTransaction"]}
```

```
@router.post("/submit")
async def submit(signed_tx_b64: str):
    async with httpx.AsyncClient(timeout=15) as cx:
        r = await cx.post(settings.HELIUS_HTTP, json={
            "jsonrpc":"2.0","id":1,"method":"sendRawTransaction",
            "params":[signed_tx_b64, {"skipPreflight": False}]
        })
        r.raise_for_status()
        return {"signature": r.json()["result"]}
```

Brancher ce router dans FastAPI:

```
# api/main.py (ajout)
from api.wallet_bridge import router as trade_router
app.include_router(trade_router)
```

10) bots/telegram_bot.py — message enrichi

```
# bots/telegram_bot.py (remplacement de msg)
msg = (
  f"Signal {ev.get('token_mint','?')} "
  f"score={ev['score']:.2f} | "
  f"smart_5m={ev['reason'].get('inputs',{}).get('buys_smart_sol_5m',0):.2f} SOL | "
  f"risk={ev['reason'].get('risk',0):.2f}"
)
```

11) Lancement réel

Créer un nouveau lanceur pour la chaîne réelle:

```
# app_real.py
import asyncio, uvicorn
from api.main import app
from workers.ingestor_real import run as run_ingestor
from signals.engine_real import run as run_signals
from bots.telegram_bot import run_bot

async def main():
    server = uvicorn.Server(uvicorn.Config("api.main:app", host="0.0.0.0", port=8000))
    await asyncio.gather(
        server.serve(),
        run_ingestor(),
        run_signals(),
        run_bot(),
    )

if __name__ == "__main__":
    asyncio.run(main())
```

Run:

```
python app_real.py
```

12) Frontend signature Phantom (exemple minimal)

Côté front, appeler:

1. POST /trade/prepare-buy pour recevoir swapTransaction (b64)
2. window.solana.signAndSendTransaction(transaction) puis POST /trade/submit

Snippet JS:

```
const txb64 = (await fetch('/trade/prepare-buy', {method:'POST', headers:{'Content-Type':
  body: JSON.stringify({ mint, user_pubkey: window.solana.publicKey.toBase58(), amount_sc
})).json();

const tx = VersionedTransaction.deserialize(Buffer.from(txb64.swapTransaction, 'base64'))
const sig = await window.solana.signAndSendTransaction(tx);
await fetch('/trade/submit', {method:'POST', headers:{'Content-Type':'application/json'},
  body: JSON.stringify({ signed_tx_b64: sig.signature })});
```

13) Checklist production

- Failover WS: ajoute un second provider (Triton) et logique de bascule.
- Persist features en Timescale si redémarrage.
- Tuning backoff + heartbeat ping/pong.
- Circuit breaker: si impact_bps > 1500 ou freeze/mint authority actifs, bloquer auto-trade.
- Règles auto-trade: service séparé lisant STREAM_SIGNALS et appelant les endpoints trade bridgés.

Ce lot bascule l'ingestion et la génération de signaux sur des données réelles (logs WS + enrichissement getTransaction), ajoute un registre Smart Money, un risk engine pratique, un contrôle honeypot léger via Jupiter, et un pont de signature Phantom pour exécuter en non-custodial.

⁂

Code prêt à l'emploi — Checklist production

Objectif: durcir le MVP "réel" avec résilience, observabilité, sécurité et gouvernance de déploiement. Les blocs ci-dessous sont plug-and-play et complètent le repo précédent.

1) Observabilité complète

1.1 Middleware FastAPI avec Prometheus + correlation-id

files: api/obs.py

```
# api/obs.py
import time, uuid
from typing import Callable
from fastapi import Request, Response
from starlette.middleware.base import BaseHTTPMiddleware
from prometheus_client import Counter, Histogram, generate_latest, CONTENT_TYPE_LATEST
from fastapi import APIRouter

HTTP_REQS = Counter("http_requests_total", "HTTP requests", ["method","path","code"])
HTTP_LATENCY = Histogram("http_request_seconds", "Latency", ["method","path"])
WS_CONN = Counter("ws_connections_total", "WebSocket connections", ["endpoint"])

class MetricsMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request: Request, call_next: Callable):
        start = time.perf_counter()
        cid = request.headers.get("x-correlation-id") or str(uuid.uuid4())
        request.state.correlation_id = cid
        response: Response = await call_next(request)
        dur = time.perf_counter() - start
        HTTP_REQS.labels(request.method, request.url.path, response.status_code).inc()
        HTTP_LATENCY.labels(request.method, request.url.path).observe(dur)
        response.headers["x-correlation-id"] = cid
        return response

router = APIRouter()

@router.get("/metrics")
async def metrics():
    return Response(content=generate_latest(), media_type=CONTENT_TYPE_LATEST)
```

Brancher dans API:

```
# api/main.py
from api.obs import MetricsMiddleware, router as metrics_router
app.add_middleware(MetricsMiddleware)
app.include_router(metrics_router)
```

1.2 Exporter Redis/DB/Goroutines

files: ops/health.py

```
# ops/health.py
import asyncio, redis.asyncio as redis
from sqlalchemy.ext.asyncio import AsyncEngine
from fastapi import APIRouter
from storage.db import engine
```

```
from config import settings

router = APIRouter(prefix="/ops", tags=["ops"])

@router.get("/ping")
async def ping():
    return {"ok": True}

@router.get("/readiness")
async def readiness():
    r = redis.from_url(settings.REDIS_URL)
    await r.ping()
    async with engine.connect() as conn:
        await conn.execute("SELECT 1")
    return {"ready": True}
```

Brancher:

```
from ops.health import router as ops_router
app.include_router(ops_router)
```

2) Failover multi-RPC et supervision WS

2.1 Gestionnaire de providers et bascule

files: connectors/providers.py

```
# connectors/providers.py
import asyncio, time, json, websockets, structlog
from contextlib import asynccontextmanager
from dataclasses import dataclass

log = structlog.get_logger("providers")

@dataclass
class RpcProvider:
    name: str
    ws_url: str
    http_url: str

class WsSupervisor:
    def __init__(self, providers: list[RpcProvider]):
        self.providers = providers
        self.i = 0

    def current(self) -> RpcProvider:
        return self.providers[self.i]

    def next(self):
        self.i = (self.i + 1) % len(self.providers)
```

```
    @asynccontextmanager
    async def connect(self, max_size=8_000_000):
        prov = self.current()
        log.info("ws_connect", provider=prov.name)
        async with websockets.connect(prov.ws_url, ping_interval=20, ping_timeout=20, max
            yield prov, ws

    async def failover(self, reason: str):
        log.warning("ws_failover", reason=reason, from_provider=self.current().name)
        self.next()
```

Configure:

```
# config.py (ajouts)
from connectors.providers import RpcProvider
PROVIDERS = [
    RpcProvider("helius", HELIUS_WS, HELIUS_HTTP),
    # Ajouter Triton/Custom
    # RpcProvider("triton", "wss://...", "https://..."),
]
```

2.2 Ingestor avec failover et auto-resubscribe

files: workers/ingestor_failover.py

```
# workers/ingestor_failover.py
import asyncio, json, structlog
from tenacity import retry, wait_exponential_jitter
from connectors.providers import WsSupervisor
from config import settings
from bus.events import bus, STREAM_RAW, now_ms
from connectors.pumpfun import PUMPFUN_PROGRAM, parse_pumpfun_logs
from connectors.raydium import RAYDIUM_CP_PROGRAM, RAYDIUM_CLMM_PROGRAM, parse_raydium_lo
from connectors.helius_tx import get_tx

log = structlog.get_logger("ingestor.failover")
sup = WsSupervisor(settings.PROVIDERS)

SUBS = [
    ("logsSubscribe", {"mentions": [PUMPFUN_PROGRAM]}),
    ("logsSubscribe", {"mentions": [RAYDIUM_CP_PROGRAM]}),
    ("logsSubscribe", {"mentions": [RAYDIUM_CLMM_PROGRAM]}),
]

async def subscribe_all(ws):
    # renvoie les id -> topic
    ids = {}
    for i, (meth, param) in enumerate(SUBS, start=1):
        req = {"jsonrpc":"2.0","id":i,"method":meth,"params":[param, {"commitment":"confi
        await ws.send(json.dumps(req))
        ack = json.loads(await ws.recv())
        ids[i] = meth
    return ids
```

```
async def run():
    while True:
        try:
            async with sup.connect() as (prov, ws):
                await subscribe_all(ws)
                while True:
                    msg = json.loads(await ws.recv())
                    if msg.get("method") != "logsNotification":
                        continue
                    res = msg["params"]["result"]
                    evt = parse_pumpfun_logs(res) or parse_raydium_logs(res)
                    if not evt:
                        continue
                    # enrich
                    tx = await get_tx(evt["signature"])
                    if not tx:
                        continue
                    await bus.publish(STREAM_RAW, {
                        "token_mint": evt.get("mint"),
                        "source": evt["source"],
                        "kind": evt["kind"],
                        "signature": evt["signature"],
                        "slot": evt.get("slot"),
                        "uniq_smart_5m": 0,  # rempli par aggregator
                        "amount_sol": 0.0,
                        "ts_ms": now_ms(),
                    })
        except Exception as e:
            await sup.failover(str(e))
            await asyncio.sleep(1.5)
```

3) Circuit breakers et politiques de trading

files: trading/policy.py

```
# trading/policy.py
from dataclasses import dataclass

@dataclass
class TradePolicy:
    max_slippage_bps: int = 200
    max_price_impact_bps: int = 1200
    min_liquidity_usd: float = 20000.0
    block_if_freeze: bool = True
    block_if_mint_auth: bool = True
    daily_notional_limit_sol: float = 200.0
    per_trade_limit_sol: float = 2.0

class PolicyState:
    def __init__(self):
        self.day_used_sol = 0.0
    def can_spend(self, amt_sol: float, p: TradePolicy) -> bool:
        return amt_sol <= p.per_trade_limit_sol and self.day_used_sol + amt_sol <= p.dail
```

```
    def track(self, amt_sol: float):
        self.day_used_sol += amt_sol

policy = TradePolicy()
policy_state = PolicyState()
```

Exemple d'usage dans api/wallet_bridge.py:

```
# avant de préparer un trade
from trading.policy import policy, policy_state
if not policy_state.can_spend(amount_sol, policy):
    raise HTTPException(429, "Notional limits exceeded")
```

4) Anti-duplication/idempotence et exactly-once

files: bus/idempotency.py

```
# bus/idempotency.py
import redis.asyncio as redis
from config import settings

r = redis.from_url(settings.REDIS_URL)

async def seen_once(key: str, ttl_sec: int = 3600) -> bool:
    # setnx returns True if the key was set
    ok = await r.setnx(f"seen:{key}", "1")
    if ok:
        await r.expire(f"seen:{key}", ttl_sec)
    return not ok  # True if already seen
```

Usage dans ingestor:

```
from bus.idempotency import seen_once
if await seen_once(evt["signature"]):
    continue
```

5) Gestion des secrets et configuration

5.1 Support Vault/KMS (facultatif simple)

files: ops/secrets.py

```
# ops/secrets.py
import os, base64
from cryptography.fernet import Fernet

FERNET_KEY = os.getenv("FERNET_KEY")  # stocké dans KMS/Vault
```

```
def encrypt(b: bytes) -> str:
    return Fernet(FERNET_KEY).encrypt(b).decode()

def decrypt(s: str) -> bytes:
    return Fernet(FERNET_KEY).decrypt(s.encode())
```

Exemple: chiffrement d'une clé de test dans DB ou fichier.

6) Logging structuré + rotation

files: ops/logging.py

```
# ops/logging.py
import logging, sys, structlog, os

def setup_logging():
    level = os.getenv("LOG_LEVEL","INFO")
    logging.basicConfig(stream=sys.stdout, level=level)
    structlog.configure(
        processors=[
            structlog.processors.TimeStamper(fmt="iso"),
            structlog.processors.add_log_level,
            structlog.processors.StackInfoRenderer(),
            structlog.processors.format_exc_info,
            structlog.processors.JSONRenderer()
        ],
        logger_factory=structlog.stdlib.LoggerFactory(),
    )
```

Appeler au démarrage:

```
from ops.logging import setup_logging
setup_logging()
```

7) Tâches périodiques de maintenance

files: ops/scheduler.py

```
# ops/scheduler.py
import asyncio, time, structlog
from trading.policy import policy_state

log = structlog.get_logger("scheduler")

async def run_scheduler():
    while True:
        # Reset des compteurs quotidiens à 00:00 UTC
        now = time.gmtime()
        if now.tm_hour == 0 and now.tm_min < 2:
```

```
            policy_state.day_used_sol = 0.0
            log.info("policy_reset_daily")
            await asyncio.sleep(120)
        await asyncio.sleep(10)
```

Brancher dans main:

```
from ops.scheduler import run_scheduler
asyncio.create_task(run_scheduler())
```

8) Déploiement: Dockerfile + Compose

files: Dockerfile

```
FROM python:3.11-slim

ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1

RUN apt-get update && apt-get install -y build-essential && rm -rf /var/lib/apt/lists/*
WORKDIR /app

COPY pyproject.toml /app/
RUN pip install --upgrade pip && pip install -e .

COPY . /app

EXPOSE 8000
CMD ["python","app_real.py"]
```

files: docker-compose.yml

```
version: "3.8"
services:
  api:
    build: .
    environment:
      - POSTGRES_DSN=${POSTGRES_DSN}
      - REDIS_URL=${REDIS_URL}
      - HELIUS_HTTP=${HELIUS_HTTP}
      - HELIUS_WS=${HELIUS_WS}
      - LOG_LEVEL=INFO
    ports: ["8000:8000"]
    depends_on: [redis, db]
  redis:
    image: redis:7
    command: ["redis-server","--appendonly","yes"]
    ports: ["6379:6379"]
  db:
    image: timescale/timescaledb:latest-pg15
    environment:
```

```
      - POSTGRES_USER=solai
      - POSTGRES_PASSWORD=solai
      - POSTGRES_DB=solai
    ports: ["5432:5432"]
```

9) CI/CD GitHub Actions (tests + build + push)

files: .github/workflows/ci.yml

```
name: CI
on:
  push:
    branches: [ main ]
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with: { python-version: '3.11' }
      - name: Install
        run: |
          python -m pip install --upgrade pip
          pip install -e . pytest ruff mypy
      - name: Lint
        run: |
          ruff check .
          mypy --ignore-missing-imports .
      - name: Test
        run: pytest -q
      - name: Docker build
        uses: docker/build-push-action@v6
        with:
          context: .
          push: false
```

10) Durcissement WebSocket client

files: utils/ws_safe.py

```
# utils/ws_safe.py
import asyncio, json, websockets, structlog
log = structlog.get_logger("ws.safe")

async def recv_with_heartbeat(ws, timeout: float = 30.0):
    while True:
        try:
            msg = await asyncio.wait_for(ws.recv(), timeout=timeout)
            return json.loads(msg)
```

```
        except asyncio.TimeoutError:
            try:
                await ws.ping()
            except Exception as e:
                raise e
```

Utiliser `recv_with_heartbeat` dans les ingestors.

11) Rate limiting API

files: api/ratelimit.py

```
# api/ratelimit.py
import time, redis.asyncio as redis
from fastapi import Request, HTTPException
from starlette.middleware.base import BaseHTTPMiddleware
from config import settings

r = redis.from_url(settings.REDIS_URL)

class RateLimitMiddleware(BaseHTTPMiddleware):
    def __init__(self, app, rpm: int = 120):
        super().__init__(app)
        self.rpm = rpm

    async def dispatch(self, request: Request, call_next):
        key = f"rl:{request.client.host}"
        cur = await r.incr(key)
        if cur == 1:
            await r.expire(key, 60)
        if cur > self.rpm:
            raise HTTPException(429, "Too Many Requests")
        return await call_next(request)
```

Brancher:

```
from api.ratelimit import RateLimitMiddleware
app.add_middleware(RateLimitMiddleware, rpm=300)
```

12) Feature flags et désactivation à chaud

files: ops/flags.py

```
# ops/flags.py
import redis.asyncio as redis
from config import settings

r = redis.from_url(settings.REDIS_URL)

async def flag_enabled(name: str, default: bool = True) -> bool:
```

```
    v = await r.get(f"flag:{name}")
    if v is None:
        return default
    return v == "1"

async def set_flag(name: str, enable: bool):
    await r.set(f"flag:{name}", "1" if enable else "0")
```

Exemple:

```
from ops.flags import flag_enabled
if not await flag_enabled("signals_emit", True):
    continue
```

13) Table d'audit immuable (WORM S3 simplifié)

files: ops/audit.py

```
# ops/audit.py
import json, time, os, aiofiles, hashlib

AUDIT_DIR = os.getenv("AUDIT_DIR","./audit")

async def append_audit(event: dict):
    os.makedirs(AUDIT_DIR, exist_ok=True)
    ts = int(time.time()*1000)
    fn = os.path.join(AUDIT_DIR, f"{ts}-{hashlib.sha1(json.dumps(event, sort_keys=True).e
    async with aiofiles.open(fn, "w") as f:
        await f.write(json.dumps(event, separators=(",",":"))+"\n")
```

Appeler à chaque trade/alerte critique.

14) Séparation des processes

- API: uvicorn workers 2–4
- Ingestors: 1–2 réplicas
- Signals engine: 1 replica (ou partitionner par mint hash)

Exemple Procfile (pour scaling via PM2/Supervisor):

```
api: uvicorn api.main:app --host 0.0.0.0 --port 8000 --workers 2
ingestor: python -m workers.ingestor_failover
signals: python -m signals.engine_real
bot: python -m bots.telegram_bot
```

15) Tests rapides de prod

files: tests/test_health.py

```
def test_dummy():
    assert True
```

Smoke test après déploiement:

- GET /ops/readiness → 200
- GET /metrics → exposé
- WS ingestor: logs “ws_ready”
- Génération d’un signal dans Redis → reçu par /ws clients

Ces ajouts couvrent la checklist de production: observabilité Prometheus, endpoints de santé, failover multi-RPC, idempotence, rate limiting, politiques de trading, gestion basique des secrets, logs structurés, tâches planifiées, conteneurisation et CI. Branche directe sur ton repo et déploie via Docker Compose ou Kubernetes avec readiness/liveness probes sur /ops/readiness et /metrics.

⁂

Affiche un schéma du projet

Voici un schéma d’architecture logique du projet (vue haut niveau). Copie-colle ce bloc dans un fichier .md ou dans un outil de diagrammes ASCII/Monodraw.

Architecture globale (temps réel)

[Utilisateur/Clients]

- Web App (React/Next.js)
- Bot Telegram/Discord
- Trader (Phantom/Solflare)

  ```
  | WebSocket + REST
  v
  ```

[API Gateway (FastAPI)]

- Endpoints REST (signals, tokens, trade/prepare, trade/submit)
- WebSocket broadcast /ws
- Middleware Observabilité (Prometheus /metrics, /ops/readiness)
- Rate limiting, Correlation-ID
  | ^
  | publish/consume | subscribe

v |
[Redis Streams (bus)]

- events.raw
- signals.out
- smart.activity
  | ^
  | |
  v |
  [Workers Ingestion] --------------------
- WS Supervisor (Helius, Triton) avec failover
- Subscriptions logs: Pump.fun, Raydium (CP/CLMM), Token Program
- Enrichissement getTransaction (Helius Enhanced)
- Idempotence (signature), parsing, publication events.raw

```
|
v
```

[Moteur de Signaux]

- Agrégateur fenêtres 1m/5m (smart buys, netflow)
- Risk engine: authorities (mint/freeze), top holders, honeypot (Jupiter)
- Quote/impact (Jupiter)
- Scoring Strength 0–3
- Écrit DB (signals) + publie signals.out

```
|                                   ^
v                                   |
```

[Base de Données] |

- PostgreSQL + TimescaleDB (tokens, ticks, signals, trades, wallets)
- Requêtes API (list_signals, timeseries) |

```
|                                   |
v                                   |
```

[Bot(s)] ⟵ consomme signals.out ----------

```
|
v
```

[Pont de Trade Non-Custodial]

- POST /trade/prepare-buy → récupère route Jupiter + tx non signée

- Client signe via Phantom/Solflare
- POST /trade/submit → sendRawTransaction (RPC)

```
|
v
```

[Providers externes]

- RPC/WS: Helius (+ Triton en backup)
- Jupiter Quote/Swap API
- Oracles (Pyth/Switchboard, optionnel)
- Indexeur holders (Flipside/Helius, optionnel)

Sécurité & Ops transverses

- Secrets: Vault/KMS (clé Fernet), variables .env
- Logs structurés (structlog), audits WORM
- Policies de trading (limites per-trade/journalières, circuit breakers)
- CI/CD GitHub Actions, Docker/Compose/K8s
- Healthchecks: /ops/readiness, /metrics

Flux principal

1. WS Ingestor reçoit logs (Pump.fun/Raydium) → enrichit → events.raw
2. Signals Engine agrège + calcule risques/score → DB + signals.out
3. API diffuse en temps réel sur /ws et sert REST
4. Bot/Front reçoit la carte signal; l'utilisateur achète
5. API prepare-buy (Jupiter) → wallet signe → submit → on-chain

Couches de données

- Chaude: Redis Streams (micro-lots temps réel)
- Tiède: TimescaleDB (séries temporelles, backtests)
- Froid (optionnel): S3/MinIO Parquet (archives, ML/backtest étendus)

Ce diagramme reflète les modules livrés dans le code: connectors (WS/Helius), signals (aggregator/rules/engine), trading (jupiter/policy), api (REST/WS/metrics), bus (Redis), storage (SQLAlchemy), ops (logging, flags, scheduler), bots.

⁂

1. image.jpg
2. image.jpg
3. image.jpg
4. image.jpg
5. image.jpg

6. image.jpg
7. image.jpg
8. image.jpg
9. image.jpg
10. image.jpg
11. image.jpg
12. image.jpg
13. image.jpg
14. image.jpg
15. image.jpg
16. image.jpg
17. image.jpg
18. image.jpg
19. image.jpg
20. image.jpg
21. image.jpg
22. image.jpg
23. image.jpg
24. image.jpg