Autonomous Crypto Trading System — Strategic Blueprint v3.1 FINAL

"DARK FOREST" PREDATOR ARCHITECTURE — PRODUCTION EDITION

Document Version: 3.1 FINAL (Expert-Reviewed Production Edition)
Date: January 2026
Classification: Personal Use / Production-Ready
Capital Trajectory: $500 → $10,000+
Location: Qatar (No KYC/Tax Considerations)
Architecture: Hybrid Defensive-Offensive / Graph-Mempool-Atomic

---

⚠️ CRITICAL RULES (READ FIRST)

> **PARTIAL HYDRA WARNING:**
> "NEVER allow HEAD 3 (Assassin) to execute unless HEAD 2 (Simulator) has blocked ≥95% of eventual losers in shadow mode."
> If simulation accuracy <95%, Assassin MUST stay disabled.
>
> **SCAVENGER, NOT WOLF:**
> You have 500ms advantage over retail, 500ms DISadvantage vs MEV bots.
> Target re-accumulation (1-2 hours after launch), not launch sniping.
>
> **INFRASTRUCTURE CAP:**
> Infrastructure cost must be <5% of capital annually. Delay expensive components until capital justifies them.

Table of Contents

1. Executive Summary
2. V2.5 → V3.0 Pragmatic Progression Path
3. Part A: Battle-Hardened Refinements
4. Part B: Hydra Architecture (Fixed)
5. Part C: Counter-Adaptation & Safety Systems
6. Part D: Day 1 Execution Plan
7. Part E: Infrastructure Economics & Realistic Timeline
8. Appendices

---

1. Executive Summary

Philosophy Statement (Revised)

> **V2.0 Mindset (Trader):** “Whale bought → I wait for API → I buy”
> **Result:** You’re exit liquidity for those with faster information.
>
> **V2.5 Mindset (Informed Trader):** “Whale bought → I simulate → I verify graph confidence → I buy”
> **Result:** You filter out 90% of traps, but still play a follower’s game.
>
> **V3.0 Mindset (Predator):** “I see the trap being set → I take the bait before the victim does”
> **Result:** You become the information edge—but now adversaries can see YOU.
>
> **V3.1 Mindset (Ghost Predator):** “I see, I act, I vanish. My patterns are noise.”
> **Result:** You extract value while remaining invisible to counter-adaptation.

The crypto market is a **Dark Forest**—a zero-sum environment where visibility means vulnerability. V3.1 transforms us from prey to predator while maintaining **epistemic humility**: our edge will decay, our signals will be poisoned, and our methods will be discovered. We build in counter-measures from day one.

What's New in V3.1 (Expert Feedback Integration)

Category	Addition	Impact
Counter-Adaptation	Signal Entropy Injection	Prevents pattern detection by adversaries
Safety	Graph Kill Switch	Emergency halt on epistemic failure
Trust Decay	Insider Half-Life (30 days)	No eternal S-Tier wallets
Metrics	False-Positive Cost Tracking	Auto-reduce V3 allocation on failures
Anti-Poison	Graph Poisoning Protection	Slow trust gain, fast trust loss
Infrastructure	Geyser Cost Fix (Helius web-hooks)	$0 mempool monitoring
Data Quality	CEX Mixer Blacklist	Filter untrackable CEX-funded wallets
Pragmatism	SQLite → Neo4j migration path	Start simple, scale later
Pragmatism	Rust Sidecar Pattern	2-4 weeks Rust, not 6 months
Timeline	Realistic 24-36 month roadmap	Honest expectations
Execution	Day 1 concrete scripts	Start TODAY
Validation	Backtest Framework	Historical replay before live

The Three Pillars (V3.1 Architecture)

```
┌──────────────────────────────────────────────────────────────────┐
│                V3.1 "GHOST PREDATOR" ARCHITECTURE                │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│  ┌────────────────────────────────────────────────────────────┐  │
│  │              COUNTER-ADAPTATION LAYER (NEW)                │  │
│  │ • Signal Entropy Injection (random delays, probabilistic skipping) │
│  │ • Rotating Execution Wallets (3-5 wallet rotation)         │  │
│  │ • Graph Kill Switch (epistemic failure detection)          │  │
│  └────────────────────────────────────────────────────────────┘  │
│                              │                                   │
│    ┌──────────────┐   ┌──────────────┐   ┌──────────────┐        │
│    │  HEAD 1:     │   │  HEAD 2:     │   │  HEAD 3:     │        │
│    │ SYBIL HUNTER │   │  SIMULATOR   │   │  ASSASSIN    │        │
│    │ (SQLite→Neo4j)│  │  (Local Fork)│   │  (Jito Bundles)│      │
│    │              │   │              │   │              │        │
│    │ + Decay 30d  │   │ + 95% accuracy│  │ + Rust Sidecar│       │
│    │ + CEX Blacklist│ │   REQUIRED   │   │   Pattern    │        │
│    │ + Poison Prot.│  │   before live│   │              │        │
│    └──────┬───────┘   └──────┬───────┘   └──────┬───────┘        │
│           │                  │                  │                │
│           ▼                  ▼                  ▼                │
│   Track wallet        MUST block ≥95%    Atomic execution        │
│   PROVENANCE          of losers in       ONLY when:              │
│   with decay          shadow mode        - Simulator ≥95%        │
│                       BEFORE Assassin    - Capital >$5K          │
│                       goes live          - Win rate >55%         │
│                                                                  │
│   ═════════════════════════════════════════════════════════════  │
│                                                                  │
│                 ┌──────────────────────────────┐                 │
│                 │ V2.0 DEFENSIVE FOUNDATION    │                 │
│                 │ (NEVER DISABLED)             │                 │
│                 │ • Fail-closed gates          │                 │
│                 │ • Veto system (not weights)  │                 │
│                 │ • Risk limits & kill switch  │                 │
│                 │ • Spread veto, panic exits   │                 │
│                 └──────────────────────────────┘                 │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Capital-Phase Mapping (Revised)

Phase	Capital	Timeline	Active Components	Risk Profile
Phase 0	$0 (Paper)	Months 1-3	Logger + Shadow Executor	Zero
Phase 1	$500	Months 4-8	V2.0 + Graph Harvest (SQLite)	Defensive
Phase 2	$2,000	Months 9-14	+ Simulator (shadow logging)	Defensive+
Phase 2.5	$3,500	Months 15-18	V2.0 + Simulation + Graph as confidence boost	Hybrid-Defensive
Phase 3	$5,000	Months 19-24	+ Jito Bundles (Live) IF metrics pass	Hybrid
Phase 4	$10,000+	Months 25-36+	Full Hydra + Counter-Adaptation	Offensive

Latency Reality Check (Know Your Position)

```
FOOD CHAIN POSITIONING:

                                          SPEED
                                            ↓
┌─────────────────────────────────────────────────────────────┐
│  TOP PREDATORS: Institutional MEV                           │
│  Latency: <50ms total                                       │
│  Tools: Co-located servers, custom Jito clients             │
│  Strategy: Front-run everything                             │
├─────────────────────────────────────────────────────────────┤
│  MID PREDATORS: Serious MEV bots                            │
│  Latency: 50-200ms                                          │
│  Tools: Optimized Rust, private RPCs                        │
│  Strategy: Back-run, sandwich                               │
├─────────────────────────────────────────────────────────────┤
│  ▶▶▶ YOU (V3.1): Ghost Scavenger ◀◀◀                        │
│  Latency: ~500ms total                                      │
│  Tools: Python brain + Rust trigger                         │
│  Strategy: RE-ACCUMULATION plays (1-2h after launch)        │
│            Graph-based insider detection                    │
│            Pick up what wolves leave behind                 │
├─────────────────────────────────────────────────────────────┤
│  PREY: API-polling retail                                   │
│  Latency: 2-5 seconds                                       │
│  Tools: Free APIs, manual trading                           │
│  Strategy: Hope                                             │
└─────────────────────────────────────────────────────────────┘

YOUR EDGE: Information (who knows what), not speed.
YOUR NICHE: Tokens 1-24 hours old, re-accumulation signals.
```

---

2. V2.5 → V3.0 Pragmatic Progression Path

The Honest Roadmap

V3.0 in 18 months was optimistic. Here’s the realistic path:

```
TIMELINE REALITY:

Year 1 (Months 1-12): FOUNDATION
├── Months 1-3:   Phase 0 - Paper trading, logging
├── Months 4-6:   Phase 1 - Live V2.0 ($500), begin graph data harvest
├── Months 7-9:   Continue V2.0, SQLite graph growing
├── Months 10-12: Add simulation gate (shadow mode)

Year 2 (Months 13-24): V2.5 HYBRID
├── Months 13-15: V2.5 live (V2.0 + simulation + graph confidence boost)
├── Months 16-18: Evaluate metrics, begin Rust sidecar learning
├── Months 19-21: If metrics pass → test Jito bundles (small $)
├── Months 22-24: Gradual V3.0 component activation

Year 3 (Months 25-36+): FULL V3.0 (CONDITIONAL)
├── Only if: Capital >$5,000, Win rate >55%, Positive ROI after infra costs
├── Full Hydra with counter-adaptation layer
├── Consider Neo4j migration (if >5,000 tracked wallets)
└── Continuous adversarial adaptation
```

V2.5: The Critical Intermediate Stage

V2.5 is not V3.0-lite. It's the **proving ground** where you validate that graph signals actually help.

```
"""
V2.5 ARCHITECTURE:
- V2.0 core logic (unchanged, battle-tested)
- Simulation gate (MANDATORY before any trade)
- Graph signals as ADDITIONAL confidence multiplier (not primary signal)
"""

@dataclass
class V25Config:
    """V2.5 Configuration - The Pragmatic Middle Ground"""

    # V2.0 core remains unchanged
    v2_gates_enabled: bool = True

    # Simulation gate (NEW - mandatory)
    simulation_required: bool = True
    simulation_min_accuracy: float = 0.90  # 90% during V2.5, 95% for V3.0

    # Graph signals as BOOST only (not primary)
    graph_signals_enabled: bool = True
    graph_is_primary_signal: bool = False  # KEY: Graph BOOSTS confidence, doesn't
CREATE signals

    # Allocation split
    v2_allocation_pct: float = 0.80  # 80% to proven V2.0 signals
    graph_boost_allocation_pct: float = 0.20  # 20% extra for graph-confirmed signals


class V25TradingEngine:
    """
    V2.5 Trading Engine:
    - V2.0 signals are PRIMARY
    - Graph analysis BOOSTS confidence of existing V2.0 signals
    - Simulation VETOS any signal that fails
    """

    def __init__(self, config: V25Config, v2_engine, simulator, graph_db):
        self.config = config
        self.v2 = v2_engine
        self.simulator = simulator
        self.graph = graph_db

    async def evaluate_signal(self, signal: WhaleSignal) -> TradeDecision:
        """
        V2.5 Signal Evaluation Pipeline:
        1. V2.0 gates (must pass)
        2. Simulation (must pass)
        3. Graph boost (optional confidence increase)
        """

        # STEP 1: V2.0 Gates (mandatory, unchanged)
        v2_result = await self.v2.evaluate(signal)

        if not v2_result.should_trade:
            return TradeDecision(
                action="REJECT",
                reason=f"V2.0 gate failed: {v2_result.rejection_reason}"
            )

        # STEP 2: Simulation Gate (mandatory in V2.5+)
        sim_result = await self.simulator.full_honeypot_check(signal.token)
        if sim_result.is_honeypot:
            return TradeDecision(
                action="REJECT",
                reason=f"Simulation failed: {sim_result.reason}"
            )

        if sim_result.effective_tax_pct > 10:
            return TradeDecision(
                action="REJECT",
                reason=f"Tax too high: {sim_result.effective_tax_pct:.1f}%"
            )

        # STEP 3: Graph Confidence Boost (optional, additive)
        base_confidence = v2_result.confidence  # From V2.0
        graph_boost = 0.0

        if self.config.graph_signals_enabled:
            insider_signal = self.graph.check_insider_signal(signal.wallet)

            if insider_signal:
                if insider_signal.signal_strength == "SCREAMING_BUY":
                    graph_boost = 0.25  # +25% confidence
                elif insider_signal.signal_strength == "STRONG_BUY":
                    graph_boost = 0.15  # +15% confidence
                else:
                    graph_boost = 0.05  # +5% confidence

        final_confidence = min(base_confidence + graph_boost, 1.0)

        # STEP 4: Position Sizing based on confidence
        position_size = self._calculate_position_size(
            base_confidence=base_confidence,
            graph_boost=graph_boost,
            max_position=v2_result.suggested_position_usd
        )

        return TradeDecision(
            action="EXECUTE",
            confidence=final_confidence,
            position_usd=position_size,
            graph_boosted=graph_boost > 0,
            components_used=["V2.0", "Simulator", "Graph"] if graph_boost > 0 else ["V
2.0", "Simulator"]
        )

    def _calculate_position_size(
        self,
        base_confidence: float,
        graph_boost: float,
        max_position: float
    ) -> float:
        """
        Position sizing:
        - Base: V2.0 suggested position * base_confidence
        - Boost: +20% if graph-confirmed
        """
        base_size = max_position * base_confidence

        if graph_boost > 0:
            # Graph-confirmed signals get 20% larger position
            boost_multiplier = 1 + self.config.graph_boost_allocation_pct
            return base_size * boost_multiplier
        return base_size
```

SQLite First, Neo4j Later

Critical insight: At 100-500 wallets, SQLite handles relationship queries fine. Neo4j adds operational complexity you don’t need yet.

```python
"""
SQLite Graph Implementation for V2.5
Migrate to Neo4j ONLY when exceeding 5,000 tracked wallets
"""

import sqlite3
from datetime import datetime, timedelta
from typing import List, Optional, Tuple
from dataclasses import dataclass

@dataclass
class MotherWallet:
    address: str
    funded_winner_count: int
    child_wallets: List[str]
    last_win_date: Optional[datetime] = None
    confidence: float = 1.0


class SQLiteGraphDB:
    """
    SQLite-based graph for wallet provenance.
    Handles up to ~5,000 wallets efficiently.
    """

    def __init__(self, db_path: str = "wallet_graph.db"):
        self.conn = sqlite3.connect(db_path)
        self._init_schema()

    def _init_schema(self):
        """Initialize database schema"""
        cursor = self.conn.cursor()

        # Wallets table
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS wallets (
                address TEXT PRIMARY KEY,
                tier TEXT DEFAULT 'UNKNOWN',
                first_seen TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                total_pnl REAL DEFAULT 0,
                win_count INTEGER DEFAULT 0,
                loss_count INTEGER DEFAULT 0,
                last_win_date TIMESTAMP,
                confidence REAL DEFAULT 1.0,
                is_cex_funded BOOLEAN DEFAULT FALSE
            )
        """)

        # Funding relationships
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS funding (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                funder_address TEXT,
                funded_address TEXT,
                amount REAL,
                timestamp TIMESTAMP,
                tx_hash TEXT,
                edge_confidence REAL DEFAULT 1.0,
                FOREIGN KEY (funder_address) REFERENCES wallets(address),
                FOREIGN KEY (funded_address) REFERENCES wallets(address)
            )
        """)
        # Token buys
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS buys (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                wallet_address TEXT,
                token_address TEXT,
                amount REAL,
                price REAL,
                timestamp TIMESTAMP,
                pnl REAL DEFAULT NULL,
                FOREIGN KEY (wallet_address) REFERENCES wallets(address)
            )
        """)

        # Indexes for performance
        cursor.execute("CREATE INDEX IF NOT EXISTS idx_funding_funder ON fund-
ing(funder_address)")
        cursor.execute("CREATE INDEX IF NOT EXISTS idx_funding_funded ON fund-
ing(funded_address)")
        cursor.execute("CREATE INDEX IF NOT EXISTS idx_buys_wallet ON
buys(wallet_address)")
        cursor.execute("CREATE INDEX IF NOT EXISTS idx_buys_token ON
buys(token_address)")

        self.conn.commit()

    # ─────────────────────────────────────────────────────────────────
    # THE "GOD VIEW" QUERY - Find Mother Wallets
    # ─────────────────────────────────────────────────────────────────

    def find_mother_wallets(self, min_funded_winners: int = 3) -> List[MotherWallet]:
        """
        The "God View" Query:
        Find wallets that funded multiple addresses who then bought winning tokens.

        This is the SQLite equivalent of the Neo4j Cypher query.
        """
        cursor = self.conn.cursor()

        query = """
            SELECT
                f.funder_address as mother,
                COUNT(DISTINCT f.funded_address) as funded_count,
                GROUP_CONCAT(DISTINCT f.funded_address) as children,
                MAX(w.last_win_date) as last_win,
                AVG(w.confidence) as avg_confidence
            FROM funding f
            JOIN wallets w ON f.funded_address = w.address
            WHERE w.win_count > 0
              AND w.is_cex_funded = FALSE
            GROUP BY f.funder_address
            HAVING funded_count >= ?
            ORDER BY funded_count DESC, avg_confidence DESC
        """

        cursor.execute(query, (min_funded_winners,))

        mothers = []
        for row in cursor.fetchall():
            mothers.append(MotherWallet(
                address=row[0],
                funded_winner_count=row[1],
                child_wallets=row[2].split(',') if row[2] else [],
                last_win_date=datetime.fromisoformat(row[3]) if row[3] else None,
                confidence=row[4] or 1.0
            ))

        return mothers

    def check_insider_signal(self, wallet_address: str) -> Optional[dict]:
        """
        Check if wallet is connected to known insider cluster.
        Returns signal info if yes, None if no connection.
        """
        cursor = self.conn.cursor()

        # Check direct funding (1 hop)
        query_1hop = """
            SELECT
                f.funder_address,
                w.tier,
                w.win_count,
                w.confidence
            FROM funding f
            JOIN wallets w ON f.funder_address = w.address
            WHERE f.funded_address = ?
              AND w.tier IN ('S_TIER', 'A_TIER')
              AND w.confidence > 0.5
        """

        cursor.execute(query_1hop, (wallet_address,))
        result = cursor.fetchone()

        if result:
            return {
                "wallet": wallet_address,
                "mother_wallet": result[0],
                "mother_tier": result[1],
                "mother_wins": result[2],
                "confidence": result[3],
                "signal_strength": "SCREAMING_BUY" if result[1] == "S_TIER" else "STRO
NG_BUY",
                "hops": 1
            }

        # Check 2 hops
        query_2hop = """
            SELECT
                f2.funder_address,
                w.tier,
                w.win_count,
                w.confidence
            FROM funding f1
            JOIN funding f2 ON f1.funder_address = f2.funded_address
            JOIN wallets w ON f2.funder_address = w.address
            WHERE f1.funded_address = ?
              AND w.tier IN ('S_TIER', 'A_TIER')
              AND w.confidence > 0.5
        """

        cursor.execute(query_2hop, (wallet_address,))
        result = cursor.fetchone()

        if result:
            return {
                "wallet": wallet_address,
                "mother_wallet": result[0],
                "mother_tier": result[1],
                "mother_wins": result[2],
                "confidence": result[3] * 0.8,  # Decay for 2 hops
                "signal_strength": "STRONG_BUY" if result[1] == "S_TIER" else "MODER-
ATE",
                "hops": 2
            }

        return None

    # ─────────────────────────────────────────────────────────────────
    # CONFIDENCE DECAY (30-day half-life)
    # ─────────────────────────────────────────────────────────────────

    def apply_confidence_decay(self):
        """
        Apply 30-day half-life decay to all wallets.
        No wallet stays S-Tier indefinitely. Insiders rotate.
        Run daily.
        """
        cursor = self.conn.cursor()

        # Calculate days since last win for each wallet
        # Confidence halves every 30 days without reinforcement
        query = """
            UPDATE wallets
            SET confidence = confidence * POWER(0.5,
                CAST((julianday('now') - julianday(COALESCE(last_win_date,
first_seen))) / 30.0 AS REAL)
            )
            WHERE confidence > 0.01
        """

        cursor.execute(query)

        # Downgrade tiers based on decayed confidence
        cursor.execute("""
            UPDATE wallets SET tier = 'A_TIER'
            WHERE tier = 'S_TIER' AND confidence < 0.7
        """)

        cursor.execute("""
            UPDATE wallets SET tier = 'B_TIER'
            WHERE tier = 'A_TIER' AND confidence < 0.5
        """)

        cursor.execute("""
            UPDATE wallets SET tier = 'C_TIER'
            WHERE tier = 'B_TIER' AND confidence < 0.3
        """)

        self.conn.commit()

        # Log demotions
        cursor.execute("SELECT COUNT(*) FROM wallets WHERE confidence < 0.1")
        low_confidence_count = cursor.fetchone()[0]

        return {"low_confidence_wallets": low_confidence_count}

    def reinforce_wallet(self, address: str, pnl: float):
        """
        Reinforce wallet confidence on successful trade.
        This is how wallets EARN trust slowly.
        """
        cursor = self.conn.cursor()

        if pnl > 0:
            # Win: Boost confidence (capped at 1.0), update last_win_date
            cursor.execute("""
                UPDATE wallets
                SET confidence = MIN(confidence * 1.1, 1.0),
                    last_win_date = CURRENT_TIMESTAMP,
                    win_count = win_count + 1,
                    total_pnl = total_pnl + ?
                WHERE address = ?
            """, (pnl, address))
        else:
            # Loss: Reduce confidence quickly (lost quickly principle)
            cursor.execute("""
                UPDATE wallets
                SET confidence = confidence * 0.7,
                    loss_count = loss_count + 1,
                    total_pnl = total_pnl + ?
                WHERE address = ?
            """, (pnl, address))

        self.conn.commit()

    # ─────────────────────────────────────────────────────────────────
    # EDGE DECAY (for funding relationships)
    # ─────────────────────────────────────────────────────────────────

    def apply_edge_decay(self):
        """
        Apply decay to funding relationship edges.
        Graph edges also have half-life - old connections matter less.
        """
        cursor = self.conn.cursor()

        # 60-day half-life for edges
        cursor.execute("""
            UPDATE funding
            SET edge_confidence = edge_confidence * POWER(0.5,
                CAST((julianday('now') - julianday(timestamp)) / 60.0 AS REAL)
            )
            WHERE edge_confidence > 0.01
        """)

        # Remove very old/weak edges
        cursor.execute("""
            DELETE FROM funding WHERE edge_confidence < 0.05
        """)

        self.conn.commit()

    # ─────────────────────────────────────────────────────────────────
    # CEX BLACKLIST
    # ─────────────────────────────────────────────────────────────────

    def mark_cex_funded(self, wallet_address: str, cex_source: str):
        """Mark wallet as CEX-funded (reduces signal quality)"""
        cursor = self.conn.cursor()
        cursor.execute("""
            UPDATE wallets SET is_cex_funded = TRUE WHERE address = ?
        """, (wallet_address,))
        self.conn.commit()

    def check_cex_temporal_clustering(
        self,
        wallets: List[str],
        time_window_seconds: int = 30
    ) -> bool:
        """
        Check for temporal clustering from CEX:
        5+ wallets funded by same CEX in same block with identical amounts
        indicates potential insider coordination even through CEX
        """
        cursor = self.conn.cursor()

        # This would require funding source tracking
        # For Phase 1: Just ignore CEX-funded wallets entirely

        return False  # Placeholder - implement if needed


# Migration helper for future Neo4j transition
def should_migrate_to_neo4j(graph_db: SQLiteGraphDB) -> bool:
    """Check if wallet count exceeds SQLite threshold"""
    cursor = graph_db.conn.cursor()
    cursor.execute("SELECT COUNT(*) FROM wallets")
    wallet_count = cursor.fetchone()[0]

    # Migrate to Neo4j when exceeding 5,000 wallets
    return wallet_count > 5000
```

---

3. Part A: Battle-Hardened Refinements

All V3.0 Part A refinements remain intact. Below are ADDITIONS from expert review.

A.14 Tiered Token Age Gate (Replaces Single 1-Hour Minimum)

```python
"""
Different asset classes need different minimum ages.
Meme coins: Wait longer (too risky early)
Large caps: Can act faster (more stable)
"""

TOKEN_AGE_MINIMUM = {
    "meme_coin": 3600,      # 1 hour - high volatility, bots extract early value
    "mid_cap": 1800,        # 30 minutes - moderate risk
    "large_cap": 0          # No minimum - established, stable
}

def get_token_class(token_info: dict) -> str:
    """Classify token for age gate"""
    market_cap = token_info.get("market_cap_usd", 0)
    liquidity = token_info.get("liquidity_usd", 0)

    if market_cap > 100_000_000 or liquidity > 10_000_000:
        return "large_cap"
    elif market_cap > 10_000_000 or liquidity > 1_000_000:
        return "mid_cap"
    else:
        return "meme_coin"


class TieredTokenAgeGate:
    """
    Asset-class-aware token age requirements.
    """

    def check(
        self,
        token_creation_time: datetime,
        token_info: dict
    ) -> Tuple[bool, str]:
        """
        Check if token is old enough for our latency class.
        """
        token_class = get_token_class(token_info)
        min_age = TOKEN_AGE_MINIMUM[token_class]

        token_age = (datetime.utcnow() - token_creation_time).total_seconds()

        if token_age < min_age:
            return False, (
                f"BLOCK: {token_class} token age {token_age/60:.1f}min < "
                f"required {min_age/60:.0f}min. Premium bots already extracted value."
            )

        return True, f"OK: {token_class} age requirement met ({token_age/60:.1f}min)"
```

A.15 Backtest Framework (Historical Replay)

```python
"""
MANDATORY before any live trading:
Replay past whale signals through full pipeline.
Compare "would have traded" vs actual outcomes.
"""

import pandas as pd
from datetime import datetime, timedelta
from dataclasses import dataclass
from typing import List
import logging


@dataclass
class BacktestSignal:
    """Historical whale signal for replay"""
    timestamp: datetime
    wallet: str
    token: str
    action: str  # BUY/SELL
    amount_usd: float
    price_at_signal: float
    price_1h_later: float
    price_24h_later: float
    actual_outcome: str  # WIN/LOSS/NEUTRAL


@dataclass
class BacktestResult:
    """Results of backtesting a strategy"""
    total_signals: int
    would_have_traded: int
    simulated_wins: int
    simulated_losses: int
    simulated_win_rate: float
    simulated_pnl: float
    actual_win_rate: float  # What actually happened
    false_positive_rate: float  # Traded but would have lost
    false_negative_rate: float  # Didn't trade but would have won
    edge_vs_actual: float  # Our strategy vs following blindly


class BacktestFramework:
    """
    Historical simulation mode.
    Load past signals, run through current strategy, compare outcomes.
    """

    def __init__(self, strategy_engine, historical_data_path: str):
        self.strategy = strategy_engine
        self.data_path = historical_data_path
        self.signals: List[BacktestSignal] = []

    def load_historical_signals(self, start_date: datetime, end_date: datetime):
        """Load historical whale signals from database/CSV"""
        # Load from your Phase 0 logging data
        df = pd.read_csv(self.data_path)

        self.signals = []
        for _, row in df.iterrows():
            signal_time = datetime.fromisoformat(row['timestamp'])
            if start_date <= signal_time <= end_date:
                self.signals.append(BacktestSignal(
                    timestamp=signal_time,
                    wallet=row['wallet'],
                    token=row['token'],
                    action=row['action'],
                    amount_usd=row['amount_usd'],
                    price_at_signal=row['price'],
                    price_1h_later=row['price_1h'],
                    price_24h_later=row['price_24h'],
                    actual_outcome=row['outcome']
                ))

        logging.info(f"Loaded {len(self.signals)} historical signals for backtest")

    async def run_backtest(self) -> BacktestResult:
        """
        Run full strategy backtest.
        For each historical signal, ask: "Would we have traded?"
        Then compare to actual outcome.
        """
        results = {
            "total": 0,
            "would_trade": 0,
            "would_win": 0,
            "would_lose": 0,
            "actual_wins": 0,
            "false_positives": 0,  # We traded, it lost
            "false_negatives": 0,  # We didn't trade, it won
            "simulated_pnl": 0.0
        }

        for signal in self.signals:
            results["total"] += 1

            # Ask strategy: Would you trade this?
            # (Replay with historical context - no future data!)
            decision = await self.strategy.evaluate_signal_historical(
                signal=signal,
                context_timestamp=signal.timestamp
            )

            would_trade = decision.action == "EXECUTE"
            actually_won = signal.actual_outcome == "WIN"

            if would_trade:
                results["would_trade"] += 1

                # Calculate simulated PnL
                if actually_won:
                    results["would_win"] += 1
                    # Assume 50% gain on winners (conservative)
                    simulated_gain = decision.position_usd * 0.50
                    results["simulated_pnl"] += simulated_gain
                else:
                    results["would_lose"] += 1
                    results["false_positives"] += 1
                    # Assume 30% loss on losers (with stops)
                    simulated_loss = decision.position_usd * -0.30
                    results["simulated_pnl"] += simulated_loss
            else:
                # We wouldn't have traded
                if actually_won:
                    results["false_negatives"] += 1

            if actually_won:
                results["actual_wins"] += 1

        # Calculate metrics
        would_trade = max(results["would_trade"], 1)  # Avoid div by zero
        total = max(results["total"], 1)

        return BacktestResult(
            total_signals=results["total"],
            would_have_traded=results["would_trade"],
            simulated_wins=results["would_win"],
            simulated_losses=results["would_lose"],
            simulated_win_rate=results["would_win"] / would_trade,
            simulated_pnl=results["simulated_pnl"],
            actual_win_rate=results["actual_wins"] / total,
            false_positive_rate=results["false_positives"] / would_trade,
            false_negative_rate=results["false_negatives"] / (total - would_trade) if
(total - would_trade) > 0 else 0,
            edge_vs_actual=(results["would_win"] / would_trade) - (results["actu-
al_wins"] / total)
        )

    def validate_for_live(self, result: BacktestResult) -> Tuple[bool, str]:
        """
        Check if backtest results meet live trading requirements.
        """
        issues = []

        # Minimum requirements for live trading
        if result.simulated_win_rate < 0.55:
            issues.append(f"Win rate {result.simulated_win_rate:.1%} < required 55%")

        if result.simulated_pnl < 0:
            issues.append(f"Simulated PnL negative: ${result.simulated_pnl:.2f}")

        if result.false_positive_rate > 0.30:
            issues.append(f"False positive rate {result.false_positive_rate:.
1%} > 30% threshold")

        if result.would_have_traded < 30:
            issues.append(f"Sample size too small: {result.would_have_traded} trades")

        if issues:
            return False, "BACKTEST FAILED:\n" + "\n".join(f"  - {i}" for i in issues)

        return True, f"BACKTEST PASSED: {result.simulated_win_rate:.1%} win rate, ${re
sult.simulated_pnl:.2f} simulated PnL"


# Backtest validation gate
async def pre_live_backtest_gate(
    strategy_engine,
    historical_data_path: str,
    lookback_days: int = 90
) -> Tuple[bool, str]:
    """
    MANDATORY gate before going live.
    Must pass backtest on last 90 days of data.
    """
    end_date = datetime.utcnow()
    start_date = end_date - timedelta(days=lookback_days)
    backtester = BacktestFramework(strategy_engine, historical_data_path)
    backtester.load_historical_signals(start_date, end_date)

    result = await backtester.run_backtest()

    logging.info(f"""
    BACKTEST RESULTS ({lookback_days} days):
    ─────────────────────────────────────
    Total signals:     {result.total_signals}
    Would have traded: {result.would_have_traded}
    Simulated wins:    {result.simulated_wins}
    Simulated losses:  {result.simulated_losses}
    Win rate:          {result.simulated_win_rate:.1%}
    Simulated PnL:     ${result.simulated_pnl:.2f}
    False positive:    {result.false_positive_rate:.1%}
    Edge vs random:    {result.edge_vs_actual:+.1%}
    """)

    return backtester.validate_for_live(result)
```

---

4. Part B: Hydra Architecture (Fixed)

Core Hydra architecture from V3.0 with critical fixes applied.

B.1 HEAD 1: Sybil Hunter (Fixed)

Fix #1: CEX Mixer Problem

```python
"""
CEX Hot Wallet Blacklist.
If wallet funded by CEX → dead end UNLESS temporal clustering detected.
For Phase 1: Just ignore CEX-funded wallets entirely.
"""

CEX_HOT_WALLETS = {
    # Solana CEX Hot Wallets
    "solana": {
        # Binance
        "5tzFkiKscXHK5ZXCGbXZxdw7gTjjD1mBwuoFbhUvuAi9": "Binance",
        "9WzDXwBbmkg8ZTbNMqUxvQRAyrZzDsGYdLVL9zYtAWWM": "Binance",
        # Coinbase
        "GJRs4FwHtemZ5ZE9x3FNvJ8TMwitKTh21yxdRPqn7npE": "Coinbase",
        "H8sMJSCQxfKiFTCfDR3DUMLPwcRbM61LGFJ8N4dK3WjS": "Coinbase",
        # Kraken
        "JA98mShDSf4qqxVpVvGTwRZ8NrqbUhpUhgHy8gF3K1bJ": "Kraken",
        # FTX (defunct but still tracked)
        "FTXkNgBQ3rvC3RZRxvdqFvGZmwn7nQ8AQZJ3EUbsxP9": "FTX",
        # OKX
        "ExXkN3vR4xvBp6RQxwZ3kM9e4qyJdNb9cTvVnfJ1mH7": "OKX",
        # Bybit
        "BybitHotWallet1111111111111111111111111111": "Bybit",
    },
    # Add EVM CEX wallets here
    "ethereum": {
        "0x28C6c06298d514Db089934071355E5743bf21d60": "Binance14",
        "0x21a31Ee1afC51d94C2eFcCAa2092aD1028285549": "Binance15",
        "0xdfd5293d8e347dfe59e90efd55b2956a1343963d": "Binance16",
        "0x503828976D22510aad0201ac7EC88293211D23Da": "Coinbase1",
        "0xddfAbCdc4D8FfC6d5beaf154f18B778f892A0740": "Coinbase2",
        "0x267be1c1d684f78cb4f6a176c4911b741e4ffdc0": "Kraken4",
    }
}


def is_cex_funded(funder_address: str, chain: str = "solana") -> Tuple[bool, str]:
    """
    Check if address is a known CEX hot wallet.
    """
    cex_wallets = CEX_HOT_WALLETS.get(chain, {})

    if funder_address in cex_wallets:
        return True, cex_wallets[funder_address]

    return False, ""


class SybilHunterWithCEXFilter:
    """
    Enhanced Sybil Hunter with CEX filtering.
    """

    def __init__(self, graph_db: SQLiteGraphDB, chain: str = "solana"):
        self.graph = graph_db
        self.chain = chain

    async def harvest_winner_with_cex_filter(
        self,
        token_address: str,
        min_gain: float = 10.0
    ):
        """
        Harvest winner but filter out CEX-funded wallets.
        """
        early_buyers = await self._fetch_early_buyers(token_address, limit=50)

        for buyer in early_buyers:
            # Get funding source
            funding_source = await self._get_primary_funder(buyer.address)

            if funding_source:
                is_cex, cex_name = is_cex_funded(funding_source, self.chain)

                if is_cex:
                    logging.debug(
                        f"Skipping {buyer.address[:8]}... - "
                        f"funded by CEX ({cex_name})"
                    )
                    # Mark as CEX-funded but don't use for signals
                    self.graph.mark_cex_funded(buyer.address, cex_name)
                    continue

            # Not CEX-funded - proceed with normal harvest
            self.graph.add_wallet(buyer.address)
            self.graph.add_buy(buyer.address, token_address, buyer.amount, buyer.price
, buyer.timestamp)

            # Trace non-CEX funding chain
            if funding_source:
                is_cex, _ = is_cex_funded(funding_source, self.chain)
                if not is_cex:
                    self.graph.add_funding_relationship(
                        funding_source, buyer.address, buyer.funding_amount, buyer.fun
ding_timestamp
                    )
```

Fix #2: Graph Poisoning Protection

```
"""
Graph Poisoning Protection:
- Mother wallets only gain trust AFTER multiple cycles (not first win)
- Weight NET REALIZED PnL, not hit rate
- Decay graph edges aggressively
- Principle: Trust earned slowly, lost quickly
"""

class PoisonResistantGraphDB(SQLiteGraphDB):
    """
    Enhanced graph with anti-poisoning measures.
    """

    # Trust thresholds
    MIN_WINS_FOR_TRUST = 3  # Need 3+ wins before considered trustworthy
    MIN_CYCLES_FOR_MOTHER = 2  # Mother wallet needs 2+ funding cycles with winners
    NET_PNL_WEIGHT = 0.7  # 70% weight on net PnL, 30% on hit rate

    def calculate_wallet_trust(self, address: str) -> float:
        """
        Calculate trust score with anti-poison measures.
        Trust = (net_pnl_normalized * 0.7) + (win_rate * 0.3)
        Only applies after MIN_WINS_FOR_TRUST
        """
        cursor = self.conn.cursor()
        cursor.execute("""
            SELECT win_count, loss_count, total_pnl, confidence
            FROM wallets WHERE address = ?
        """, (address,))

        row = cursor.fetchone()
        if not row:
            return 0.0

        wins, losses, total_pnl, base_confidence = row
        total_trades = wins + losses

        # Not enough history
        if wins < self.MIN_WINS_FOR_TRUST:
            return 0.0  # No trust yet

        # Calculate components
        win_rate = wins / total_trades if total_trades > 0 else 0

        # Normalize PnL (cap at 10x for normalization)
        # Positive PnL > 0 is good, negative is bad
        pnl_normalized = min(max(total_pnl / 1000, -1), 1)  # -1 to 1 range

        # Weighted trust score
        trust = (pnl_normalized * self.NET_PNL_WEIGHT) + (win_rate * (1 - self.NET_PNL
_WEIGHT))

        # Apply base confidence decay
        trust *= base_confidence

        return max(trust, 0)

    def promote_to_mother(self, address: str) -> Tuple[bool, str]:
        """
        Promote wallet to Mother status.
        Requires multiple funding cycles with winning children.
        """
        cursor = self.conn.cursor()

        # Count distinct funding cycles with winners
        cursor.execute("""
            SELECT COUNT(DISTINCT strftime('%Y-%m', f.timestamp))
            FROM funding f
            JOIN wallets w ON f.funded_address = w.address
            WHERE f.funder_address = ?
              AND w.win_count >= 1
        """, (address,))

        cycles = cursor.fetchone()[0]

        if cycles < self.MIN_CYCLES_FOR_MOTHER:
            return False, f"Only {cycles} winning cycles, need {self.MIN_CYCLES_FOR_MO
THER}"

        # Check net PnL of children
        cursor.execute("""
            SELECT SUM(w.total_pnl)
            FROM funding f
            JOIN wallets w ON f.funded_address = w.address
            WHERE f.funder_address = ?
        """, (address,))

        children_net_pnl = cursor.fetchone()[0] or 0

        if children_net_pnl <= 0:
            return False, f"Children net PnL is ${children_net_pnl:.2f}, must be pos-
itive"

        # Promote
        cursor.execute("""
            UPDATE wallets SET tier = 'MOTHER' WHERE address = ?
        """, (address,))
        self.conn.commit()

        return True, f"Promoted to MOTHER: {cycles} cycles, ${children_net_pnl:.2f}
children PnL"
```

B.2 HEAD 2: Simulator (Partial Hydra Rule)

Critical Rule: 95% Accuracy Required

```
"""
PARTIAL HYDRA WARNING (CRITICAL):
NEVER allow HEAD 3 (Assassin) to execute unless HEAD 2 (Simulator)
has blocked ≥95% of eventual losers in shadow mode.

If simulation accuracy <95%, Assassin MUST stay disabled.
"""

@dataclass
class SimulatorAccuracyTracker:
    """Track simulator's accuracy in blocking losers"""

    # Counters
    total_signals: int = 0
    simulator_blocked: int = 0
    simulator_passed: int = 0

    # Outcome tracking (after the fact)
    passed_that_won: int = 0
    passed_that_lost: int = 0
    blocked_that_would_have_won: int = 0
    blocked_that_would_have_lost: int = 0

    @property
    def blocker_accuracy(self) -> float:
        """What % of losers did we correctly block?"""
        total_losers = self.passed_that_lost + self.blocked_that_would_have_lost
        if total_losers == 0:
            return 0.0
        return self.blocked_that_would_have_lost / total_losers

    @property
    def assassin_ready(self) -> bool:
        """Is Assassin allowed to go live?"""
        # Minimum sample size
        if self.total_signals < 50:
            return False

        # 95% accuracy required
        return self.blocker_accuracy >= 0.95


class SimulatorGateWithTracking:
    """
    Simulator gate that tracks accuracy for Assassin readiness.
    """

    def __init__(self, simulator: TransactionSimulator):
        self.simulator = simulator
        self.tracker = SimulatorAccuracyTracker()
        self.pending_outcomes: Dict[str, SimulationDecision] = {}

    async def evaluate(self, signal: WhaleSignal) -> SimulationDecision:
        """
        Evaluate signal through simulation.
        Track decision for later outcome correlation.
        """
        self.tracker.total_signals += 1

        # Run simulation
        result = await self.simulator.full_honeypot_check(signal.token)
        decision = SimulationDecision(
            signal_id=signal.id,
            token=signal.token,
            timestamp=datetime.utcnow(),
            passed=not result.is_honeypot and result.effective_tax_pct <= 10,
            reason=result.reason,
            tax_pct=result.effective_tax_pct
        )

        if decision.passed:
            self.tracker.simulator_passed += 1
        else:
            self.tracker.simulator_blocked += 1

        # Store for outcome tracking
        self.pending_outcomes[signal.id] = decision

        return decision

    def record_outcome(self, signal_id: str, pnl: float):
        """
        Record actual outcome for accuracy tracking.
        Call this after trade completes (or would have completed).
        """
        if signal_id not in self.pending_outcomes:
            return

        decision = self.pending_outcomes.pop(signal_id)
        won = pnl > 0

        if decision.passed:
            if won:
                self.tracker.passed_that_won += 1
            else:
                self.tracker.passed_that_lost += 1
        else:
            if won:
                self.tracker.blocked_that_would_have_won += 1
            else:
                self.tracker.blocked_that_would_have_lost += 1

        # Log accuracy periodically
        if self.tracker.total_signals % 10 == 0:
            logging.info(
                f"Simulator accuracy: {self.tracker.blocker_accuracy:.1%} "
                f"(Assassin ready: {self.tracker.assassin_ready})"
            )

    def get_assassin_status(self) -> Tuple[bool, str]:
        """Check if Assassin can be enabled"""
        if self.tracker.total_signals < 50:
            return False, f"Need 50+ signals, have {self.tracker.total_signals}"

        accuracy = self.tracker.blocker_accuracy
        if accuracy < 0.95:
            return False, f"Accuracy {accuracy:.1%} < required 95%"

        return True, f"Assassin READY: {accuracy:.1%} accuracy over {self.tracker.tota
l_signals} signals"


@dataclass
class SimulationDecision:
    signal_id: str
    token: str
    timestamp: datetime
    passed: bool
    reason: str
    tax_pct: float
```

B.3 HEAD 3: Assassin (Rust Sidecar Pattern)

Pragmatic Rust Implementation

```python
"""
Rust Sidecar Pattern:
- Keep "Brain" (Strategy, Graph, Simulation) in Python - fast enough for decisions
(50ms)
- Use Rust ONLY for the Trigger/Executor
- Don't write Jito client from scratch
- Fork existing open-source Jito backrun bot
- Strip logic, turn into tiny local server accepting (Token, Amount) from Python
- This cuts Rust learning from 6 months to 2-4 weeks
"""

# assassin_sidecar.py - Python side

import aiohttp
import asyncio
from dataclasses import dataclass
from typing import Optional


@dataclass
class ExecutionRequest:
    """Request to Rust sidecar"""
    token_address: str
    amount_sol: float
    direction: str  # "BUY" or "SELL"
    max_slippage_bps: int = 300  # 3%
    tip_lamports: int = 10000  # 0.00001 SOL tip


@dataclass
class ExecutionResult:
    """Response from Rust sidecar"""
    success: bool
    signature: Optional[str] = None
    slot: Optional[int] = None
    error: Optional[str] = None
    cost_lamports: int = 0  # Should be 0 if bundle dropped


class RustAssassinSidecar:
    """
    Python interface to Rust execution sidecar.
    The Rust service is a minimal wrapper around existing Jito backrun bots.
    """

    def __init__(self, sidecar_url: str = "http://127.0.0.1:3030"):
        self.url = sidecar_url
        self.enabled = False
        self.session: Optional[aiohttp.ClientSession] = None

    async def start(self):
        """Initialize connection to sidecar"""
        self.session = aiohttp.ClientSession()

        # Health check
        try:
            async with self.session.get(f"{self.url}/health") as resp:
                if resp.status == 200:
                    self.enabled = True
                    logging.info("Rust Assassin sidecar connected")
                else:
                    logging.warning("Rust sidecar unhealthy, falling back to standard
execution")
        except Exception as e:
            logging.warning(f"Rust sidecar unavailable: {e}. Using standard execu-
tion.")

    async def execute(self, request: ExecutionRequest) -> ExecutionResult:
        """
        Send execution request to Rust sidecar.
        Returns immediately if sidecar unavailable.
        """
        if not self.enabled or not self.session:
            return ExecutionResult(
                success=False,
                error="Sidecar not available"
            )

        payload = {
            "token": request.token_address,
            "amount": request.amount_sol,
            "direction": request.direction,
            "max_slippage_bps": request.max_slippage_bps,
            "tip_lamports": request.tip_lamports
        }

        try:
            async with self.session.post(
                f"{self.url}/execute",
                json=payload,
                timeout=aiohttp.ClientTimeout(total=5)
            ) as resp:
                data = await resp.json()

                return ExecutionResult(
                    success=data.get("success", False),
                    signature=data.get("signature"),
                    slot=data.get("slot"),
                    error=data.get("error"),
                    cost_lamports=data.get("cost", 0)
                )

        except asyncio.TimeoutError:
            return ExecutionResult(success=False, error="Sidecar timeout")
        except Exception as e:
            return ExecutionResult(success=False, error=str(e))

    async def stop(self):
        """Cleanup"""
        if self.session:
            await self.session.close()
```

Rust Sidecar Implementation (Minimal):

```rust
// assassin_sidecar/src/main.rs
// Minimal Rust sidecar - wraps existing Jito client
// Learning time: 2-4 weeks (not 6 months)

use axum::{routing::post, Router, Json};
use serde::{Deserialize, Serialize};
use std::sync::Arc;
use tokio::sync::Mutex;

// Fork from: https://github.com/jito-labs/jito-solana
// Or: https://github.com/Ellipsis-Labs/phoenix-v1
use jito_client::JitoClient;

#[derive(Deserialize)]
struct ExecuteRequest {
    token: String,
    amount: f64,
    direction: String,
    max_slippage_bps: u16,
    tip_lamports: u64,
}

#[derive(Serialize)]
struct ExecuteResponse {
    success: bool,
    signature: Option<String>,
    slot: Option<u64>,
    error: Option<String>,
    cost: u64,
}

struct AppState {
    jito: JitoClient,
}

#[tokio::main]
async fn main() {
    // Initialize Jito client (from existing library)
    let jito = JitoClient::new(
        std::env::var("JITO_BLOCK_ENGINE").unwrap_or("https://mainnet.block-en-
gine.jito.wtf".into()),
        std::env::var("WALLET_KEYPAIR_PATH").unwrap_or("~/.config/solana/id.json".into
()),
    ).await.expect("Failed to init Jito client");

    let state = Arc::new(Mutex::new(AppState { jito }));

    let app = Router::new()
        .route("/health", axum::routing::get(|| async { "OK" }))
        .route("/execute", post(execute_trade))
        .with_state(state);

    println!("Assassin sidecar listening on 127.0.0.1:3030");
    axum::Server::bind(&"127.0.0.1:3030".parse().unwrap())
        .serve(app.into_make_service())
        .await
        .unwrap();
}

async fn execute_trade(
    state: axum::extract::State<Arc<Mutex<AppState>>>,
    Json(req): Json<ExecuteRequest>,
) -> Json<ExecuteResponse> {
    let state = state.lock().await;

    // Build swap transaction using Jupiter
    let swap_tx = match build_swap(&req).await {
        Ok(tx) => tx,
        Err(e) => return Json(ExecuteResponse {
            success: false,
            signature: None,
            slot: None,
            error: Some(e.to_string()),
            cost: 0,
        }),
    };

    // Send via Jito bundle
    match state.jito.send_bundle(swap_tx, req.tip_lamports).await {
        Ok(result) => Json(ExecuteResponse {
            success: true,
            signature: Some(result.signature),
            slot: Some(result.slot),
            error: None,
            cost: if result.landed { req.tip_lamports } else { 0 },
        }),
        Err(e) => Json(ExecuteResponse {
            success: false,
            signature: None,
            slot: None,
            error: Some(e.to_string()),
            cost: 0, // Bundle dropped = $0 cost
        }),
    }
}

async fn build_swap(req: &ExecuteRequest) -> Result<so-
lana_sdk::transaction::Transaction, Box<dyn std::error::Error>> {
    // Use Jupiter API to build optimal swap route
    // This is a simplified version - real impl uses jupiter-core

    let jupiter_quote = reqwest::get(format!(
        "https://quote-api.jup.ag/v6/quote?input-
Mint=So11111111111111111111111111111111111111112&outputMint={}&amount={}&slippage-
Bps={}",
        req.token,
        (req.amount * 1e9) as u64,  // Convert to lamports
        req.max_slippage_bps
    )).await?.json::<serde_json::Value>().await?;

    // Build transaction from quote
    // ... (implementation details)

    todo!("Implement Jupiter swap transaction building")
}
```

B.4 Geyser Cost Trap Fix

Use Helius Webhooks Instead of Raw Mempool

```python
"""
Geyser Cost Trap Fix:
Replace "stream whole mempool" ($200+/month) with:
- Helius Enhanced Webhooks (Free/Pro tier)
- OR standard accountSubscribe WebSockets
- Subscribe ONLY to the 50 specific Mother Wallets in Black Book
- Reduces data load by 99.9%, costs $0
- Still fast enough to back-run (same block)
"""

import asyncio
import aiohttp
import websockets
from typing import List, Callable
import json


class HeliusWebhookManager:
    """
    Use Helius webhooks to monitor specific wallets.
    Free tier: 100 webhooks, 10 requests/second
    Pro tier: Unlimited, $99/month (only if needed)

    This replaces expensive mempool streaming.
    """

    def __init__(self, api_key: str):
        self.api_key = api_key
        self.base_url = "https://api.helius.xyz/v0"
        self.webhooks: dict[str, str] = {}  # wallet -> webhook_id

    async def setup_wallet_monitors(
        self,
        wallets: List[str],
        callback_url: str
    ):
        """
        Create webhooks for each wallet in the Black Book.
        Helius will POST to callback_url when these wallets transact.
        """
        async with aiohttp.ClientSession() as session:
            for wallet in wallets[:50]:  # Free tier limit
                payload = {
                    "webhookURL": callback_url,
                    "transactionTypes": ["Any"],  # Or specific: ["SWAP", "TRANSFER"]
                    "accountAddresses": [wallet],
                    "webhookType": "enhanced"  # Rich transaction data
                }

                async with session.post(
                    f"{self.base_url}/webhooks?api-key={self.api_key}",
                    json=payload
                ) as resp:
                    if resp.status == 200:
                        data = await resp.json()
                        self.webhooks[wallet] = data["webhookID"]
                        print(f"✓ Monitoring {wallet[:8]}...")
                    else:
                        error = await resp.text()
                        print(f"✗ Failed to monitor {wallet[:8]}: {error}")

    async def remove_wallet(self, wallet: str):
        """Remove webhook when wallet is no longer trusted"""
        if wallet not in self.webhooks:
            return

        webhook_id = self.webhooks[wallet]

        async with aiohttp.ClientSession() as session:
            await session.delete(
                f"{self.base_url}/webhooks/{webhook_id}?api-key={self.api_key}"
            )

        del self.webhooks[wallet]


class WebSocketWalletMonitor:
    """
    Alternative: Standard Solana WebSocket accountSubscribe.
    Completely free, works with any RPC.
    """

    def __init__(self, rpc_ws_url: str = "wss://api.mainnet-beta.solana.com"):
        self.rpc_url = rpc_ws_url
        self.subscriptions: dict[str, int] = {}  # wallet -> subscription_id
        self.callback: Callable = None

    async def start_monitoring(
        self,
        wallets: List[str],
        callback: Callable
    ):
        """
        Subscribe to account changes for each wallet.
        """
        self.callback = callback

        async with websockets.connect(self.rpc_url) as ws:
            # Subscribe to each wallet
            for i, wallet in enumerate(wallets[:50]):  # Limit subscriptions
                subscribe_msg = {
                    "jsonrpc": "2.0",
                    "id": i,
                    "method": "accountSubscribe",
                    "params": [
                        wallet,
                        {"encoding": "jsonParsed", "commitment": "confirmed"}
                    ]
                }
                await ws.send(json.dumps(subscribe_msg))

                # Get subscription ID
                response = await ws.recv()
                data = json.loads(response)
                if "result" in data:
                    self.subscriptions[wallet] = data["result"]

            print(f"Monitoring {len(self.subscriptions)} wallets via WebSocket")

            # Listen for updates
            async for message in ws:
                data = json.loads(message)
                if "method" in data and data["method"] == "accountNotification":
                    await self._handle_notification(data)
    async def _handle_notification(self, notification: dict):
        """Handle account change notification"""
        # Extract wallet address and changes
        params = notification.get("params", {})
        result = params.get("result", {})

        # Call user callback
        if self.callback:
            await self.callback(result)


# Cost comparison
INFRASTRUCTURE_COSTS = {
    "geyser_streaming": {
        "provider": "Various",
        "cost_monthly": 200,  # $200-500/month
        "data_volume": "All mempool transactions",
        "relevance": "1% useful"
    },
    "helius_webhooks_free": {
        "provider": "Helius",
        "cost_monthly": 0,
        "data_volume": "50 specific wallets only",
        "relevance": "100% useful"
    },
    "helius_webhooks_pro": {
        "provider": "Helius",
        "cost_monthly": 99,
        "data_volume": "Unlimited wallets",
        "relevance": "100% useful"
    },
    "standard_websocket": {
        "provider": "Any RPC",
        "cost_monthly": 0,
        "data_volume": "Subscribed accounts only",
        "relevance": "100% useful"
    }
}
```

5. Part C: Counter-Adaptation & Safety Systems

NEW section addressing adversarial counter-measures.

C.1 Signal Entropy Injection (Anti-Pattern Detection)

```python
"""
Counter-Adaptation Defense:
Once you're visible (consistent patterns), adversaries can:
1. Detect your trades
2. Poison your signals
3. Front-run YOU

Solution: Be unpredictable. Inject randomness.
"""

import random
import asyncio
from datetime import datetime
from typing import List


class EntropyInjector:
    """
    Inject randomness to avoid being trackable.
    """

    def __init__(self, config: dict = None):
        self.config = config or {
            "delay_min_ms": 5,
            "delay_max_ms": 30,
            "skip_probability": 0.10,  # Skip 10% of valid signals randomly
            "wallet_rotation_enabled": True,
            "num_execution_wallets": 5
        }

        self.execution_wallets: List[str] = []
        self.current_wallet_index = 0

    def add_jitter_delay(self) -> float:
        """
        Add random delay before execution.
        Prevents timing pattern detection.
        Returns delay in seconds.
        """
        delay_ms = random.uniform(
            self.config["delay_min_ms"],
            self.config["delay_max_ms"]
        )
        return delay_ms / 1000

    def should_skip_signal(self) -> bool:
        """
        Probabilistically skip some valid signals.
        This prevents adversaries from correlating ALL your entries.

        Counter-intuitive but crucial:
        Missing 10% of opportunities is worth it to stay invisible.
        """
        return random.random() < self.config["skip_probability"]

    def get_execution_wallet(self) -> str:
        """
        Rotate through execution wallets.
        Each trade uses different wallet, harder to track.
        """
        if not self.execution_wallets:
            raise ValueError("No execution wallets configured")
        wallet = self.execution_wallets[self.current_wallet_index]

        # Rotate to next wallet
        self.current_wallet_index = (
            self.current_wallet_index + 1
        ) % len(self.execution_wallets)

        return wallet

    def randomize_position_size(self, base_size: float) -> float:
        """
        Slightly randomize position size.
        Prevents exact-amount pattern matching.
        """
        # ±5% randomization
        multiplier = random.uniform(0.95, 1.05)
        return base_size * multiplier


class AntiPatternExecutor:
    """
    Executor that wraps standard execution with entropy injection.
    """

    def __init__(self, executor, entropy: EntropyInjector):
        self.executor = executor
        self.entropy = entropy
        self.skipped_signals: List[dict] = []

    async def execute_with_entropy(self, trade_intent):
        """
        Execute trade with anti-pattern measures.
        """
        # Check if we should skip (unpredictability)
        if self.entropy.should_skip_signal():
            logging.info(
                f"ENTROPY SKIP: Randomly skipping valid signal for {trade_intent.token
[:8]}..."
            )
            self.skipped_signals.append({
                "token": trade_intent.token,
                "timestamp": datetime.utcnow(),
                "reason": "ENTROPY_SKIP"
            })
            return None

        # Add random delay
        delay = self.entropy.add_jitter_delay()
        await asyncio.sleep(delay)

        # Get rotated wallet
        wallet = self.entropy.get_execution_wallet()

        # Randomize position size slightly
        position = self.entropy.randomize_position_size(trade_intent.position_usd)

        # Execute
        logging.debug(
            f"Executing with entropy: wallet={wallet[:8]}, "
            f"delay={delay*1000:.1f}ms, position=${position:.2f}"
        )
        return await self.executor.execute(
            token=trade_intent.token,
            amount_usd=position,
            wallet=wallet
        )
```

C.2 Graph Kill Switch (Epistemic Failure Protection)

```python
"""
Graph Kill Switch:
When the graph-based system shows signs of failure, HALT immediately.

Triggers:
1. Sudden explosion of new mother wallets (>10 in 24h)
2. Correlated cluster emergence across unrelated tokens
3. Win rate collapse across multiple clusters simultaneously

Action: Freeze Sybil signals, revert to V2.0 only, enter 72h observation
"""

from dataclasses import dataclass
from datetime import datetime, timedelta
from typing import List, Optional
import logging


@dataclass
class KillSwitchState:
    """Current state of kill switch"""
    triggered: bool = False
    trigger_reason: str = ""
    triggered_at: Optional[datetime] = None
    observation_end: Optional[datetime] = None
    mode: str = "NORMAL"  # NORMAL, OBSERVATION, LOCKDOWN


class GraphKillSwitch:
    """
    Emergency halt system for graph-based signals.
    """

    # Thresholds
    MAX_NEW_MOTHERS_24H = 10
    WIN_RATE_COLLAPSE_THRESHOLD = 0.30  # If win rate drops below 30%
    MIN_CLUSTERS_FOR_COLLAPSE = 3  # Need 3+ clusters collapsing simultaneously
    OBSERVATION_PERIOD_HOURS = 72

    def __init__(self, graph_db, metrics_tracker):
        self.graph = graph_db
        self.metrics = metrics_tracker
        self.state = KillSwitchState()
        self.mother_discovery_history: List[datetime] = []
        self.cluster_win_rates: dict[str, List[float]] = {}

    def check_triggers(self) -> Optional[str]:
        """
        Check all kill switch triggers.
        Returns trigger reason if triggered, None otherwise.
        """
        # Already in observation mode
        if self.state.mode == "OBSERVATION":
            if datetime.utcnow() < self.state.observation_end:
                return None  # Still observing
            else:
                self._exit_observation()

        # Trigger 1: Mother wallet explosion
        trigger = self._check_mother_explosion()
        if trigger:
            return trigger
        # Trigger 2: Correlated cluster emergence
        trigger = self._check_cluster_correlation()
        if trigger:
            return trigger

        # Trigger 3: Win rate collapse
        trigger = self._check_win_rate_collapse()
        if trigger:
            return trigger

        return None

    def _check_mother_explosion(self) -> Optional[str]:
        """
        Check for sudden spike in new mother wallet discoveries.
        This indicates either:
        - Market regime change (many new winners)
        - Adversarial poisoning (fake mother wallets)
        """
        cutoff = datetime.utcnow() - timedelta(hours=24)
        recent_discoveries = [t for t in self.mother_discovery_history if t > cutoff]

        if len(recent_discoveries) > self.MAX_NEW_MOTHERS_24H:
            return (
                f"MOTHER EXPLOSION: {len(recent_discoveries)} new mother wallets in
24h "
                f"(threshold: {self.MAX_NEW_MOTHERS_24H}). Possible poisoning attack."
            )

        return None

    def _check_cluster_correlation(self) -> Optional[str]:
        """
        Check for suspicious correlation across unrelated token clusters.
        If multiple unrelated clusters emerge simultaneously, likely manipulation.
        """
        # Get clusters created in last 24h
        recent_clusters = self.graph.get_recent_clusters(hours=24)

        if len(recent_clusters) < 3:
            return None

        # Check if clusters are buying unrelated tokens
        cluster_tokens = {}
        for cluster in recent_clusters:
            tokens = self.graph.get_cluster_tokens(cluster.id)
            cluster_tokens[cluster.id] = set(tokens)

        # Count overlaps
        cluster_ids = list(cluster_tokens.keys())
        suspicious_pairs = 0

        for i, c1 in enumerate(cluster_ids):
            for c2 in cluster_ids[i+1:]:
                overlap = cluster_tokens[c1] & cluster_tokens[c2]
                if len(overlap) > 3:  # 3+ common tokens
                    suspicious_pairs += 1

        if suspicious_pairs > 2:
            return (
                f"CLUSTER CORRELATION: {suspicious_pairs} cluster pairs with "
                f"suspicious token overlap. Possible coordinated manipulation."
            )

        return None

    def _check_win_rate_collapse(self) -> Optional[str]:
        """
        Check if win rate is collapsing across multiple clusters.
        """
        collapsing_clusters = 0

        for cluster_id, rates in self.cluster_win_rates.items():
            if len(rates) < 5:
                continue

            recent_rate = sum(rates[-5:]) / 5

            if recent_rate < self.WIN_RATE_COLLAPSE_THRESHOLD:
                collapsing_clusters += 1

        if collapsing_clusters >= self.MIN_CLUSTERS_FOR_COLLAPSE:
            return (
                f"WIN RATE COLLAPSE: {collapsing_clusters} clusters below "
                f"{self.WIN_RATE_COLLAPSE_THRESHOLD*100:.0f}% win rate. "
                f"Graph signals unreliable."
            )

        return None

    def trigger(self, reason: str):
        """
        Trigger the kill switch.
        """
        logging.critical(f"🚨 GRAPH KILL SWITCH TRIGGERED: {reason}")

        self.state = KillSwitchState(
            triggered=True,
            trigger_reason=reason,
            triggered_at=datetime.utcnow(),
            observation_end=datetime.utcnow() + timedelta(hours=self.OBSERVA-
TION_PERIOD_HOURS),
            mode="OBSERVATION"
        )

        # Actions
        logging.warning("IMMEDIATE ACTIONS:")
        logging.warning("  1. Freezing all Sybil/graph-based signals")
        logging.warning("  2. Reverting to V2.0 signals only")
        logging.warning(f"  3. Entering {self.OBSERVATION_PERIOD_HOURS}h observation
mode")

    def _exit_observation(self):
        """Exit observation mode after cooling period"""
        logging.info("Exiting observation mode. Graph signals re-enabled with cau-
tion.")
        self.state.mode = "NORMAL"
        self.state.triggered = False

    def is_graph_enabled(self) -> bool:
        """Check if graph signals are allowed"""
        return self.state.mode == "NORMAL"

    def record_mother_discovery(self):
        """Record when a new mother wallet is discovered"""
        self.mother_discovery_history.append(datetime.utcnow())

        # Cleanup old entries
        cutoff = datetime.utcnow() - timedelta(hours=48)
        self.mother_discovery_history = [
            t for t in self.mother_discovery_history if t > cutoff
        ]
```

C.3 False-Positive Cost Metric

```
"""
Track and alert on false positives.
If insider false positives exceed threshold → reduce V3 allocation automatically.
"""

@dataclass
class FalsePositiveMetrics:
    """Track false positive costs"""

    # Assassin attempts
    total_assassin_attempts: int = 0
    assassin_successes: int = 0

    # Simulation pass but price reversed
    simulation_passed_count: int = 0
    simulation_passed_but_lost: int = 0

    # Opportunity cost
    missed_non_insider_trades: int = 0
    missed_pnl_estimate: float = 0.0

    @property
    def assassin_success_rate(self) -> float:
        if self.total_assassin_attempts == 0:
            return 0.0
        return self.assassin_successes / self.total_assassin_attempts

    @property
    def simulation_false_positive_rate(self) -> float:
        if self.simulation_passed_count == 0:
            return 0.0
        return self.simulation_passed_but_lost / self.simulation_passed_count


class AllocationAdjuster:
    """
    Automatically adjust V3 allocation based on performance.
    """

    # Thresholds
    FP_RATE_WARNING = 0.20  # 20% false positive rate
    FP_RATE_REDUCE = 0.30   # 30% → reduce allocation
    FP_RATE_DISABLE = 0.40  # 40% → disable V3 entirely

    def __init__(self, metrics: FalsePositiveMetrics):
        self.metrics = metrics
        self.v3_allocation_multiplier = 1.0

    def check_and_adjust(self) -> str:
        """
        Check metrics and adjust allocation.
        Returns action taken.
        """
        fp_rate = self.metrics.simulation_false_positive_rate

        if fp_rate >= self.FP_RATE_DISABLE:
            self.v3_allocation_multiplier = 0.0
            return f"V3 DISABLED: False positive rate {fp_rate:.1%} ≥ {self.FP_RATE_DI
SABLE:.0%}"

        elif fp_rate >= self.FP_RATE_REDUCE:
            # Reduce proportionally
            reduction = (fp_rate - self.FP_RATE_WARNING) / (self.FP_RATE_DISABLE - sel
f.FP_RATE_WARNING)
            self.v3_allocation_multiplier = max(0.5, 1.0 - reduction)
            return f"V3 REDUCED to {self.v3_allocation_multiplier:.0%}: FP rate {fp_ra
te:.1%}"

        elif fp_rate >= self.FP_RATE_WARNING:
            return f"V3 WARNING: False positive rate {fp_rate:.1%} approaching
threshold"

        else:
            self.v3_allocation_multiplier = 1.0
            return f"V3 HEALTHY: False positive rate {fp_rate:.1%}"

    def get_adjusted_position(self, base_position: float) -> float:
        """Apply allocation adjustment to position size"""
        return base_position * self.v3_allocation_multiplier
```

6. Part D: Day 1 Execution Plan

Concrete code to start TODAY.

D.1 Phase 0 Complete Setup

```
#!/bin/bash
# day1_setup.sh - Complete Phase 0 setup

echo "=== PHASE 0: DAY 1 SETUP ==="

# 1. Create project structure
mkdir -p ~/whale_hunter/{data,logs,scripts,config}
cd ~/whale_hunter

# 2. Initialize Python environment
python3 -m venv venv
source venv/bin/activate

# 3. Install dependencies
pip install aiohttp websockets pandas sqlite3 python-dotenv

# 4. Create SQLite database
python3 << 'EOF'
import sqlite3
conn = sqlite3.connect('data/wallet_graph.db')
cursor = conn.cursor()

# Create schema
cursor.execute("""
    CREATE TABLE IF NOT EXISTS wallets (
        address TEXT PRIMARY KEY,
        tier TEXT DEFAULT 'UNKNOWN',
        first_seen TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        total_pnl REAL DEFAULT 0,
        win_count INTEGER DEFAULT 0,
        loss_count INTEGER DEFAULT 0,
        last_win_date TIMESTAMP,
        confidence REAL DEFAULT 1.0,
        is_cex_funded BOOLEAN DEFAULT FALSE
    )
""")

cursor.execute("""
    CREATE TABLE IF NOT EXISTS signals (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        wallet TEXT,
        token TEXT,
        price REAL,
        amount_usd REAL,
        latency_estimate REAL,
        price_1h REAL,
        price_24h REAL,
        outcome TEXT,
        notes TEXT
    )
""")

cursor.execute("""
    CREATE TABLE IF NOT EXISTS funding (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        funder_address TEXT,
        funded_address TEXT,
        amount REAL,
        timestamp TIMESTAMP,
        edge_confidence REAL DEFAULT 1.0
    )
""")

cursor.execute("CREATE INDEX IF NOT EXISTS idx_signals_token ON signals(token)")
cursor.execute("CREATE INDEX IF NOT EXISTS idx_signals_wallet ON signals(wallet)")
cursor.execute("CREATE INDEX IF NOT EXISTS idx_funding_funder ON fund-
ing(funder_address)")

conn.commit()
print("✓ Database initialized")
EOF

# 5. Create config file
cat > config/settings.env << 'EOF'
# API Keys (get free tiers)
BIRDEYE_API_KEY=your_key_here
HELIUS_API_KEY=your_key_here

# RPC Endpoints
SOLANA_RPC=https://api.mainnet-beta.solana.com
SOLANA_WS=wss://api.mainnet-beta.solana.com

# Safety limits
MAX_POSITION_USD=50
MAX_DAILY_TRADES=10
EOF

echo "✓ Setup complete! Run: source venv/bin/activate && python scripts/
phase0_logger.py"
```

D.2 Phase 0 Logger Script (Enhanced)

```python
#!/usr/bin/env python3
"""
phase0_logger.py - Enhanced Phase 0 Signal Logger

Usage: python phase0_logger.py

Commands:
  - Paste token address: Log a whale signal
  - 'stats': Show current statistics
  - 'export': Export data to CSV
  - 'backtest': Run simple backtest on logged data
  - 'quit': Exit
"""

import asyncio
import aiohttp
import sqlite3
from datetime import datetime, timedelta
from dataclasses import dataclass
import os
import sys

# Load config
from dotenv import load_dotenv
load_dotenv('config/settings.env')

BIRDEYE_API = "https://public-api.birdeye.so/public/price"
DB_PATH = "data/wallet_graph.db"


@dataclass
class SignalLog:
    timestamp: datetime
    wallet: str
    token: str
    price: float
    amount_usd: float
    latency_estimate: float


class Phase0Logger:
    def __init__(self):
        self.conn = sqlite3.connect(DB_PATH)
        self.session = None

    async def start(self):
        self.session = aiohttp.ClientSession()

    async def stop(self):
        if self.session:
            await self.session.close()
        self.conn.close()

    async def get_price(self, token: str) -> dict:
        """Fetch current price from BirdEye"""
        try:
            headers = {"X-API-KEY": os.getenv("BIRDEYE_API_KEY", "")}
            params = {"address": token}

            async with self.session.get(BIRDEYE_API, params=params, headers=headers) a
s resp:
                data = await resp.json()
                return {
                    "price": data.get("data", {}).get("value", 0),
                    "success": True
                }
        except Exception as e:
            return {"price": 0, "success": False, "error": str(e)}

    def log_signal(self, signal: SignalLog):
        """Save signal to database"""
        cursor = self.conn.cursor()
        cursor.execute("""
            INSERT INTO signals (timestamp, wallet, token, price, amount_usd,
latency_estimate)
            VALUES (?, ?, ?, ?, ?, ?)
        """, (
            signal.timestamp.isoformat(),
            signal.wallet,
            signal.token,
            signal.price,
            signal.amount_usd,
            signal.latency_estimate
        ))
        self.conn.commit()

    def update_outcome(self, signal_id: int, price_1h: float, price_24h: float, out-
come: str):
        """Update signal with outcome data"""
        cursor = self.conn.cursor()
        cursor.execute("""
            UPDATE signals SET price_1h = ?, price_24h = ?, outcome = ?
            WHERE id = ?
        """, (price_1h, price_24h, outcome, signal_id))
        self.conn.commit()

    def get_stats(self) -> dict:
        """Get logging statistics"""
        cursor = self.conn.cursor()

        cursor.execute("SELECT COUNT(*) FROM signals")
        total = cursor.fetchone()[0]

        cursor.execute("SELECT COUNT(*) FROM signals WHERE outcome = 'WIN'")
        wins = cursor.fetchone()[0]

        cursor.execute("SELECT COUNT(*) FROM signals WHERE outcome = 'LOSS'")
        losses = cursor.fetchone()[0]

        cursor.execute("SELECT COUNT(*) FROM signals WHERE outcome IS NULL")
        pending = cursor.fetchone()[0]

        cursor.execute("SELECT COUNT(DISTINCT wallet) FROM signals")
        unique_wallets = cursor.fetchone()[0]

        cursor.execute("SELECT COUNT(DISTINCT token) FROM signals")
        unique_tokens = cursor.fetchone()[0]

        return {
            "total_signals": total,
            "wins": wins,
            "losses": losses,
            "pending": pending,
            "win_rate": wins / (wins + losses) if (wins + losses) > 0 else 0,
            "unique_wallets": unique_wallets,
            "unique_tokens": unique_tokens
        }

    def simple_backtest(self) -> dict:
        """Run simple backtest on logged data"""
        cursor = self.conn.cursor()
        cursor.execute("""
            SELECT token, price, price_1h, price_24h, outcome
            FROM signals
            WHERE price_1h IS NOT NULL AND price_24h IS NOT NULL
        """)

        results = {
            "total": 0,
            "1h_winners": 0,
            "24h_winners": 0,
            "avg_1h_return": 0,
            "avg_24h_return": 0
        }

        returns_1h = []
        returns_24h = []

        for row in cursor.fetchall():
            token, price, price_1h, price_24h, outcome = row
            results["total"] += 1

            if price > 0:
                ret_1h = (price_1h - price) / price
                ret_24h = (price_24h - price) / price

                returns_1h.append(ret_1h)
                returns_24h.append(ret_24h)

                if ret_1h > 0:
                    results["1h_winners"] += 1
                if ret_24h > 0:
                    results["24h_winners"] += 1

        if returns_1h:
            results["avg_1h_return"] = sum(returns_1h) / len(returns_1h)
        if returns_24h:
            results["avg_24h_return"] = sum(returns_24h) / len(returns_24h)

        return results


async def main():
    logger = Phase0Logger()
    await logger.start()

    print("=" * 60)
    print("PHASE 0 WHALE SIGNAL LOGGER (Enhanced)")
    print("=" * 60)
    print("\nCommands:")
    print("  - Paste token address: Log a new signal")
    print("  - 'stats': Show statistics")
    print("  - 'backtest': Run backtest")
    print("  - 'quit': Exit")
    print()

    while True:
        try:
            user_input = input("\n> ").strip()

            if user_input.lower() == 'quit':
                break

            elif user_input.lower() == 'stats':
                stats = logger.get_stats()
                print(f"""
Statistics:
───────────────────────────
Total signals:    {stats['total_signals']}
Wins:             {stats['wins']}
Losses:           {stats['losses']}
Pending:          {stats['pending']}
Win rate:         {stats['win_rate']:.1%}
Unique wallets:   {stats['unique_wallets']}
Unique tokens:    {stats['unique_tokens']}
""")

            elif user_input.lower() == 'backtest':
                results = logger.simple_backtest()
                print(f"""
Backtest Results:
───────────────────────────
Total with data:  {results['total']}
1h winners:       {results['1h_winners']} ({results['1h_winners']/
results['total']*100:.1f}% if results['total'] > 0 else 0)
24h winners:      {results['24h_winners']} ({results['24h_winners']/results['total']*1
00:.1f}% if results['total'] > 0 else 0)
Avg 1h return:    {results['avg_1h_return']*100:+.1f}%
Avg 24h return:   {results['avg_24h_return']*100:+.1f}%
""")

            elif len(user_input) > 30:  # Looks like a token address
                token = user_input

                # Get wallet (optional)
                wallet = input("Wallet address (or press Enter to skip): ").strip()
                if not wallet:
                    wallet = "UNKNOWN"

                # Get latency estimate
                latency_str = input("Estimated seconds since signal [default=5]: ").st
rip()
                latency = float(latency_str) if latency_str else 5.0

                # Get amount (optional)
                amount_str = input("USD amount (or press Enter for unknown): ").strip(
)
                amount = float(amount_str) if amount_str else 0.0

                # Fetch price
                print("Fetching price...")
                price_data = await logger.get_price(token)

                if not price_data["success"]:
                    print(f"Warning: Could not fetch price:
{price_data.get('error')}")

                # Log signal
                signal = SignalLog(
                    timestamp=datetime.utcnow(),
                    wallet=wallet,
                    token=token,
                    price=price_data["price"],
                    amount_usd=amount,
                    latency_estimate=latency
                )

                logger.log_signal(signal)

                print(f"""
✓ LOGGED:
  Token:    {token[:20]}...
  Wallet:   {wallet[:20] if wallet != "UNKNOWN" else "Unknown"}...
  Price:    ${price_data['price']:.8f}
  Latency:  {latency}s
  Saved to database
""")

            else:
                print("Unknown command. Type 'quit' to exit.")

        except KeyboardInterrupt:
            break
        except Exception as e:
            print(f"Error: {e}")

    await logger.stop()
    print("\nGoodbye!")


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

D.3 Harvester Script (Reverse Search)

```
#!/usr/bin/env python3
"""
harvester.py - Winner Reverse Search Script

Finds profitable wallets by reverse-searching 10x tokens.
Run daily to build your Black Book.
"""

import asyncio
import aiohttp
import sqlite3
from datetime import datetime, timedelta
from typing import List, Dict
import os

DB_PATH = "data/wallet_graph.db"
BIRDEYE_API = "https://public-api.birdeye.so"
HELIUS_API = "https://api.helius.xyz/v0"


class WinnerHarvester:
    def __init__(self):
        self.conn = sqlite3.connect(DB_PATH)
        self.session = None
        self.birdeye_key = os.getenv("BIRDEYE_API_KEY", "")
        self.helius_key = os.getenv("HELIUS_API_KEY", "")

    async def start(self):
        self.session = aiohttp.ClientSession()

    async def stop(self):
        if self.session:
            await self.session.close()
        self.conn.close()

    async def get_top_gainers(self, hours: int = 24, min_gain: float = 10.0) ->
List[Dict]:
        """
        Get tokens with 10x+ gains in last 24h.
        Uses BirdEye API.
        """
        gainers = []

        try:
            headers = {"X-API-KEY": self.birdeye_key}

            async with self.session.get(
                f"{BIRDEYE_API}/defi/token_trending",
                headers=headers,
                params={"sort_by": "v24hChangePercent", "sort_type": "desc", "offset":
0, "limit": 50}
            ) as resp:
                data = await resp.json()

                for token in data.get("data", {}).get("tokens", []):
                    change_pct = token.get("v24hChangePercent", 0)

                    if change_pct >= min_gain * 100:  # 10x = 1000%
                        gainers.append({
                            "address": token.get("address"),
                            "symbol": token.get("symbol"),
                            "change_pct": change_pct,
                            "market_cap": token.get("mc", 0)
                        })

        except Exception as e:
            print(f"Error fetching gainers: {e}")

        return gainers[:10]  # Top 10

    async def get_early_buyers(self, token_address: str, limit: int = 50) ->
List[Dict]:
        """
        Get first N buyers of a token using Helius.
        """
        buyers = []

        try:
            # Get token creation signature
            async with self.session.get(
                f"{HELIUS_API}/addresses/{token_address}/transactions",
                params={"api-key": self.helius_key, "type": "SWAP", "limit": limit}
            ) as resp:
                data = await resp.json()

                for tx in data:
                    # Extract buyer wallet
                    if tx.get("type") == "SWAP":
                        # Parse transaction for buyer address
                        fee_payer = tx.get("feePayer")
                        timestamp = tx.get("timestamp")

                        if fee_payer:
                            buyers.append({
                                "address": fee_payer,
                                "timestamp": datetime.fromtimestamp(timestamp) if
timestamp else None
                            })

        except Exception as e:
            print(f"Error fetching buyers for {token_address[:8]}: {e}")

        return buyers

    async def get_wallet_funding_source(self, wallet: str) -> Dict:
        """
        Get the primary funding source for a wallet.
        """
        try:
            async with self.session.get(
                f"{HELIUS_API}/addresses/{wallet}/transactions",
                params={"api-key": self.helius_key, "type": "TRANSFER", "limit": 10}
            ) as resp:
                data = await resp.json()

                for tx in data:
                    # Look for incoming SOL transfers
                    if tx.get("type") == "TRANSFER":
                        # Parse for funder
                        from_addr = tx.get("nativeTransfers", [{}])[0].get("fromUser-
Account")
                        if from_addr and from_addr != wallet:
                            return {
                                "funder": from_addr,
                                "amount": tx.get("nativeTransfers", [{}])[0].get("amou
nt", 0) / 1e9,
                                "timestamp": tx.get("timestamp")
                            }

        except Exception as e:
            print(f"Error fetching funding for {wallet[:8]}: {e}")

        return {}

    def save_wallet(self, address: str, tier: str = "CANDIDATE"):
        """Save wallet to database"""
        cursor = self.conn.cursor()
        cursor.execute("""
            INSERT OR IGNORE INTO wallets (address, tier) VALUES (?, ?)
        """, (address, tier))
        self.conn.commit()

    def save_funding(self, funder: str, funded: str, amount: float, timestamp):
        """Save funding relationship"""
        cursor = self.conn.cursor()
        cursor.execute("""
            INSERT INTO funding (funder_address, funded_address, amount, timestamp)
            VALUES (?, ?, ?, ?)
        """, (funder, funded, amount, timestamp))
        self.conn.commit()

    async def harvest_daily(self):
        """
        Daily harvest routine:
        1. Get top 10x tokens
        2. Get early buyers
        3. Trace funding sources
        4. Build graph
        """
        print("=" * 60)
        print(f"DAILY HARVEST - {datetime.utcnow().isoformat()}")
        print("=" * 60)

        # Step 1: Get winners
        print("\n[1/3] Finding 10x+ tokens...")
        gainers = await self.get_top_gainers(hours=24, min_gain=10.0)
        print(f"Found {len(gainers)} tokens with 10x+ gains")

        for gainer in gainers:
            print(f"  • {gainer['symbol']}: +{gainer['change_pct']:.0f}%")

        # Step 2: Get early buyers for each
        print("\n[2/3] Finding early buyers...")
        all_buyers = []

        for gainer in gainers:
            buyers = await self.get_early_buyers(gainer["address"], limit=20)
            print(f"  • {gainer['symbol']}: {len(buyers)} early buyers")

            for buyer in buyers:
                buyer["token"] = gainer["address"]
                buyer["token_symbol"] = gainer["symbol"]
                all_buyers.append(buyer)

            await asyncio.sleep(0.5)  # Rate limiting

        # Step 3: Trace funding sources
        print("\n[3/3] Tracing funding sources...")
        funding_graph = {}

        for buyer in all_buyers[:50]:  # Limit to avoid rate limits
            funding = await self.get_wallet_funding_source(buyer["address"])

            if funding.get("funder"):
                self.save_wallet(buyer["address"], "EARLY_BUYER")
                self.save_wallet(funding["funder"], "FUNDER")
                self.save_funding(
                    funding["funder"],
                    buyer["address"],
                    funding["amount"],
                    funding.get("timestamp")
                )

                # Track for mother wallet detection
                funder = funding["funder"]
                if funder not in funding_graph:
                    funding_graph[funder] = []
                funding_graph[funder].append(buyer["address"])

                print(f"  • {buyer['address'][:8]}... ← funded by {funder[:8]}...")

            await asyncio.sleep(0.3)  # Rate limiting

        # Step 4: Identify potential mother wallets
        print("\n[RESULTS] Potential Mother Wallets:")
        for funder, children in sorted(funding_graph.items(), key=lambda x: -len(x[1])
):
            if len(children) >= 2:
                print(f"  🎯 {funder[:12]}... funded {len(children)} early buyers")

        print("\n✓ Harvest complete!")


async def main():
    harvester = WinnerHarvester()
    await harvester.start()

    try:
        await harvester.harvest_daily()
    finally:
        await harvester.stop()


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

D.4 God View Query Script

```
#!/usr/bin/env python3
"""
god_view.py - Find Mother Wallets in Your Data

The "God View" query to identify insider clusters.
"""

import sqlite3
from datetime import datetime, timedelta
from typing import List, Tuple

DB_PATH = "data/wallet_graph.db"


def find_mother_wallets(min_children: int = 2) -> List[Tuple]:
    """
    Find wallets that funded multiple winning traders.
    This is the core "God View" query.
    """
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()

    query = """
        SELECT
            f.funder_address as mother,
            COUNT(DISTINCT f.funded_address) as child_count,
            GROUP_CONCAT(DISTINCT f.funded_address) as children,
            SUM(w.win_count) as total_child_wins,
            AVG(w.confidence) as avg_confidence
        FROM funding f
        JOIN wallets w ON f.funded_address = w.address
        WHERE w.tier IN ('EARLY_BUYER', 'S_TIER', 'A_TIER', 'B_TIER')
        GROUP BY f.funder_address
        HAVING child_count >= ?
        ORDER BY child_count DESC, total_child_wins DESC
        LIMIT 50
    """

    cursor.execute(query, (min_children,))
    results = cursor.fetchall()
    conn.close()

    return results


def find_connected_wallets(wallet: str, max_hops: int = 2) -> List[Tuple]:
    """
    Find all wallets connected to a given wallet within N hops.
    """
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()

    if max_hops == 1:
        query = """
            SELECT funded_address, 'CHILD', amount, timestamp
            FROM funding WHERE funder_address = ?
            UNION
            SELECT funder_address, 'PARENT', amount, timestamp
            FROM funding WHERE funded_address = ?
        """
        cursor.execute(query, (wallet, wallet))
    else:
        # 2-hop query
        query = """
            -- Direct children
            SELECT f1.funded_address, 'CHILD_1HOP', f1.amount, f1.timestamp
            FROM funding f1 WHERE f1.funder_address = ?

            UNION

            -- Grandchildren
            SELECT f2.funded_address, 'CHILD_2HOP', f2.amount, f2.timestamp
            FROM funding f1
            JOIN funding f2 ON f1.funded_address = f2.funder_address
            WHERE f1.funder_address = ?

            UNION

            -- Direct parents
            SELECT f1.funder_address, 'PARENT_1HOP', f1.amount, f1.timestamp
            FROM funding f1 WHERE f1.funded_address = ?

            UNION

            -- Grandparents
            SELECT f2.funder_address, 'PARENT_2HOP', f2.amount, f2.timestamp
            FROM funding f1
            JOIN funding f2 ON f1.funder_address = f2.funded_address
            WHERE f1.funded_address = ?
        """
        cursor.execute(query, (wallet, wallet, wallet, wallet))

    results = cursor.fetchall()
    conn.close()

    return results


def get_black_book() -> List[Tuple]:
    """
    Get the Black Book: Top mother wallets to monitor.
    """
    mothers = find_mother_wallets(min_children=2)

    # Format for output
    black_book = []
    for mother in mothers:
        address, child_count, children, wins, confidence = mother
        black_book.append({
            "address": address,
            "children": child_count,
            "total_wins": wins or 0,
            "confidence": confidence or 0
        })

    return black_book


def main():
    print("=" * 60)
    print("GOD VIEW - MOTHER WALLET DISCOVERY")
    print("=" * 60)

    print("\n[1] Finding Mother Wallets...")
    mothers = find_mother_wallets(min_children=2)
    if not mothers:
        print("No mother wallets found yet. Run harvester.py first!")
        return

    print(f"\nFound {len(mothers)} potential mother wallets:\n")
    print("-" * 80)
    print(f"{'Mother Wallet':<45} {'Children':<10} {'Wins':<8} {'Conf':<8}")
    print("-" * 80)

    for mother in mothers[:20]:
        address, child_count, children, wins, confidence = mother
        print(f"{address:<45} {child_count:<10} {wins or 0:<8} {confidence or 0:.2f}")

    print("\n[2] Black Book (Monitor These):")
    print("-" * 60)

    black_book = get_black_book()
    for i, entry in enumerate(black_book[:10], 1):
        print(f"{i}. {entry['address']}")
        print(f"   Children: {entry['children']}, Wins: {entry['total_wins']}, Confid-
ence: {entry['confidence']:.2f}")

    print("\n✓ Export these addresses for wallet monitoring!")


if __name__ == "__main__":
    main()
```

7. Part E: Infrastructure Economics & Realistic Timeline

E.1 Infrastructure Cost Analysis

```
CRITICAL RULE: Infrastructure cost must be <5% of capital annually.
Delay expensive components until capital justifies them.
```

Phase	Capital	Monthly Infra Cost	Annual Cost	Cost as % Capital	Status
Phase 0	$0	$0	$0	0%	✅ Free
Phase 1	$500-$2,000	$0-$20	$0-$240	0-12%	⚠️ Keep minimal
Phase 2	$2,000-$5,000	$20-$50	$240-$600	4-12%	⚠️ At threshold
Phase 3	$5,000-$10,000	$50-$100	$600-$1,200	6-12%	⚠️ Careful
Phase 4	$10,000+	$100-$200	$1,200-$2,400	2-4%	✅ Justified

Cost Breakdown by Service

Service	Free Tier	Paid Tier	When to Upgrade
Helius RPC	500K credits/month	$99/month	When hitting limits
Helius Webhooks	100 webhooks	Included in Pro	When >50 tracked wallets
BirdEye API	100 req/min	$29/month	When building real-time
VPS (Hetzner)	N/A	€5-20/month	Phase 2+
Jito Tips	N/A	Variable (0.001+ SOL/tx)	Phase 3+
Neo4j	Free self-hosted	N/A	When >5,000 wallets

Phase-Specific Recommendations

Phase 0-1 ($0-$2,000):
- Use ONLY free tiers
- SQLite for graph
- Standard Solana RPC
- Total cost: $0

Phase 2 ($2,000-$5,000):
- Helius free tier (likely sufficient)
- Small VPS (~$10/month)
- Total cost: $10-$30/month

Phase 3 ($5,000-$10,000):
- Helius Pro ($99/month) - IF needed

- Better VPS ($20-$40/month)
- Jito tips (~$20-$50/month)
- Total cost: $50-$150/month

E.2 Realistic Timeline (Revised)

```
TIMELINE REALITY CHECK:

Original V3.0 estimate: 18 months
Revised realistic estimate: 24-36 months

Rust proficiency alone: 6 months (not 2 months)
Full Hydra system: 24-36 months
```

Detailed Milestone Timeline

Month	Phase	Milestone	Key Deliverable
1-2	0	Setup + Learning	Database, logger script, manual tracking
3-4	0	Paper Trading	100+ logged signals with outcomes
5-6	1	Live V2.0 ($500)	First real trades, cautious
7-9	1	V2.0 Refinement	Backtested performance data
10-12	1-2	Simulation Gate	Shadow executor running
13-15	2	Graph Integration	SQLite graph live, harvester running
16-18	2.5	V2.5 Live	Graph as confidence boost working
19-21	2.5	Rust Learning	Sidecar pattern development
22-24	3	Jito Testing	Small-scale atomic execution tests
25-30	3	V3.0 Components	Gradual activation IF metrics pass
31-36	4	Full Hydra	Complete system with counter-adaptation

Go/No-Go Gates

Gate 1: Phase 0 → Phase 1 (Month 5)
- [ ] 100+ logged signals
- [ ] Simulated win rate >50%
- [ ] Backtest shows positive expected value
- [ ] Understand all V2.0 gates

Gate 2: Phase 1 → Phase 2 (Month 10)
- [ ] 50+ live trades completed
- [ ] Actual win rate >45%

- [ ] No catastrophic losses (>20% capital in single trade)
- [ ] Capital ≥$1,500

Gate 3: Phase 2 → Phase 2.5 (Month 15)
- [ ] Simulation accuracy >90%
- [ ] Graph database has 500+ wallets
- [ ] At least 5 mother wallets identified
- [ ] Capital ≥$2,500

Gate 4: Phase 2.5 → Phase 3 (Month 19)
- [ ] Capital >$5,000
- [ ] Win rate >55% sustained over 3 months
- [ ] Positive ROI after all infrastructure costs
- [ ] Simulation accuracy >95%
- [ ] Rust sidecar operational

Gate 5: Phase 3 → Phase 4 (Month 25)
- [ ] Capital >$10,000
- [ ] Jito bundles profitable over 2+ months
- [ ] Counter-adaptation measures tested
- [ ] Full system integration validated

---

8. Appendices

Appendix A: CEX Hot Wallet Blacklist

```
"""
Complete CEX Hot Wallet Blacklist
Update periodically from blockchain explorers
"""

CEX_HOT_WALLETS_SOLANA = {
    # Binance
    "5tzFkiKscXHK5ZXCGbXZxdw7gTjjD1mBwuoFbhUvuAi9": "Binance-1",
    "9WzDXwBbmkg8ZTbNMqUxvQRAyrZzDsGYdLVL9zYtAWWM": "Binance-2",
    "3yFwqXBfZY4jBVUafQ1YEXw189y2dN3V5KQq9uzBDy1E": "Binance-3",
    "HVdSdT2M4Sy9WdFBDqhpPYbW4Z3tz2fEfXfZJyKvTJiV": "Binance-4",

    # Coinbase
    "GJRs4FwHtemZ5ZE9x3FNvJ8TMwitKTh21yxdRPqn7npE": "Coinbase-1",
    "H8sMJSCQxfKiFTCfDR3DUMLPwcRbM61LGFJ8N4dK3WjS": "Coinbase-2",
    "2AQdpHJ2JpcEgPiATUXjQxA8QmafFegfQwSLWSprPicm": "Coinbase-3",

    # Kraken
    "JA98mShDSf4qqxVpVvGTwRZ8NrqbUhpUhgHy8gF3K1bJ": "Kraken-1",
    "CuieVDEDtLo7FypA9SbLM9saXFdb1dsshEkyErMqkRQq": "Kraken-2",

    # OKX
    "5VCwKtCXgCJ6kit5FybXjvriW3xELsFDhYrPSqtJNmcD": "OKX-1",

    # Bybit
    "AC5RDfQFmDS1deWZos921JfqscXdByf8BKHs5ACWjtW2": "Bybit-1",

    # KuCoin
    "BmFdpraQhkiDQE6SnfG5omcA1VwzqfXrwtNYBwWTymy6": "KuCoin-1",

    # Gate.io
    "u6PJ8DtQuPFnfmwHbGFULQ4u4EgjDiyYKjVEsynXq2w": "Gate-1",

    # Crypto.com
    "AobVSwdW9BbpMdJvTqeCN4hPAmh4rHm7vwLnQ5ATSPo9": "Crypto.com-1",
}

CEX_HOT_WALLETS_ETHEREUM = {
    # Binance
    "0x28C6c06298d514Db089934071355E5743bf21d60": "Binance-14",
    "0x21a31Ee1afC51d94C2eFcCAa2092aD1028285549": "Binance-15",
    "0xDFd5293D8e347dFe59E90eFd55b2956a1343963d": "Binance-16",
    "0x56Eddb7aa87536c09CCc2793473599fD21A8b17F": "Binance-17",
    "0x9696f59E4d72E237BE84fFD425DCaD154Bf96976": "Binance-18",

    # Coinbase
    "0x503828976D22510aad0201ac7EC88293211D23Da": "Coinbase-1",
    "0xddfAbCdc4D8FfC6d5beaf154f18B778f892A0740": "Coinbase-2",
    "0x71660c4005BA85c37ccec55d0C4493E66Fe775d3": "Coinbase-3",
    "0xA9D1e08C7793af67e9d92fe308d5697FB81d3E43": "Coinbase-4",

    # Kraken
    "0x2910543Af39abA0Cd09dBb2D50200b3E800A63D2": "Kraken-1",
    "0x0A869d79a7052C7f1b55a8EbAbbEa3420F0D1E13": "Kraken-2",
    "0xE853c56864A2ebe4576a807D26Fdc4A0adA51919": "Kraken-3",
    "0x267be1C1D684F78cb4F6a176C4911b741E4Ffdc0": "Kraken-4",

    # OKX
    "0x6cC5F688a315f3dC28A7781717a9A798a59fDA7b": "OKX-1",
    "0x236F9F97e0E62388479bf9E5BA4889e46B0273C3": "OKX-2",
}
```

```
def is_cex_hot_wallet(address: str, chain: str = "solana") -> tuple:
    """Check if address is a known CEX hot wallet"""
    if chain == "solana":
        wallets = CEX_HOT_WALLETS_SOLANA
    elif chain == "ethereum":
        wallets = CEX_HOT_WALLETS_ETHEREUM
    else:
        return False, ""

    if address in wallets:
        return True, wallets[address]

    return False, ""
```

Appendix B: Quick Reference Card

```
╔══════════════════════════════════════════════════════════════════════╗
║                       V3.1 QUICK REFERENCE CARD                      ║
╠══════════════════════════════════════════════════════════════════════╣
║                                                                      ║
║  CRITICAL RULES:                                                     ║
║  ────────────────────────────────────────────────────────────────    ║
║  □ Assassin ONLY if Simulator ≥95% accuracy                          ║
║  □ Infrastructure cost <5% of capital annually                       ║
║  □ Target re-accumulation (1-2h tokens), not launch sniping          ║
║  □ Trust earned slowly, lost quickly (30-day half-life)              ║
║  □ When in doubt, use V2.0 defensive mode                            ║
║                                                                      ║
║  KILL SWITCH TRIGGERS:                                               ║
║  ────────────────────────────────────────────────────────────────    ║
║  □ >10 new mother wallets in 24h → HALT                              ║
║  □ Win rate <30% across 3+ clusters → HALT                           ║
║  □ Correlated cluster emergence → HALT                               ║
║  → Action: Freeze graph signals, V2.0 only, 72h observation          ║
║                                                                      ║
║  DAILY CHECKLIST:                                                    ║
║  ────────────────────────────────────────────────────────────────    ║
║  □ Check kill switch triggers                                        ║
║  □ Run confidence decay                                              ║
║  □ Review overnight signals                                          ║
║  □ Update Black Book if winners identified                           ║
║                                                                      ║
║  POSITION SIZING:                                                    ║
║  ────────────────────────────────────────────────────────────────    ║
║  □ Phase 1: Max 10% per trade ($50 on $500)                          ║
║  □ Phase 2: Max 5% per trade ($100 on $2,000)                        ║
║  □ Phase 3: Max 3% per trade ($150 on $5,000)                        ║
║  □ Phase 4: Max 2% per trade ($200 on $10,000)                       ║
║                                                                      ║
║  TOKEN AGE MINIMUMS:                                                 ║
║  ────────────────────────────────────────────────────────────────    ║
║  □ Meme coins: 1 hour minimum                                        ║
║  □ Mid caps: 30 minutes minimum                                      ║
║  □ Large caps: No minimum                                            ║
║                                                                      ║
║  GO/NO-GO GATES:                                                     ║
║  ────────────────────────────────────────────────────────────────    ║
║  □ V2.5 → V3.0: Capital >$5K, Win rate >55%, ROI positive            ║
║  □ Full Hydra: Capital >$10K, 3 months profitable                    ║
║                                                                      ║
╚══════════════════════════════════════════════════════════════════════╝
```

Appendix C: File Structure

```
~/whale_hunter/
├── config/
│   └── settings.env          # API keys, RPC endpoints
├── data/
│   ├── wallet_graph.db       # SQLite graph database
│   └── phase0_signals.csv    # Legacy CSV format
├── scripts/
│   ├── phase0_logger.py      # Phase 0 signal logging
│   ├── harvester.py          # Daily winner harvester
│   ├── god_view.py           # Mother wallet discovery
│   ├── backtest.py           # Historical backtesting
│   └── monitor.py            # Live monitoring (Phase 2+)
├── assassin_sidecar/         # Rust execution sidecar (Phase 3+)
│   ├── Cargo.toml
│   └── src/
│       └── main.rs
├── logs/
│   ├── trades.log
│   ├── signals.log
│   └── errors.log
└── venv/                     # Python virtual environment
```

Document History

Version	Date	Changes
3.0	Jan 2026	Initial Dark Forest architecture
3.1 FINAL	Jan 2026	Expert feedback integration: +17 improvements

---

END OF DOCUMENT

“In the Dark Forest, the survivors are not the strongest or fastest—they are the ones who remain invisible while striking precisely. Be the ghost predator.”