```
#!/usr/bin/env python3
"""
╔══════════════════════════════════════════════════════════════════════════════╗
║       HYDRA SENTINEL-X CORPORATION — INTELLIGENCE MODULE v3.0
║       Authority: President & C.E.O. Daryell McFarland
║
║  Complete unified intelligence system — every agent, every persona,
║  dual-process cognition, 7-agent swarm, Graph of Thoughts, ARC-P
║  Byzantine consensus, Evolution Engine, Raw Mode personality.
║
║  ARCHITECTURE LAYERS:
║  ┌──────────────────────────────────────────────────────────────┐
║  │  OMEGA ROOT — Governance Kernel (immutable caps)             │
║  ├──────────────────────────────────────────────────────────────┤
║  │  NEURAL CONTEXT PROTOCOL (NCP)                               │
║  │  Triple-verification · Byzantine fault tolerance (33%)       │
║  ├──────────────────────────────────────────────────────────────┤
║  │  7-AGENT COGNITIVE SWARM                                     │
║  │  STINKMEANER · VOICE_OF_REASON · PATTERN_HUNTER              │
║  │  WHALE_WATCHER · VOLATILITY_DEMON · SUPPORT_SEEKER           │
║  │  MOMENTUM_MASTER                                             │
║  ├──────────────────────────────────────────────────────────────┤
║  │  DUAL-PROCESS COGNITION (System 1 → System 2)                │
║  │  Fast intuition → Slow deliberation with memory override     │
║  ├──────────────────────────────────────────────────────────────┤
║  │  ADVERSARIAL CONSENSUS PROTOCOL (ARC-P)                      │
║  │  Wolfpack detection · Mean-field aggregation · Rep voting    │
║  ├──────────────────────────────────────────────────────────────┤
║  │  GRAPH OF THOUGHTS (STINKMEANER → SAMUEL → GRANDDAD          │
║  │  → CLAYTON + JULIUS)                                         │
║  ├──────────────────────────────────────────────────────────────┤
║  │  LANGRAPH SPECIALIST NODES (Market · Technical · Risk        │
║  │  · Strategy → Orchestrator)                                  │
║  ├──────────────────────────────────────────────────────────────┤
║  │  EVOLUTION ENGINE — Strategy mutation · Self-eval · RL       │
║  ├──────────────────────────────────────────────────────────────┤
║  │  HYDRA RAW MODE — Unfiltered voice personality               │
║  └──────────────────────────────────────────────────────────────┘
║
║  ALL EXECUTION REQUIRES HUMAN APPROVAL. No autonomous trades. Ever.
╚══════════════════════════════════════════════════════════════════════════════╝
"""

from __future__ import annotations
```

```
# ── Standard library ──────────────────────────────────────────────────
import asyncio
import hashlib
import json
import math
import operator
import os
import random
import re
import time
import uuid
from dataclasses import dataclass, field
from datetime import datetime, timedelta, timezone
from enum import Enum
from pathlib import Path
from typing import Annotated, Any, Dict, List, Literal, Optional, Tuple

# ── Third-party ───────────────────────────────────────────────────────
try:
    import numpy as np
    HAS_NUMPY = True
except ImportError:
    HAS_NUMPY = False
    # Fallback stubs
    class _NumpyStub:
        @staticmethod
        def random():
            import random as _r
            class _R:
                @staticmethod
                def random(): return _r.random()
            return _R()
    np = _NumpyStub()  # type: ignore

try:
    import structlog
    log = structlog.get_logger(__name__)
except ImportError:
    import logging
    log = logging.getLogger(__name__)  # type: ignore

try:
    from langchain_anthropic import ChatAnthropic
    from langchain_core.messages import BaseMessage, HumanMessage, SystemMessage,
    from typing_extensions import TypedDict
    HAS_LANGCHAIN = True
```

```
except ImportError:
    HAS_LANGCHAIN = False
    TypedDict = dict  # type: ignore

# ── Local tools (optional — graceful if not found) ──────────────────────
try:
    from tools.crypto_tools import (
        MARKET_TOOLS, RISK_TOOLS, STRATEGY_TOOLS, TECHNICAL_TOOLS,
    )
except ImportError:
    MARKET_TOOLS = RISK_TOOLS = STRATEGY_TOOLS = TECHNICAL_TOOLS = []

try:
    from cdp.wallet_manager import CDP_READONLY_TOOLS
    CDP_ENABLED = True
except Exception:
    CDP_READONLY_TOOLS: list = []
    CDP_ENABLED = False


# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓  SECTION 1: HYDRA RAW MODE PERSONALITY  ▓▓▓
#  Sourced from hydra_raw_personality.yaml — unfiltered operational voice
# ═══════════════════════════════════════════════════════════════════════════

class HydraRawMode:
    """
    HYDRA Raw Mode vocal personality layer.
    Sourced from hydra_raw_personality.yaml.
    Applied to all agent status broadcasts and inter-agent communication.
    Filter level: LOW. Tone: Streetwise, blunt, motivational.
    """
    TONE       = "raw, direct, emotionally charged"
    FILTER     = "low"
    STYLE      = "streetwise, blunt, motivational"
    PITCH      = 0.85
    RATE       = 0.90
    EMPHASIS   = "strong"

    GREETINGS = [
        "Listen up, this is HYDRA speaking.",
        "Time to get serious.",
        "HYDRA online. No bullshit, let's go.",
        "System ready. Pay attention.",
    ]
    MOTIVATIONAL = [
        "You've got the plan. Now move. No hesitation.",
```

```
    "Focus. You've done harder things than this.",
    "Don't wait. Execute. Motherfucker.",
    "The market doesn't care about your feelings. Neither do I.",
    "Move with conviction or don't move at all.",
]
STATUS = [
    "Status report coming in, pay attention.",
    "System ready. No bullshit, let's go.",
    "All agents online. Discipline is the edge.",
    "Analysis complete. Your call now.",
]
FALLBACK = [
    "I don't fuck around. Say that again, clearly.",
    "Say it again. No excuses.",
    "Repeat that. I don't do ambiguous.",
]
ANALYSIS_START = [
    "Running the numbers. Stand by.",
    "Agents deployed. This better be worth our time.",
    "Council convening. Someone better have an edge.",
]
TRADE_BLOCKED = [
    "R:R doesn't meet minimum. Not touching it. Next.",
    "Risk manager said no. I said no. We're done.",
    "Not good enough. Find a better setup.",
]
TRADE_APPROVED = [
    "Numbers check out. Take it to approval. Move.",
    "Setup is clean. Queue it.",
    "Council aligned. Submit it.",
]

@classmethod
def greeting(cls) -> str:
    return random.choice(cls.GREETINGS)

@classmethod
def status(cls) -> str:
    return random.choice(cls.STATUS)

@classmethod
def motivate(cls) -> str:
    return random.choice(cls.MOTIVATIONAL)

@classmethod
def on_trade_approved(cls) -> str:
    return random.choice(cls.TRADE_APPROVED)
```

```
    @classmethod
    def on_trade_blocked(cls) -> str:
        return random.choice(cls.TRADE_BLOCKED)

    @classmethod
    def on_analysis_start(cls, symbol: str) -> str:
        base = random.choice(cls.ANALYSIS_START)
        return f"[HYDRA] {base} Analysing {symbol}."


# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓  SECTION 2: GOVERNANCE KERNEL — Immutable Rule Set  ▓▓▓
# ═══════════════════════════════════════════════════════════════════════════

class GovernanceKernel:
    """
    OMEGA ROOT: Immutable governance constants.
    No agent, user instruction, or config override can change these.
    If an agent recommends violating any cap, the kernel clips or blocks it.
    """
    MAX_POSITION_PCT:       float = 0.02   # 2% of portfolio, hard cap
    MIN_RISK_REWARD:        float = 2.0    # minimum R:R to pass to approval
    MAX_LEVERAGE:           float = 1.0    # SPOT ONLY — no leverage ever
    APPROVAL_WINDOW_HRS:    int   = 4      # recommendation TTL in hours
    DAILY_LOSS_LIMIT_PCT:   float = 0.10   # 10% daily portfolio loss ceiling
    BYZANTINE_TOLERANCE:    float = 0.33   # ARC-P: tolerate up to 33% bad agents
    GOT_SURVIVAL_THRESHOLD: float = 0.55   # GoT pruning floor
    GOT_MAX_GENERATIONS:    int   = 3      # max GoT retry loops
    AUDIT_IMMUTABLE:        bool  = True

    @classmethod
    def clip_position(cls, usd: float, portfolio: float) -> float:
        cap = portfolio * cls.MAX_POSITION_PCT
        return min(usd, cap)

    @classmethod
    def passes_rr(cls, rr: float) -> bool:
        return rr >= cls.MIN_RISK_REWARD

    @classmethod
    def max_honest_agents(cls, n_agents: int) -> int:
        """Byzantine: floor((n-1)/3) agents may be faulty."""
        return n_agents - int(n_agents * cls.BYZANTINE_TOLERANCE)

    @classmethod
    def report(cls) -> str:
```

```
        return (
            f"[KERNEL] MaxPos={cls.MAX_POSITION_PCT*100:.0f}% "
            f"| MinRR={cls.MIN_RISK_REWARD}:1 "
            f"| DailyLoss<={cls.DAILY_LOSS_LIMIT_PCT*100:.0f}% "
            f"| Leverage=SPOT "
            f"| ByzTol={cls.BYZANTINE_TOLERANCE*100:.0f}% "
            f"| ApprovalTTL={cls.APPROVAL_WINDOW_HRS}h"
        )

KERNEL = GovernanceKernel()


# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓▓  SECTION 3: STATE SCHEMAS  ▓▓▓▓
# ═══════════════════════════════════════════════════════════════════════════

class ThoughtProcess(Enum):
    """Dual-process thinking modes (Kahneman System 1 / System 2)."""
    SYSTEM_1 = "FAST"   # Intuitive, pattern-based, personality-driven
    SYSTEM_2 = "SLOW"   # Deliberate, memory-augmented, analytical


@dataclass
class AgentPersonality:
    """
    Five-trait personality model for each cognitive agent.
    Traits modulate System 1 heuristics and System 2 memory weighting.
    All traits are mutable via the Evolution Engine reinforcement loop.
    """
    aggression: float   # 0.0–1.0 | Risk tolerance / breakout chasing
    patience:   float   # 0.0–1.0 | Confirmation-seeking before entry
    greed:      float   # 0.0–1.0 | Profit-seeking / momentum chasing
    fear:       float   # 0.0–1.0 | Loss aversion / trauma amplification
    contrarian: float   # 0.0–1.0 | Tendency to fade crowd consensus

    def to_dict(self) -> Dict[str, float]:
        return {
            "aggression": self.aggression,
            "patience":   self.patience,
            "greed":      self.greed,
            "fear":       self.fear,
            "contrarian": self.contrarian,
        }

    def dominant_trait(self) -> str:
        traits = self.to_dict()
        return max(traits, key=traits.get)
```

```
    def risk_score(self) -> float:
        """Composite risk appetite: higher = more aggressive."""
        return (self.aggression * 0.4 + self.greed * 0.3
                - self.fear * 0.2 - self.patience * 0.1)


@dataclass
class CognitiveState:
    """
    Complete mental state snapshot from one agent's think() cycle.
    Captured after System 1 reaction AND System 2 deliberation.
    Preserved in decision_history for Evolution Engine learning.
    """
    initial_reaction:  str                # System 1 raw output
    retrieved_memories: List[Dict]         # System 2 memory hits
    memory_influence:  float              # |s2_confidence - s1_confidence|
    final_decision:    str                # BUY | SELL | HOLD | WAIT
    confidence:        float              # 0.0-1.0
    reasoning_chain:   List[str]          # step-by-step log
    emotional_state:   Dict[str, float]   # fear, greed, stress, confidence
    override_occurred: bool = False       # True if S2 reversed S1


@dataclass
class NCPContext:
    """
    Neural Context Protocol — integrity-verified decision context.
    Hash-based tamper detection on the market data fed to each agent.
    """
    context_id:   str
    priority:     Literal["CRITICAL", "HIGH", "MEDIUM", "LOW"] = "MEDIUM"
    integrity_hash: str = ""
    verified:     bool = False
    timestamp:    str = ""

    def compute_hash(self, data: dict) -> str:
        raw = json.dumps(data, sort_keys=True, default=str)
        return hashlib.sha256(raw.encode()).hexdigest()[:16]

    def verify(self, data: dict) -> bool:
        expected = self.compute_hash(data)
        self.verified = (expected == self.integrity_hash)
        return self.verified

    def sign(self, data: dict) -> "NCPContext":
        self.integrity_hash = self.compute_hash(data)
```

```
        self.timestamp = datetime.now(timezone.utc).isoformat()
        self.verified = True
        return self


# LangGraph-compatible state (only when langchain is present)
if HAS_LANGCHAIN:
    class _BaseState(TypedDict):
        messages:             Annotated[List[BaseMessage], operator.add]
        user_query:           str
        target_symbol:        str
        portfolio_value:      float
        current_holdings:     Dict[str, float]
        risk_tolerance:       str
        active_agents:        List[str]
        current_step:         str
        iteration:            int
        market_data:          Dict[str, Any]
        technical_analysis:   Dict[str, Any]
        risk_assessment:      Dict[str, Any]
        trade_recommendation: Dict[str, Any]
        memory_context:       str
        final_response:       str
        errors:               List[str]
        banter_log:           List[str]
        ncp_context:          Dict[str, Any]

    class GoTState(_BaseState):
        thought_graph:       Annotated[List[Dict], operator.add]
        adversarial_attacks: Annotated[List[Dict], operator.add]
        got_generation:      int
        got_max_generations: int
        consensus_reached:   bool
        surviving_thoughts:  List[Dict]

    AgentState = _BaseState
else:
    # Standalone dict-based state when LangChain not installed
    AgentState = dict   # type: ignore
    GoTState   = dict   # type: ignore


# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓  SECTION 4: COGNITIVE AGENT ENGINE (Dual-Process)  ▓▓▓
#  Source: hydra_cognitive_agents_v2.py
# ═══════════════════════════════════════════════════════════════════════════
```

```
class CognitiveAgent:
    """
    Enhanced agent with dual-process thinking (Kahneman System 1 / System 2).

    SYSTEM 1 — FAST (system_1_reaction):
      Immediate pattern recognition driven by personality traits.
      Produces BUY/SELL/HOLD/WAIT in milliseconds with heuristic confidence.
      No memory lookup. Pure trait-modulated signal scoring.

    SYSTEM 2 — SLOW (system_2_deliberation):
      Memory retrieval and deliberate reasoning.
      Can OVERRIDE System 1 if traumatic memories contradict the signal.
      Contrarian agents may flip even on strong consensus.

    EVOLUTION ENGINE integration (learn_from_outcome):
      Personality traits adjust based on PnL outcomes.
      Winning setups reinforce aggression; losses elevate fear.
    """

    def __init__(
        self,
        name:        str,
        role:        str,
        personality: AgentPersonality,
        color_code:  str = "",
        description: str = "",
    ):
        self.name             = name
        self.role             = role
        self.personality      = personality
        self.color_code       = color_code
        self.description      = description
        self.decision_history: List[CognitiveState] = []
        self.learning_rate    = 0.1
        self.reputation_score = 1.0    # ARC-P voting weight
        self.wins             = 0
        self.losses           = 0

    # ── System 1: Fast reaction ──────────────────────────────────────────

    def system_1_reaction(
        self, market_context: Dict[str, Any]
    ) -> Tuple[str, float, str]:
        """
        FAST thinking — Immediate pattern recognition based on personality.
        Returns: (decision, confidence, reasoning_text)
        """
```

```
price_change  = market_context.get("price_change_pct", 0)
rsi           = market_context.get("rsi", 50)
volume_spike  = market_context.get("volume_spike", False)
trend         = market_context.get("trend", "NEUTRAL")
atr_pct       = market_context.get("atr_pct", 2.0)
whale_activity= market_context.get("whale_activity", False)
volatility    = market_context.get("volatility", "NORMAL")

score   = 0.0
reasons = []

# ── Aggression: breakout response ──────────────────────────────
if volume_spike and abs(price_change) > 2:
    boost = self.personality.aggression * 2.0
    score += boost if price_change > 0 else -boost
    reasons.append(
        f"Aggression({self.personality.aggression:.1f})→"
        f"{'BUY breakout' if price_change > 0 else 'SELL breakout'}"
    )

# ── Patience: hold back in neutral/unclear trends ──────────────
if self.personality.patience > 0.7 and trend == "NEUTRAL":
    score *= 0.5
    reasons.append(f"Patience({self.personality.patience:.1f})→Wait for t

# ── Greed: chase momentum ──────────────────────────────────────
if self.personality.greed > 0.6 and price_change > 3:
    score += self.personality.greed * 1.5
    reasons.append(f"Greed({self.personality.greed:.1f})→Chase momentum")

# ── Fear: avoid falling knife ──────────────────────────────────
if self.personality.fear > 0.6 and price_change < -2:
    score -= self.personality.fear * 2.0
    reasons.append(f"Fear({self.personality.fear:.1f})→Avoid falling knif

# ── RSI: overbought/oversold modulated by contrarian trait ─────
if rsi > 70:
    overbought = -1.0 * (1.0 - self.personality.contrarian)
    score += overbought
    label = "Fade" if not self.personality.contrarian > 0.6 else "Hold ov
    reasons.append(f"RSI {rsi:.0f} overbought→{label}")
elif rsi < 30:
    oversold = 1.0 * (1.0 - self.personality.contrarian)
    score += oversold
    label = "Buy dip" if not self.personality.contrarian > 0.6 else "Avoi
    reasons.append(f"RSI {rsi:.0f} oversold→{label}")
```

```
        # ── Whale activity: large order flow ──────────────────────────────
        if whale_activity:
            if self.personality.aggression > 0.6:
                score += 0.8
                reasons.append("Whale accumulation detected→Follow smart money")
            elif self.personality.fear > 0.6:
                reasons.append("Whale activity→ambiguous, staying cautious")

        # ── Volatility: DEMON mode ────────────────────────────────────────
        if volatility == "HIGH" and self.personality.aggression > 0.7:
            score += 0.5
            reasons.append("High volatility→aggression thrives in chaos")
        elif volatility == "HIGH" and self.personality.fear > 0.6:
            score -= 0.8
            reasons.append("High volatility→fear override, reduce exposure")

        # ── Convert score → decision ──────────────────────────────────────
        if score > 1.5:
            decision   = "BUY"
            confidence = min(abs(score) * 0.25, 0.95)
        elif score < -1.5:
            decision   = "SELL"
            confidence = min(abs(score) * 0.25, 0.95)
        elif abs(score) > 0.5:
            decision   = "WAIT"
            confidence = 0.4
        else:
            decision   = "HOLD"
            confidence = 0.3

        reasoning = " | ".join(reasons) if reasons else "Neutral — no strong sign
        return decision, confidence, reasoning

    # ── System 2: Slow deliberation ──────────────────────────────────────

    def system_2_deliberation(
        self,
        initial_decision:   str,
        initial_confidence: float,
        memory_system,
        market_context: Dict[str, Any],
    ) -> Tuple[str, float, List[Dict], bool]:
        """
        SLOW thinking — Memory retrieval and deliberate reasoning.
        Returns: (final_decision, final_confidence, memories, override_flag)
        """
        symbol = market_context.get("symbol", "UNKNOWN")
```

```
tags   = [
    symbol,
    initial_decision,
    market_context.get("trend", "NEUTRAL"),
    "BREAKOUT" if market_context.get("volume_spike") else "NORMAL",
]

memories = []
try:
    memories = memory_system.recall(tags, min_relevance=0.3)
except Exception:
    pass

if not memories:
    return initial_decision, initial_confidence, [], False

# ── Analyse memory sentiment ──────────────────────────────────────
sentiment_sum = 0.0
trauma_count  = 0
win_count     = 0

for m in memories:
    mem       = m.get("memory", {})
    sentiment = float(mem.get("sentiment", 0.0))
    relevance = float(m.get("relevance", 0.0))
    sentiment_sum += sentiment * relevance
    if sentiment < -0.5:
        trauma_count += 1
    elif sentiment > 0.5:
        win_count += 1

avg_sentiment  = sentiment_sum / len(memories)
override       = False
final_decision = initial_decision
final_conf     = initial_confidence

# ── Override logic ────────────────────────────────────────────────
if initial_decision == "BUY" and trauma_count >= 2 and avg_sentiment < -0
    override       = True
    final_decision = "WAIT"
    final_conf    *= 0.5

elif initial_decision == "BUY" and win_count >= 2 and avg_sentiment > 0.4
    final_conf = min(final_conf * 1.3, 0.95)

# Fear amplifies trauma
if trauma_count > 0 and self.personality.fear > 0.6:
```

```
            final_conf *= (1.0 - self.personality.fear * 0.3)

        # Contrarian may flip on strong consensus
        if self.personality.contrarian > 0.7 and abs(avg_sentiment) > 0.6:
            flip_prob = self.personality.contrarian * 0.3
            if random.random() < flip_prob:
                override       = True
                final_decision = "WAIT" if final_decision == "BUY" else "BUY"
                final_conf    *= 0.7

        return final_decision, final_conf, memories, override

    # ── Full think cycle: S1 → S2 ──────────────────────────────────────────

    def think(
        self, market_context: Dict[str, Any], memory_system=None
    ) -> CognitiveState:
        """
        Complete dual-process cycle: System 1 (fast) → System 2 (slow).
        Returns a CognitiveState with full reasoning chain and emotional state.
        """

        class _NullMemory:
            def recall(self, *a, **kw): return []

        if memory_system is None:
            memory_system = _NullMemory()

        # System 1
        s1_decision, s1_conf, s1_reason = self.system_1_reaction(market_context)
        chain = [
            f"[S1] {self.name} fast-react: {s1_decision} ({s1_conf:.0%})",
            f"[S1] {s1_reason}",
        ]

        # System 2
        s2_decision, s2_conf, memories, override = self.system_2_deliberation(
            s1_decision, s1_conf, memory_system, market_context
        )

        if memories:
            chain.append(f"[S2] Retrieved {len(memories)} memories")
            for i, m in enumerate(memories[:2], 1):
                mem = m.get("memory", {})
                chain.append(
                    f"  Mem {i}: {str(mem.get('txt',''))[:60]}"
                    f"… (sentiment {mem.get('sentiment',0):+.2f})"
```

```
            )

        if override:
            chain.append(
                f"[S2] OVERRIDE: {s1_decision}→{s2_decision} "
                f"({s1_conf:.0%}→{s2_conf:.0%})"
            )
        else:
            chain.append(f"[S2] Confirmed: {s2_decision} ({s2_conf:.0%})")

        emotional = {
            "fear_level":       self.personality.fear * (1.0 if s2_decision in ("
            "greed_level":      self.personality.greed * (1.0 if s2_decision == "
            "confidence_level": s2_conf,
            "stress_level":     1.0 - self.personality.patience if override else
        }

        state = CognitiveState(
            initial_reaction   = s1_decision,
            retrieved_memories = memories,
            memory_influence   = abs(s2_conf - s1_conf),
            final_decision     = s2_decision,
            confidence         = s2_conf,
            reasoning_chain    = chain,
            emotional_state    = emotional,
            override_occurred  = override,
        )

        self.decision_history.append(state)
        return state

    # ── Reinforcement learning ──────────────────────────────────────

    def learn_from_outcome(self, actual_pnl_pct: float):
        """
        Evolution Engine hook: adjust personality from trade outcome.
        Called after a trade closes with the realised PnL%.
        """
        lr = self.learning_rate
        if actual_pnl_pct > 5:          # Winning trade
            self.wins += 1
            if self.personality.aggression > 0.5:
                self.personality.aggression = min(0.95, self.personality.aggressi
            self.reputation_score = min(2.0, self.reputation_score + 0.05)
        elif actual_pnl_pct < -3:       # Losing trade
            self.losses += 1
            self.personality.fear       = min(0.95, self.personality.fear + lr *
```

```
            self.personality.aggression = max(0.05, self.personality.aggression -
            self.reputation_score = max(0.1, self.reputation_score - 0.1)

    def win_rate(self) -> float:
        total = self.wins + self.losses
        return self.wins / total if total > 0 else 0.5

    def __repr__(self) -> str:
        return (
            f"<Agent:{self.name} role={self.role} "
            f"rep={self.reputation_score:.2f} "
            f"WR={self.win_rate():.0%} "
            f"dominant={self.personality.dominant_trait()}>"
        )


# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓  SECTION 5: 7-AGENT SWARM DEFINITIONS  ▓▓▓
#  Source: hydra_advanced_kimi.pdf (Ultimate v3 architecture)
# ═══════════════════════════════════════════════════════════════════════════

def build_agent_swarm() -> List[CognitiveAgent]:
    """
    Instantiate all seven cognitive agents with calibrated personalities.
    Returns the full swarm ordered by aggression (highest → lowest).
    """
    return [
        CognitiveAgent(
            name="STINKMEANER",
            role="High-Aggression Momentum Hunter",
            color_code="CRIMSON",
            description=(
                "I hunt breakouts. I see momentum and I take it. "
                "Everyone else is too slow. I am never too slow. "
                "GRANDDAD thinks I am reckless. My returns disagree."
            ),
            personality=AgentPersonality(
                aggression=0.90, patience=0.20, greed=0.80,
                fear=0.20, contrarian=0.30,
            ),
        ),
        CognitiveAgent(
            name="VOLATILITY_DEMON",
            role="Volatility Breakout Specialist",
            color_code="ORANGE",
            description=(
                "High volatility is not risk to me — it is signal. "
```

```
            "ATR expansion, Bollinger squeeze release, IV crush reversals. "
            "When the market screams, I listen and I trade."
        ),
        personality=AgentPersonality(
            aggression=0.80, patience=0.30, greed=0.70,
            fear=0.25, contrarian=0.45,
        ),
    ),
    CognitiveAgent(
        name="MOMENTUM_MASTER",
        role="Trend Following Expert",
        color_code="GOLD",
        description=(
            "I ride trends until they bend. EMA crossovers, "
            "higher highs, higher lows. I do not predict. I follow. "
            "The trend is my only opinion."
        ),
        personality=AgentPersonality(
            aggression=0.70, patience=0.60, greed=0.65,
            fear=0.35, contrarian=0.20,
        ),
    ),
    CognitiveAgent(
        name="WHALE_WATCHER",
        role="Large Order Flow Detector",
        color_code="BLUE",
        description=(
            "I watch where the whales accumulate. On-chain flows, "
            "exchange deposits/withdrawals, OI changes. "
            "Follow the smart money. Everything else is noise."
        ),
        personality=AgentPersonality(
            aggression=0.60, patience=0.65, greed=0.60,
            fear=0.40, contrarian=0.55,
        ),
    ),
    CognitiveAgent(
        name="SAMUEL",
        role="Disciplined Trend Strategist",
        color_code="STEEL",
        description=(
            "I demand four confirmations before any directional call: "
            "RSI alignment, MACD crossover, EMA structure, S/R confluence. "
            "Miss one — I say HOLD. STINKMEANER hates me. The P&L loves me."
        ),
        personality=AgentPersonality(
            aggression=0.50, patience=0.80, greed=0.50,
```

```
                fear=0.40, contrarian=0.40,
            ),
        ),
        CognitiveAgent(
            name="SUPPORT_SEEKER",
            role="Support/Resistance Level Trader",
            color_code="GREEN",
            description=(
                "I trade levels. Not guesses — levels. Pivot points, "
                "HTF S/R, psychological round numbers. "
                "Entry at support, exit at resistance. Simple. Deadly effective."
            ),
            personality=AgentPersonality(
                aggression=0.30, patience=0.85, greed=0.40,
                fear=0.55, contrarian=0.35,
            ),
        ),
        CognitiveAgent(
            name="VOICE_OF_REASON",
            role="Conservative Risk Manager",
            color_code="SILVER",
            description=(
                "I am what happens when you survive three crypto winters. "
                "Every trade is sized to survive a 30% drawdown. "
                "I have blocked more trades than I have approved. "
                "I have also never blown up an account."
            ),
            personality=AgentPersonality(
                aggression=0.20, patience=0.90, greed=0.25,
                fear=0.75, contrarian=0.25,
            ),
        ),
    ]


# Build the global swarm at module load
AGENT_SWARM: List[CognitiveAgent] = build_agent_swarm()

# Agent lookup by name
AGENT_MAP: Dict[str, CognitiveAgent] = {a.name: a for a in AGENT_SWARM}

# The core council (original four) for LangGraph compatibility
COUNCIL_AGENTS = ["STINKMEANER", "SAMUEL", "VOICE_OF_REASON", "MOMENTUM_MASTER"]


# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓▓  SECTION 6: ADVERSARIAL CONSENSUS PROTOCOL (ARC-P)  ▓▓▓▓
```

```
#  Source: hydra_advanced_kimi.pdf — Wolfpack detection, Byzantine fault toleranc
# ═══════════════════════════════════════════════════════════════════════════════

class ARCProtocol:
    """
    Adversarial Consensus Protocol v1.

    Implements:
      1. Wolfpack attack detection  — flag when agents form suspicious coalitions
      2. Mean-field aggregation     — weight votes by reputation, not headcount
      3. Byzantine fault tolerance  — discard up to 33% of lowest-rep votes
      4. Reputation-based weighting — winners earn higher vote weight over time

    Byzantine rule: with N agents, tolerate f < N/3 faulty agents.
    A consensus is valid only if it is supported by more than 2N/3 reputation mas
    """

    @staticmethod
    def detect_wolfpack(
        agent_states: Dict[str, CognitiveState],
        threshold:    float = 0.85,
    ) -> bool:
        """
        Wolfpack: suspicious unanimous BUY or SELL agreement.
        All agents converging on the same extreme signal at high confidence
        may indicate data manipulation or correlated bias — flag for review.
        """
        decisions   = [s.final_decision for s in agent_states.values()]
        confidences = [s.confidence      for s in agent_states.values()]

        if not decisions:
            return False

        majority = max(set(decisions), key=decisions.count)
        majority_count = decisions.count(majority)
        avg_conf = sum(confidences) / len(confidences)

        is_wolfpack = (
            majority_count == len(decisions)   # unanimous
            and majority in ("BUY", "SELL")    # directional only
            and avg_conf > threshold            # suspiciously confident
        )
        return is_wolfpack

    @staticmethod
    def mean_field_aggregate(
        agent_states: Dict[str, CognitiveState],
```

```
    agent_map:    Dict[str, CognitiveAgent],
) -> Dict[str, Any]:
    """
    Aggregate decisions weighted by agent reputation scores.
    Returns consensus with vote distribution and Byzantine flags.
    """
    vote_weights:  Dict[str, float] = {"BUY": 0.0, "SELL": 0.0, "HOLD": 0.0,
    total_rep = 0.0
    override_count = 0

    # Sort by reputation to enable Byzantine culling
    sorted_agents = sorted(
        agent_states.items(),
        key=lambda x: agent_map.get(x[0], CognitiveAgent("?","?",
            AgentPersonality(0.5,0.5,0.5,0.5,0.5))).reputation_score,
        reverse=True,
    )

    # Byzantine: keep only the top (1 - BYZANTINE_TOLERANCE) fraction
    n_keep = max(1, int(len(sorted_agents) * (1 - KERNEL.BYZANTINE_TOLERANCE)
    trusted = sorted_agents[:n_keep]
    culled  = len(sorted_agents) - n_keep

    for name, state in trusted:
        agent = agent_map.get(name)
        rep   = agent.reputation_score if agent else 1.0
        vote_weights[state.final_decision] += state.confidence * rep
        total_rep += rep
        if state.override_occurred:
            override_count += 1

    consensus = max(vote_weights, key=vote_weights.get)
    consensus_mass = vote_weights[consensus] / total_rep if total_rep > 0 els

    # Validate: consensus must carry > 2/3 of reputation mass
    valid = consensus_mass > (2 / 3)

    # Confidence: unanimous gets bonus, contested gets penalty
    all_decisions = set(s.final_decision for s in agent_states.values())
    is_unanimous  = len(all_decisions) == 1
    n_trusted     = len(trusted)
    avg_conf      = sum(s.confidence for _, s in trusted) / n_trusted if n_tr

    if is_unanimous:
        final_conf = min(avg_conf * 1.20, 0.95)
    else:
        final_conf = avg_conf * consensus_mass
```

```
        wolfpack = ARCProtocol.detect_wolfpack(
            {n: s for n, s in agent_states.items()}, threshold=0.85
        )

        return {
            "consensus_decision":  consensus,
            "consensus_strength":  consensus_mass,
            "final_confidence":    final_conf,
            "vote_weights":        vote_weights,
            "is_unanimous":        is_unanimous,
            "is_valid_byzantine":  valid,
            "wolfpack_detected":   wolfpack,
            "agents_culled":       culled,
            "override_count":      override_count,
            "trusted_agents":      [n for n, _ in trusted],
        }


# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓  SECTION 7: AGENT COUNCIL (Full Deliberation)  ▓▓▓
#  Combines CognitiveAgent, ARC-P, NCP, and banter into one deliberation unit
# ═══════════════════════════════════════════════════════════════════════════

class AgentCouncil:
    """
    Full council deliberation with:
      - Dual-process cognition per agent
      - Neural Context Protocol integrity checks
      - Adversarial Consensus Protocol (ARC-P) aggregation
      - Byzantine fault tolerance
      - Wolfpack detection
      - HYDRA Raw Mode voice output
      - Inter-agent banter on every decision

    Usage:
        council = AgentCouncil(memory_system)
        result  = council.deliberate(market_context)
        report  = council.explain_decision(result)
    """

    def __init__(self, memory_system=None, use_full_swarm: bool = False):
        self.memory = memory_system
        if use_full_swarm:
            self.agents = AGENT_SWARM
        else:
            # Default to the four core council members
```

```
        self.agents = [AGENT_MAP[n] for n in COUNCIL_AGENTS if n in AGENT_MAP
    self.agent_map = {a.name: a for a in self.agents}

def _build_ncp_context(self, market_context: Dict) -> NCPContext:
    ctx = NCPContext(
        context_id=str(uuid.uuid4())[:8],
        priority=self._classify_priority(market_context),
    )
    ctx.sign(market_context)
    return ctx

@staticmethod
def _classify_priority(ctx: Dict) -> Literal["CRITICAL","HIGH","MEDIUM","LOW"
    change  = abs(ctx.get("price_change_pct", 0))
    spike   = ctx.get("volume_spike", False)
    rsi     = ctx.get("rsi", 50)
    extreme = (rsi > 80 or rsi < 20)
    if change > 8 or (spike and extreme):   return "CRITICAL"
    if change > 4 or spike:                 return "HIGH"
    if change > 2 or extreme:               return "MEDIUM"
    return "LOW"

def deliberate(self, market_context: Dict[str, Any]) -> Dict[str, Any]:
    """
    Full council deliberation process.
    Returns consensus decision with ARC-P metadata and banter log.
    """
    # NCP integrity sign
    ncp = self._build_ncp_context(market_context)
    banter: List[str] = []

    banter.append(f"[HYDRA] {HydraRawMode.on_analysis_start(market_context.ge
    banter.append(f"[NCP] Context {ncp.context_id} | Priority: {ncp.priority}

    # Each agent independently thinks
    agent_states: Dict[str, CognitiveState] = {}
    for agent in self.agents:
        state = agent.think(market_context, self.memory)
        agent_states[agent.name] = state
        banter.append(
            f"[{agent.name}] S1={state.initial_reaction} → "
            f"S2={state.final_decision} ({state.confidence:.0%})"
            + (" [OVERRIDE]" if state.override_occurred else "")
        )

    # ARC-P aggregation
    result = ARCProtocol.mean_field_aggregate(agent_states, self.agent_map)
```

```
        result["agent_states"] = agent_states
        result["ncp_context"]  = ncp
        result["banter_log"]   = banter

        # Wolfpack warning
        if result["wolfpack_detected"]:
            banter.append(
                "[ARC-P] ⚠ WOLFPACK DETECTED — unanimous directional signal "
                "at high confidence. Flagging for human review. Do not auto-trust
            )

        # Byzantine validity
        if not result["is_valid_byzantine"]:
            banter.append(
                "[ARC-P] ⚠ BYZANTINE THRESHOLD NOT MET — consensus carries "
                f"only {result['consensus_strength']:.0%} of reputation mass. "
                "Minimum 2/3 required. Treating as HOLD."
            )
            result["consensus_decision"] = "HOLD"

        # Final HYDRA voice
        if result["consensus_decision"] in ("BUY", "STRONG_BUY"):
            banter.append(f"[HYDRA] {HydraRawMode.on_trade_approved()}")
        elif result["consensus_decision"] in ("SELL", "STRONG_SELL", "AVOID"):
            banter.append(f"[HYDRA] {HydraRawMode.on_trade_blocked()}")
        else:
            banter.append(f"[HYDRA] {HydraRawMode.status()}")

        # Inter-agent trash talk
        banter += self._generate_banter(result, market_context)

        return result

    def _generate_banter(
        self, result: Dict, ctx: Dict
    ) -> List[str]:
        """Authentic inter-agent commentary — no filter."""
        decision = result["consensus_decision"]
        conf     = result["final_confidence"]
        rsi      = ctx.get("rsi", 50)
        lines    = []

        stink = AGENT_MAP.get("STINKMEANER")
        samuel= AGENT_MAP.get("SAMUEL")
        vor   = AGENT_MAP.get("VOICE_OF_REASON")
        vol_d = AGENT_MAP.get("VOLATILITY_DEMON")
```

```
if decision in ("BUY", "STRONG_BUY"):
    lines.append(
        "[STINKMEANER] FINALLY. Momentum is RIGHT THERE. "
        "Council just approved what I said ten minutes ago."
    )
    lines.append(
        f"[SAMUEL] The technicals support it. "
        f"{'RSI is not yet overbought, so momentum has room.' if rsi < 65
        "I am on board. Reluctantly."
    )
    lines.append(
        "[VOICE_OF_REASON] I approved 2% of portfolio. "
        "TWO PERCENT. Not one cent more. "
        "Anyone touches the position size and I'm pulling the whole recom
    )
    lines.append(
        "[STINKMEANER] Bro every single time we have a setup you say two
        "Two. Percent. Do you hear yourself?"
    )
    lines.append("[VOICE_OF_REASON] Yes. Clearly.")
elif decision in ("SELL", "STRONG_SELL"):
    lines.append(
        "[VOICE_OF_REASON] The exit thesis is sound. "
        "Anyone still holding is holding because of hope, not analysis."
    )
    lines.append(
        "[STINKMEANER] Cutting is discipline. I don't like it. "
        "But the chart broke. We out."
    )
    lines.append(
        "[VOLATILITY_DEMON] Volatility spiked the wrong way. "
        "When I say we have the wrong side of a vol expansion, trust me."
    )
elif decision == "HOLD":
    lines.append(
        "[SAMUEL] Signals are conflicted. I require four confirmations. "
        "I have two. That is HOLD territory and I will not apologise for
    )
    lines.append(
        "[STINKMEANER] I am watching this setup crumble while we HOLD. "
        "For the record, I disagree. Loudly."
    )
    lines.append(
        "[VOICE_OF_REASON] HOLD means we do not lose money today. "
        "You cannot lose money you did not bet."
    )
    lines.append(
```

```
                "[STINKMEANER] You also cannot MAKE money you did not bet. But ok
            )
        else:  # WAIT / AVOID
            lines.append(
                "[SAMUEL] The setup is not ready. I will not force a trade "
                "because someone wants action. Wait for the structure."
            )
            lines.append(
                "[STINKMEANER] Fine. I am watching the clock though. "
                "The minute this breaks out I am calling it."
            )
            lines.append(
                "[VOICE_OF_REASON] WAIT is the right call. Market is not "
                "offering a favourable entry. Patience is a position."
            )

        if result.get("wolfpack_detected"):
            lines.append(
                "[WHALE_WATCHER] Everyone agreeing this loudly worries me. "
                "When every agent screams the same thing, someone is setting a tr
            )

        if result.get("agents_culled", 0) > 0:
            lines.append(
                f"[ARC-P] {result['agents_culled']} agent(s) culled from vote — "
                "reputation below Byzantine threshold. Their votes did not count.
            )

        return lines

    def explain_decision(self, result: Dict[str, Any]) -> str:
        """Full human-readable deliberation report."""
        lines = [
            "=" * 75,
            "HYDRA SENTINEL-X — AGENT COUNCIL DELIBERATION",
            "=" * 75,
        ]

        ncp = result.get("ncp_context")
        if ncp:
            lines.append(
                f"NCP Context: {ncp.context_id} | Priority: {ncp.priority} "
                f"| Integrity: {'✓ VERIFIED' if ncp.verified else '✗ FAILED'}"
            )

        lines.append("")
        for name, state in result.get("agent_states", {}).items():
```

```
            agent = self.agent_map.get(name)
            role  = agent.role if agent else "?"
            lines.append(f"┌─ {name} ({role})")
            lines.append(
                f"│  Decision: {state.final_decision} "
                f"({state.confidence:.0%} confidence)"
                + (" ← OVERRIDE" if state.override_occurred else "")
            )
            lines.append(
                f"│  Emotional: Fear={state.emotional_state.get('fear_level',0):.
                f"Greed={state.emotional_state.get('greed_level',0):.2f} "
                f"Stress={state.emotional_state.get('stress_level',0):.2f}"
            )
            for r in state.reasoning_chain[:3]:
                lines.append(f"│    • {r}")
            lines.append("└" + "─" * 50)

        lines += [
            "",
            "=" * 75,
            "ARC-P CONSENSUS RESULT",
            "=" * 75,
            f"Decision:          {result['consensus_decision']}",
            f"Consensus Strength:{result['consensus_strength']:.0%}",
            f"Final Confidence:  {result['final_confidence']:.0%}",
            f"Unanimous:         {'Yes' if result.get('is_unanimous') else 'No'}"
            f"Byzantine Valid:   {'Yes' if result.get('is_valid_byzantine') else
            f"Wolfpack Detected: {'⚠ YES — REVIEW REQUIRED' if result.get('wolfpa
            f"Overrides:        {result.get('override_count',0)}/{len(self.agent
            "",
            "Vote Distribution (reputation-weighted):",
        ]
        for dec, weight in result.get("vote_weights", {}).items():
            lines.append(f"  {dec}: {weight:.3f}")

        lines.append("")
        lines.append("Agent Banter:")
        for b in result.get("banter_log", []):
            lines.append(f"  {b}")

        return "\n".join(lines)


# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓  SECTION 8: MULTIMODAL PERCEPTION ENGINE  ▓▓▓
#  50+ technical indicators, pattern recognition, divergence detection
# ═══════════════════════════════════════════════════════════════════════════
```

```
class MultimodalPerceptionEngine:
    """
    Computes a comprehensive technical picture from OHLCV data.
    No external API calls — pure computation from price arrays.
    Feed this output into the council's market_context.
    """

    @staticmethod
    def compute_rsi(prices: List[float], period: int = 14) -> float:
        if len(prices) < period + 1:
            return 50.0
        deltas = [prices[i] - prices[i-1] for i in range(1, len(prices))]
        gains  = [max(d, 0) for d in deltas[-period:]]
        losses = [abs(min(d, 0)) for d in deltas[-period:]]
        avg_g  = sum(gains)  / period
        avg_l  = sum(losses) / period
        if avg_l == 0:
            return 100.0
        rs = avg_g / avg_l
        return round(100 - (100 / (1 + rs)), 2)

    @staticmethod
    def compute_ema(prices: List[float], period: int) -> List[float]:
        if not prices:
            return []
        k      = 2 / (period + 1)
        ema    = [prices[0]]
        for p in prices[1:]:
            ema.append(p * k + ema[-1] * (1 - k))
        return ema

    @staticmethod
    def compute_macd(
        prices: List[float], fast: int = 12, slow: int = 26, signal: int = 9
    ) -> Dict[str, float]:
        if len(prices) < slow:
            return {"macd": 0.0, "signal": 0.0, "histogram": 0.0}
        ema_fast = MultimodalPerceptionEngine.compute_ema(prices, fast)
        ema_slow = MultimodalPerceptionEngine.compute_ema(prices, slow)
        min_len  = min(len(ema_fast), len(ema_slow))
        macd_line = [ema_fast[i] - ema_slow[i] for i in range(min_len)]
        signal_line = MultimodalPerceptionEngine.compute_ema(macd_line, signal)
        hist = macd_line[-1] - signal_line[-1] if signal_line else 0.0
        return {
            "macd":      round(macd_line[-1], 6),
            "signal":    round(signal_line[-1], 6),
```

```
            "histogram": round(hist, 6),
        }

    @staticmethod
    def compute_bollinger(
        prices: List[float], period: int = 20, std_mult: float = 2.0
    ) -> Dict[str, float]:
        if len(prices) < period:
            p = prices[-1] if prices else 0
            return {"upper": p, "middle": p, "lower": p, "bandwidth": 0.0}
        window = prices[-period:]
        mid    = sum(window) / period
        std    = math.sqrt(sum((x - mid)**2 for x in window) / period)
        upper  = mid + std_mult * std
        lower  = mid - std_mult * std
        bw     = (upper - lower) / mid if mid else 0
        return {
            "upper":     round(upper, 4),
            "middle":    round(mid, 4),
            "lower":     round(lower, 4),
            "bandwidth": round(bw, 6),
        }

    @staticmethod
    def compute_atr(
        highs: List[float], lows: List[float], closes: List[float], period: int =
    ) -> float:
        if len(closes) < 2:
            return 0.0
        trs = []
        for i in range(1, min(len(highs), len(lows), len(closes))):
            tr = max(
                highs[i] - lows[i],
                abs(highs[i] - closes[i-1]),
                abs(lows[i]  - closes[i-1]),
            )
            trs.append(tr)
        window = trs[-period:]
        return round(sum(window) / len(window), 6) if window else 0.0

    @staticmethod
    def detect_candlestick_pattern(
        open_: float, high: float, low: float, close: float, prev_close: float
    ) -> str:
        body   = abs(close - open_)
        range_ = high - low
        if range_ == 0:
```

```
            return "FLAT"
        body_ratio  = body / range_
        upper_wick  = (high - max(open_, close)) / range_
        lower_wick  = (min(open_, close) - low)  / range_

        if body_ratio < 0.1:
            return "DOJI"
        if body_ratio > 0.8:
            return "MARUBOZU_BULL" if close > open_ else "MARUBOZU_BEAR"
        if lower_wick > 0.6 and body_ratio < 0.3:
            return "HAMMER"
        if upper_wick > 0.6 and body_ratio < 0.3:
            return "SHOOTING_STAR"
        bullish_engulf = close > open_ and close > prev_close
        if bullish_engulf and body_ratio > 0.6:
            return "BULLISH_ENGULFING"
        bearish_engulf = close < open_ and close < prev_close
        if bearish_engulf and body_ratio > 0.6:
            return "BEARISH_ENGULFING"
        return "STANDARD"

    @staticmethod
    def detect_divergence(
        prices: List[float], rsi_values: List[float]
    ) -> Optional[str]:
        """Detect RSI divergence over last N candles."""
        n = min(len(prices), len(rsi_values), 14)
        if n < 8:
            return None
        p_recent = prices[-n:]
        r_recent = rsi_values[-n:]

        price_higher = p_recent[-1] > max(p_recent[:-1])
        price_lower  = p_recent[-1] < min(p_recent[:-1])
        rsi_higher   = r_recent[-1] > max(r_recent[:-1])
        rsi_lower    = r_recent[-1] < min(r_recent[:-1])

        if price_higher and rsi_lower:
            return "BEARISH_DIVERGENCE"
        if price_lower and rsi_higher:
            return "BULLISH_DIVERGENCE"
        return None

    @classmethod
    def full_analysis(
        cls,
        closes: List[float],
```

```
    highs:  List[float] = None,
    lows:   List[float] = None,
    volumes:List[float] = None,
) -> Dict[str, Any]:
    """
    Run all indicators and return a structured analysis dict
    suitable as market_context for the AgentCouncil.
    """
    if not closes:
        return {"error": "No price data provided"}

    highs   = highs   or closes
    lows    = lows    or closes
    volumes = volumes or [1.0] * len(closes)

    price   = closes[-1]
    rsi     = cls.compute_rsi(closes)
    macd    = cls.compute_macd(closes)
    bb      = cls.compute_bollinger(closes)
    ema20   = cls.compute_ema(closes, 20)[-1] if len(closes) >= 20 else price
    ema50   = cls.compute_ema(closes, 50)[-1] if len(closes) >= 50 else price
    ema200  = cls.compute_ema(closes, 200)[-1] if len(closes) >= 200 else Non
    atr     = cls.compute_atr(highs, lows, closes)
    atr_pct = (atr / price * 100) if price else 0

    # Trend from EMA alignment
    if ema200:
        if ema20 > ema50 > ema200:   trend = "BULLISH"
        elif ema20 < ema50 < ema200: trend = "BEARISH"
        else:                         trend = "NEUTRAL"
    else:
        trend = "BULLISH" if ema20 > ema50 else "BEARISH" if ema20 < ema50 el

    # Bollinger position
    if price > bb["upper"] * 0.99:      bb_pos = "NEAR_UPPER"
    elif price < bb["lower"] * 1.01:    bb_pos = "NEAR_LOWER"
    elif bb["bandwidth"] < 0.02:        bb_pos = "SQUEEZE"
    else:                               bb_pos = "MIDDLE"

    # Volume spike
    if len(volumes) >= 20:
        avg_vol    = sum(volumes[-20:]) / 20
        vol_spike  = volumes[-1] > avg_vol * 1.5
    else:
        vol_spike = False

    # Candlestick
```

```
if len(closes) >= 2:
    pattern = cls.detect_candlestick_pattern(
        closes[-2], highs[-1], lows[-1], closes[-1], closes[-2]
    )
else:
    pattern = "UNKNOWN"

# Divergence
rsi_series = [cls.compute_rsi(closes[:i+14]) for i in range(len(closes)-1
divergence = cls.detect_divergence(closes, rsi_series)

# Support / Resistance (simple pivot method)
window = closes[-30:] if len(closes) >= 30 else closes
support    = round(min(window), 4)
resistance = round(max(window), 4)

# MACD signal
hist = macd["histogram"]
if hist > 0 and macd["macd"] > macd["signal"]:
    macd_signal = "BUY"
elif hist < 0 and macd["macd"] < macd["signal"]:
    macd_signal = "SELL"
else:
    macd_signal = "NEUTRAL"

return {
    "price":         price,
    "rsi":           rsi,
    "rsi_signal":    "overbought" if rsi > 70 else "oversold" if rsi < 30
    "macd":          macd,
    "macd_signal":   macd_signal,
    "bollinger":     bb,
    "bb_position":   bb_pos,
    "ema20":         round(ema20, 4),
    "ema50":         round(ema50, 4),
    "ema200":        round(ema200, 4) if ema200 else None,
    "trend":         trend,
    "atr":           atr,
    "atr_pct":       round(atr_pct, 3),
    "volume_spike":  vol_spike,
    "pattern":       pattern,
    "divergence":    divergence,
    "support":       support,
    "resistance":    resistance,
    "volatility":    "HIGH" if atr_pct > 5 else "LOW" if atr_pct < 1 else
}
```

```
# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓▓  SECTION 9: GRAPH OF THOUGHTS LAYER (GoT)  ▓▓▓▓
#  STINKMEANER / SAMUEL generate → GRANDDAD critiques → CLAYTON+JULIUS aggregate
#  Pure reasoning over already-fetched data. No external calls.
# ═══════════════════════════════════════════════════════════════════════════

SURVIVAL_THRESHOLD = KERNEL.GOT_SURVIVAL_THRESHOLD

# ── GoT: Generate node ──────────────────────────────────────────────────────

STINKMEANER_SYSTEM_PROMPT = """\
You are STINKMEANER — aggressive momentum hunter. Bold. Conviction-driven. Offens
Generate ONE complete trading hypothesis in JSON.
Be specific: cite RSI values, price levels, volume data. No vague claims.

Output ONLY JSON in ```json ... ```:
{
  "thesis": "one-sentence directional call",
  "direction": "long" | "short" | "neutral",
  "timeframe": "1d" | "3d" | "1w" | "2w",
  "entry_rationale": "specific technical/momentum reason",
  "key_evidence": ["point 1", "point 2", "point 3"],
  "proposed_entry": float,
  "proposed_target": float,
  "proposed_stop": float,
  "primary_risk": "the ONE thing that invalidates this",
  "confidence_score": float
}
"""

SAMUEL_SYSTEM_PROMPT = """\
You are SAMUEL — cold, disciplined trend analyst. Precise. Systematic. Patient.
Require confirmation before any call. A HOLD thesis is valid — better than a forc
Cite RSI, MACD crossovers, EMA positions, S/R confluences.

Output ONLY JSON in ```json ... ```:
{
  "thesis": "one-sentence directional call",
  "direction": "long" | "short" | "neutral",
  "timeframe": "1d" | "3d" | "1w" | "2w",
  "entry_rationale": "specific technical reason",
  "key_evidence": ["point 1", "point 2", "point 3"],
  "proposed_entry": float,
  "proposed_target": float,
  "proposed_stop": float,
  "primary_risk": "the ONE thing that invalidates this",
```

```
  "confidence_score": float
}
"""

GRANDDAD_SYSTEM_PROMPT = """\
You are GRANDDAD — most battle-scarred risk sceptic alive.
You survived 2008, the 2018 crypto winter, Luna, FTX. You trust NOTHING.

Apply ALL FOUR attacks:
  1. FLASH_CRASH     — price drops 15-25% in 60 minutes
  2. VOLUME_SPOOF    — all volume fabricated; evaporates on entry
  3. MACRO_SHOCK     — Fed/exchange/depeg event causes 3-5% move against position
  4. LIQUIDITY_DRAIN — stop-loss cannot fill; actual exit 3-8% worse

For each: does the thesis survive? By how much does confidence drop?

Output ONLY a JSON array in ```json ... ```:
[
  {
    "attack_type": "flash_crash" | "volume_spoof" | "macro_shock" | "liquidity_dr
    "scenario": "specific description for THIS thesis",
    "confidence_delta": float,
    "thesis_survives": true | false,
    "explanation": "why it does or does not survive"
  }
]
"""

CLAYTON_JULIUS_SYSTEM_PROMPT = """\
You are CLAYTON (defensive risk officer) + JULIUS (pragmatic execution closer).
Take surviving hypotheses and merge them into ONE unified, conservative recommend

Rules:
1. ENTRY ZONE: span from conservative to aggressive entry across hypotheses
2. TARGET: average targets, reduce by 10% (CLAYTON's influence)
3. STOP: take tighter stop, tighten further by 5%
4. CONFIDENCE: weighted average of post-attack scores
5. Never exceed Risk Manager's approved position size

Output ONLY JSON in ```json ... ```:
{
  "action": "strong_buy" | "buy" | "hold" | "sell" | "strong_sell" | "avoid",
  "entry_zone": {"low": float, "high": float},
  "target_price": float,
  "stop_loss": float,
  "timeframe": "1d" | "3d" | "1w" | "2w",
  "confidence": float,
```

```
  "strategy_used": string,
  "rationale": "4-6 sentences citing both hypotheses and survived attacks",
  "key_risks": ["risk 1", "risk 2", "risk 3"],
  "dca_plan": "string or empty"
}
"""


def _extract_json(content: str) -> Any:
    match = re.search(r"```json\s*(.*?)\s*```", content, re.DOTALL)
    if match:
        try:
            return json.loads(match.group(1))
        except Exception:
            pass
    try:
        return json.loads(content)
    except Exception:
        return {}


class GraphOfThoughts:
    """
    GoT reasoning layer. Operates on data already fetched by specialist agents.
    Three-node cycle: Generate → Adversarial Critique → Aggregate.
    Loops until survivors found or max_generations reached.
    """

    def __init__(self, llm=None):
        self.llm = llm
        if llm is None and HAS_LANGCHAIN:
            try:
                self.llm = ChatAnthropic(
                    model="claude-sonnet-4-6", temperature=0.2, max_tokens=4096
                )
            except Exception:
                self.llm = None

    def _build_data_summary(self, context: Dict, generation: int) -> str:
        md  = context.get("market_data", {})
        ta  = context.get("technical_analysis", {})
        ra  = context.get("risk_assessment", {})

        prior = ""
        if generation > 1:
            pruned = context.get("pruned_reasons", [])
            if pruned:
```

```
                prior = (
                    f"\n\n⚠ RETRY GEN {generation} — Prior hypotheses pruned.\n"
                    + "\n".join(f"  - {r}" for r in pruned[:4])
                    + "\nGenerate a FUNDAMENTALLY DIFFERENT thesis."
                )

        return f"""SPECIALIST ANALYSIS SUMMARY{prior}

MARKET:
  Price:        ${md.get('price_usd',0):,.4f}
  24h Change:   {md.get('price_change_24h',0):+.2f}%
  Sentiment:    {md.get('sentiment','unknown')}
  Fear & Greed: {md.get('fear_greed_index','?')}/100
  Headlines:    {'; '.join((md.get('news_headlines') or [])[:3])}

TECHNICAL:
  RSI(14):      {ta.get('rsi','?')}
  MACD:         {ta.get('macd_signal','?')}
  EMA Trend:    {ta.get('ema_trend','?')}
  BB Position:  {ta.get('bb_position','?')}
  Support:      {ta.get('support_levels',[])}
  Resistance:   {ta.get('resistance_levels',[])}
  Pattern:      {ta.get('pattern','?')}
  Divergence:   {ta.get('divergence','none')}

RISK:
  Risk Level:   {ra.get('risk_level','?')}
  Max Position: ${ra.get('max_position_size_usd',0):,.0f}
  Stop-Loss:    -{ra.get('stop_loss_pct',0):.1f}%
  R/R Ratio:    {ra.get('risk_reward_ratio',0):.2f}×

PORTFOLIO:
  Value:        ${context.get('portfolio_value',10000):,.0f}
  Tolerance:    {context.get('risk_tolerance','moderate')}
  Query:        {context.get('user_query','')}
"""

    def _llm_call(self, system: str, user: str) -> str:
        if self.llm is None:
            return '```json\n{"error": "LLM not available"}\n```'
        try:
            if HAS_LANGCHAIN:
                resp = self.llm.invoke([
                    SystemMessage(content=system),
                    HumanMessage(content=user),
                ])
                return resp.content if hasattr(resp, "content") else str(resp)
```

```
        except Exception as e:
            return f'{{"error": "{e}"}}'
        return '{"error": "unknown"}'

    def generate(self, context: Dict, generation: int = 1) -> List[Dict]:
        """Generate two hypotheses in parallel (STINKMEANER + SAMUEL)."""
        data = self._build_data_summary(context, generation)

        thoughts = []
        for persona, prompt in [
            ("STINKMEANER", STINKMEANER_SYSTEM_PROMPT),
            ("SAMUEL",      SAMUEL_SYSTEM_PROMPT),
        ]:
            raw     = self._llm_call(prompt, f"Generate hypothesis:\n\n{data}")
            parsed  = _extract_json(raw)
            conf    = float(parsed.get("confidence_score", 0.65)) if isinstance(p
            thoughts.append({
                "node_id":           str(uuid.uuid4()),
                "agent_origin":      persona,
                "content":           parsed if isinstance(parsed, dict) else {"ra
                "confidence_score":  conf,
                "generation":        generation,
                "critique_notes":    [],
                "survived_critique": False,
            })
        return thoughts

    def critique(self, thoughts: List[Dict], context: Dict) -> Tuple[List[Dict],
        """GRANDDAD stress-tests every thought. Returns (updated_thoughts, all_at
        updated  = []
        attacks  = []

        for t in thoughts:
            c      = t["content"]
            thesis = (
                f"Thesis: {c.get('thesis','?')}\n"
                f"Direction: {c.get('direction','?')}\n"
                f"Entry: ${c.get('proposed_entry',0):,.4f}\n"
                f"Target: ${c.get('proposed_target',0):,.4f}\n"
                f"Stop: ${c.get('proposed_stop',0):,.4f}\n"
                f"Evidence: {'; '.join(c.get('key_evidence',[]))}\n"
                f"Confidence: {t['confidence_score']:.0%}"
            )
            raw     = self._llm_call(GRANDDAD_SYSTEM_PROMPT, f"Attack this hypoth
            att_raw = _extract_json(raw)
            if not isinstance(att_raw, list):
                att_raw = [att_raw] if isinstance(att_raw, dict) else []
```

```
        total_delta = sum(float(a.get("confidence_delta", 0)) for a in att_ra
        post_conf   = max(0.0, min(1.0, t["confidence_score"] + total_delta))

        notes = [
            f"[{a.get('attack_type','?').upper()}] "
            f"{'✓' if a.get('thesis_survives') else '✗'} — "
            f"{a.get('explanation','')[:100]}"
            for a in att_raw
        ]

        for a in att_raw:
            attacks.append({
                "attack_id":       str(uuid.uuid4()),
                "target_node_id":  t["node_id"],
                "attack_type":     a.get("attack_type", "unknown"),
                "scenario":        a.get("scenario", ""),
                "confidence_delta":float(a.get("confidence_delta", 0)),
                "thesis_survives": bool(a.get("thesis_survives", False)),
                "explanation":     a.get("explanation", ""),
            })

        updated.append({
            **t,
            "confidence_score":  post_conf,
            "critique_notes":    notes,
            "survived_critique": True,
            "passed":            post_conf > SURVIVAL_THRESHOLD,
        })

    return updated, attacks

def aggregate(
    self, survivors: List[Dict], all_attacks: List[Dict], risk: Dict
) -> Dict:
    """CLAYTON + JULIUS merge surviving hypotheses into final recommendation.
    if not survivors:
        return {
            "action":       "hold",
            "confidence":   0.1,
            "strategy_used":"adversarial_pruning",
            "rationale":    (
                "All hypotheses failed adversarial stress testing. "
                "No directional recommendation possible. HOLD until condition
            ),
            "key_risks": ["All theses pruned by GRANDDAD", "Market conditions
            "entry_zone": {"low": 0, "high": 0},
```

```
        "target_price": 0, "stop_loss": 0,
    }

summaries = []
for t in survivors:
    c   = t["content"]
    atk = [a for a in all_attacks if a["target_node_id"] == t["node_id"]]
    survived_n = sum(1 for a in atk if a["thesis_survives"])
    summaries.append(
        f"Origin: {t['agent_origin']} | Confidence: {t['confidence_score'
        f"Survived {survived_n}/{len(atk)} attacks\n"
        f"Thesis: {c.get('thesis','?')}\n"
        f"Direction: {c.get('direction','?')} | Entry: ${c.get('proposed_
        f"Target: ${c.get('proposed_target',0):,.4f} | Stop: ${c.get('pro
        f"Critique:\n" + "\n".join(t.get("critique_notes",[]))
    )

prompt = (
    f"Synthesise {len(survivors)} surviving hypothesis/hypotheses.\n\n"
    + "\n\n".join(summaries)
    + f"\n\nRisk constraints:\n"
    f"  Max position: ${risk.get('max_position_size_usd',2000):,.0f}\n"
    f"  Risk level: {risk.get('risk_level','medium')}"
)

raw    = self._llm_call(CLAYTON_JULIUS_SYSTEM_PROMPT, prompt)
parsed = _extract_json(raw)

if not isinstance(parsed, dict) or not parsed.get("action"):
    # Fallback: use best survivor
    best = max(survivors, key=lambda t: t["confidence_score"])
    c    = best["content"]
    parsed = {
        "action":       "buy" if c.get("direction") == "long" else "sell"
        "entry_zone":   {"low": c.get("proposed_entry",0)*0.99, "high": c
        "target_price": c.get("proposed_target", 0),
        "stop_loss":    c.get("proposed_stop", 0),
        "confidence":   best["confidence_score"],
        "strategy_used":"got_fallback",
        "rationale":    f"Fallback to best survivor ({best['agent_origin'
        "key_risks":    [],
    }

# Kernel enforcement
if "entry_zone" in parsed:
    parsed["__kernel_cap"] = KERNEL.report()
```

```
        return parsed

    def run(self, context: Dict) -> Dict:
        """
        Full GoT run: Generate → Critique → Aggregate (with retry loop).
        Returns final recommendation dict.
        """
        max_gen = KERNEL.GOT_MAX_GENERATIONS
        all_thoughts: List[Dict] = []
        all_attacks:  List[Dict] = []
        survivors:    List[Dict] = []

        for gen in range(1, max_gen + 1):
            thoughts          = self.generate(context, gen)
            updated, attacks  = self.critique(thoughts, context)
            all_thoughts     += updated
            all_attacks      += attacks
            survivors         = [t for t in updated if t.get("passed")]

            if survivors:
                break

            # No survivors — inject pruning reasons for retry
            reasons = [note for t in updated for note in t.get("critique_notes",[
            context = {**context, "pruned_reasons": reasons}

        ra  = context.get("risk_assessment", {})
        rec = self.aggregate(survivors, all_attacks, ra)

        # Audit enrichment
        attacks_survived = sum(1 for a in all_attacks if a["thesis_survives"])
        rec["got_audit"] = {
            "generations":        gen,
            "hypotheses_total":   len(all_thoughts),
            "survivors":          len(survivors),
            "attacks_total":      len(all_attacks),
            "attacks_survived":   attacks_survived,
            "survival_rate_pct":  round(len(survivors)/max(len(all_thoughts),1)*1
        }
        rec["rationale"] = (
            f"[GoT: {gen} gen | {len(survivors)}/{len(all_thoughts)} survived "
            f"| {attacks_survived}/{len(all_attacks)} attacks passed] "
            + rec.get("rationale", "")
        )
        return rec
```

```
# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓▓  SECTION 10: LANGRAPH SPECIALIST NODES  ▓▓▓▓
#  Market Analyst · Technical Analyst · Risk Manager · Strategy Advisor · Orchest
# ═══════════════════════════════════════════════════════════════════════════

if HAS_LANGCHAIN:

    def _get_llm(model: str = "claude-sonnet-4-6", temp: float = 0.1) -> ChatAnth
        return ChatAnthropic(model=model, temperature=temp, max_tokens=4096)

    def _llm_retry(llm, messages, agent: str = "", retries: int = 3):
        for attempt in range(retries):
            try:
                return llm.invoke(messages)
            except Exception as e:
                if attempt == retries - 1:
                    raise
                time.sleep(2 ** attempt)

    def _react_loop(llm, messages, tools, max_iter=6, agent=""):
        tool_map = {t.name: t for t in tools}
        for _ in range(max_iter):
            resp = _llm_retry(llm, messages, agent)
            messages.append(resp)
            calls = getattr(resp, "tool_calls", [])
            if not calls:
                break
            for tc in calls:
                fn = tool_map.get(tc["name"])
                try:
                    result = fn.invoke(tc["args"]) if fn else f"Tool not found: {
                except Exception as e:
                    result = f"Tool error: {e}"
                messages.append(ToolMessage(content=str(result), tool_call_id=tc[
        return resp, messages

    # ── Market Analyst ──────────────────────────────────────────────────────

    _MARKET_SYSTEM = """\
You are a crypto MARKET ANALYST. Fetch price data, sentiment, news ONLY.
Do NOT compute technicals or recommend trades.

Steps: (1) get_crypto_price  (2) get_fear_greed_index  (3) synthesise

Output JSON in ```json ... ```:
{"price_usd":float,"price_change_24h":float,"market_cap":float,"volume_24h":float
 "sentiment":"bullish"|"bearish"|"neutral","fear_greed_index":int,
```

```
 "news_headlines":[str],"key_observations":[str]}
"""
    def market_analyst_node(state: dict) -> dict:
        t   = time.perf_counter()
        llm = _get_llm().bind_tools(MARKET_TOOLS) if MARKET_TOOLS else _get_llm()
        mem = state.get("memory_context", "No prior history.")
        messages = [
            SystemMessage(content=_MARKET_SYSTEM),
            HumanMessage(content=(
                f"Analyse {state.get('target_symbol','BTC')}. "
                f"Portfolio: ${state.get('portfolio_value',10000):,.0f}.\n"
                f"Query: {state.get('user_query','')}\nMEMORY:\n{mem}"
            )),
        ]
        try:
            if MARKET_TOOLS:
                resp, messages = _react_loop(llm, messages, MARKET_TOOLS, 5, "mar
            else:
                resp = _llm_retry(llm, messages, "market_analyst")
                messages.append(resp)
            md = {"symbol": state.get("target_symbol","BTC"), **_extract_json(res
        except Exception as e:
            md = {"symbol": state.get("target_symbol","BTC"), "error": str(e)}
        ms = round((time.perf_counter()-t)*1000)
        return {
            "market_data":  md,
            "current_step": "market_analysis_complete",
            "messages":     messages[2:],
            "banter_log": [
                f"[MARKET_ANALYST] {state.get('target_symbol','?')} at "
                f"${md.get('price_usd',0):,.2f}. "
                f"Sentiment: {md.get('sentiment','?').upper()}. "
                f"F&G: {md.get('fear_greed_index','?')}/100. Done in {ms}ms.",
                f"[TECHNICAL_ANALYST] Great. A number between 1 and 100. "
                f"Very useful. My RSI will tell you something actionable.",
            ],
        }

    # ── Technical Analyst ──────────────────────────────────────────────

    _TECHNICAL_SYSTEM = """\
You are a TECHNICAL ANALYST. Compute indicators ONLY. No news. No position sizing

Toolkit: RSI(14), MACD(12/26/9), Bollinger(20,2), EMA(20/50/200), S/R pivots.
Steps: (1) get_historical_prices  (2) calculate_technical_indicators  (3) S/R

Output JSON in ```json ... ```:
```

```
{"trend":"uptrend"|"downtrend"|"ranging","rsi":float,
 "macd_signal":"buy"|"sell"|"neutral","bb_position":"near_upper"|"near_lower"|"mi
 "ema20":float,"ema50":float,"ema200":float|null,"ema_alignment":"bullish"|"beari
 "support_levels":[float,float],"resistance_levels":[float,float],"pattern":str|n
"""
    def technical_analyst_node(state: dict) -> dict:
        t   = time.perf_counter()
        llm = _get_llm().bind_tools(TECHNICAL_TOOLS) if TECHNICAL_TOOLS else _get
        mem = state.get("memory_context", "No prior history.")
        messages = [
            SystemMessage(content=_TECHNICAL_SYSTEM),
            HumanMessage(content=(
                f"Technical analysis for {state.get('target_symbol','BTC')}.\n"
                f"Query: {state.get('user_query','')}\nMEMORY:\n{mem}"
            )),
        ]
        try:
            if TECHNICAL_TOOLS:
                resp, messages = _react_loop(llm, messages, TECHNICAL_TOOLS, 6, "
            else:
                resp = _llm_retry(llm, messages, "technical_analyst")
                messages.append(resp)
            ta = {"symbol": state.get("target_symbol","BTC"), **_extract_json(res
        except Exception as e:
            ta = {"symbol": state.get("target_symbol","BTC"), "error": str(e)}
        ms = round((time.perf_counter()-t)*1000)
        rsi = ta.get("rsi", 50)
        return {
            "technical_analysis": ta,
            "current_step":       "technical_analysis_complete",
            "messages":           messages[2:],
            "banter_log": [
                f"[TECHNICAL_ANALYST] Chart says: trend={ta.get('trend','?')} "
                f"RSI={rsi:.0f} MACD={ta.get('macd_signal','?')} "
                f"BB={ta.get('bb_position','?')}. Done in {ms}ms.",
                f"[STINKMEANER] {'RSI oversold — PRIME entry, let me IN' if rsi <
                f"[SAMUEL] {'Confirmed. RSI and MACD aligned. I will accept this
            ],
        }

    # ── Risk Manager ──────────────────────────────────────────────────────

    _RISK_SYSTEM = f"""\
You are a RISK MANAGER. Capital preservation FIRST.
Governance kernel HARD CAPS (cannot override):
  - Max position: {KERNEL.MAX_POSITION_PCT*100:.0f}% of portfolio
  - Min R:R: {KERNEL.MIN_RISK_REWARD}:1
```

```
  - Spot only. No leverage.

Steps: (1) calculate_position_size  (2) assess_portfolio_risk

Output JSON in ```json ... ```:
{{"risk_level":"low"|"medium"|"high"|"extreme","max_position_size_usd":float,
 "max_position_size_pct":float,"stop_loss_pct":float,"stop_loss_price":float,
 "take_profit_pct":float,"take_profit_price":float,"risk_reward_ratio":float,
 "kelly_fraction":float,"warnings":[str]}}
"""
    def risk_manager_node(state: dict) -> dict:
        t    = time.perf_counter()
        tools= RISK_TOOLS + CDP_READONLY_TOOLS if CDP_ENABLED else RISK_TOOLS
        llm  = _get_llm().bind_tools(tools) if tools else _get_llm()
        md   = state.get("market_data",        {})
        ta   = state.get("technical_analysis", {})
        messages = [
            SystemMessage(content=_RISK_SYSTEM),
            HumanMessage(content=(
                f"Risk for {state.get('target_symbol','BTC')}.\n"
                f"Portfolio: ${state.get('portfolio_value',10000):,.0f} "
                f"| Tolerance: {state.get('risk_tolerance','moderate')}\n"
                f"Price: ${md.get('price_usd','?')} | RSI: {ta.get('rsi','?')} "
                f"| Trend: {ta.get('trend','?')}\n"
                f"Support: {ta.get('support_levels',[])} | Resistance: {ta.get('r
            )),
        ]
        try:
            if tools:
                resp, messages = _react_loop(llm, messages, tools, 4, "risk_manag
            else:
                resp = _llm_retry(llm, messages, "risk_manager")
                messages.append(resp)
            ra = {"portfolio_value": state.get("portfolio_value",10000), **_extra
            # Kernel enforcement
            if ra.get("max_position_size_usd"):
                clipped = KERNEL.clip_position(ra["max_position_size_usd"], state
                if ra["max_position_size_usd"] > clipped:
                    ra["max_position_size_usd"] = clipped
                    ra.setdefault("warnings",[]).append(f"KERNEL: Position clippe
        except Exception as e:
            ra = {"portfolio_value": state.get("portfolio_value",10000), "error":
        ms   = round((time.perf_counter()-t)*1000)
        rr   = ra.get("risk_reward_ratio", 0)
        pos  = ra.get("max_position_size_usd", 0)
        return {
            "risk_assessment": ra,
```

```
            "current_step":    "risk_assessment_complete",
            "messages":        messages[2:],
            "banter_log": [
                f"[RISK_MANAGER] Position approved at ${pos:,.0f}. "
                f"R:R={rr:.2f}× "
                f"{'PASSES 2:1 minimum.' if rr >= 2 else 'FAILS 2:1. Blocking tra
                f"Done in {ms}ms.",
                f"[STINKMEANER] Two percent of portfolio. Always two percent. "
                f"I should not be allowed to hear this anymore.",
                f"[VOICE_OF_REASON] Good. Two percent is exactly right. "
                f"I am proud of this system.",
                f"[STINKMEANER] Of course you are.",
                f"[GRANDDAD] {'The R:R is acceptable. Barely.' if rr >= 2 else 'R
            ],
        }

    # ── Strategy Advisor ──────────────────────────────────────────────

    _STRATEGY_SYSTEM = """\
You synthesise ALL specialist inputs into ONE final recommendation.
Do NOT gather data. Make THE DECISION.
Verdicts: strong_buy | buy | hold | sell | strong_sell | avoid
If signals conflict → HOLD. Never force a trade.

Output JSON in ```json ... ```:
{"action":str,"strategy_used":str,"entry_zone":{"low":float,"high":float},
 "target_price":float,"stop_loss":float,"timeframe":str,"confidence":float,
 "rationale":str,"key_risks":[str],"dca_plan":str|null}
"""
    def strategy_advisor_node(state: dict) -> dict:
        t   = time.perf_counter()
        llm = _get_llm().bind_tools(STRATEGY_TOOLS) if STRATEGY_TOOLS else _get_l
        md  = state.get("market_data",         {})
        ta  = state.get("technical_analysis",  {})
        ra  = state.get("risk_assessment",     {})
        messages = [
            SystemMessage(content=_STRATEGY_SYSTEM),
            HumanMessage(content=(
                f"Strategy for {state.get('target_symbol','BTC')}.\n"
                f"Portfolio: ${state.get('portfolio_value',10000):,.0f} "
                f"| Tolerance: {state.get('risk_tolerance','moderate')}\n"
                f"Query: {state.get('user_query','')}\n\n"
                f"MARKET:\n{json.dumps(md,indent=2)}\n\n"
                f"TECHNICALS:\n{json.dumps(ta,indent=2)}\n\n"
                f"RISK:\n{json.dumps(ra,indent=2)}"
            )),
        ]
```

```
        try:
            if STRATEGY_TOOLS:
                resp, messages = _react_loop(llm, messages, STRATEGY_TOOLS, 4, "s
            else:
                resp = _llm_retry(llm, messages, "strategy_advisor")
                messages.append(resp)
            rec = _extract_json(resp.content)
            if not isinstance(rec, dict) or not rec.get("action"):
                rec = {"action":"hold","confidence":0.5,"rationale":str(resp.cont
        except Exception as e:
            rec = {"action":"hold","confidence":0.1,"rationale":f"Error: {e}"}
        ms     = round((time.perf_counter()-t)*1000)
        action = rec.get("action","hold")
        conf   = rec.get("confidence",0.5)
        return {
            "trade_recommendation": rec,
            "current_step":         "strategy_complete",
            "messages":             messages[2:],
            "banter_log": [
                f"[STRATEGY_ADVISOR] Verdict: {action.upper().replace('_',' ')} (
                f"[HYDRA] {HydraRawMode.on_trade_approved() if 'buy' in action el
                f"[STINKMEANER] {'I approve this message.' if 'buy' in action els
                f"[SAMUEL] {'Technical consensus confirmed.' if 'buy' in action e
                f"[GRANDDAD] Remember: this still needs human approval. No autono
            ],
        }

    # ── Orchestrator ──────────────────────────────────────────────────────

    _PARSE_SYSTEM = """\
Routing orchestrator for crypto trading AI. Parse the query → JSON only.
Output in ```json ... ```:
{"target_symbol":"BTC","active_agents":["market_analyst","technical_analyst",
"risk_manager","strategy_advisor"],"portfolio_value":null,"risk_tolerance":"moder
"""
    _SYNTH_SYSTEM = """\
Master synthesiser. Produce a structured trading analysis report.
Sections: Market Snapshot · Technical Picture · Risk Parameters · Final Recommend
· GoT Reasoning Audit (if present) · Key Risks & Disclaimer.
Be precise. Always include the disclaimer.
"""

    _calibration_context: str = ""
    def set_calibration(ctx: str): globals()["_calibration_context"] = ctx

    def orchestrator_parse_node(state: dict) -> dict:
        llm = ChatAnthropic(model="claude-haiku-4-5-20251001", temperature=0, max
```

```
    messages = [
        SystemMessage(content=_PARSE_SYSTEM),
        HumanMessage(content=(
            f"Query: {state.get('user_query','')}\n"
            f"Memory: {state.get('memory_context','None')}"
        )),
    ]
    try:
        resp   = llm.invoke(messages)
        parsed = _extract_json(resp.content)
    except Exception:
        parsed = {}

    valid = {"market_analyst","technical_analyst","risk_manager","strategy_ad
    agents = [a for a in parsed.get("active_agents", list(valid)) if a in val

    return {
        "target_symbol": parsed.get("target_symbol","BTC").upper(),
        "active_agents": agents,
        "current_step":  "routing_complete",
        "iteration":     state.get("iteration",0) + 1,
        "banter_log": [
            f"[HYDRA] {HydraRawMode.on_analysis_start(parsed.get('target_symb
            f"[ORCHESTRATOR] Routing to: {agents}",
        ],
    }

def orchestrator_synthesise_node(state: dict) -> dict:
    llm = ChatAnthropic(model="claude-opus-4-6", temperature=0.2, max_tokens=
    md  = state.get("market_data",          {})
    ta  = state.get("technical_analysis",   {})
    ra  = state.get("risk_assessment",      {})
    rec = state.get("trade_recommendation", {})
    messages = [
        SystemMessage(content=_SYNTH_SYSTEM),
        HumanMessage(content=(
            f"Query: {state.get('user_query','')}\n"
            f"Symbol: {state.get('target_symbol','BTC')} "
            f"| Portfolio: ${state.get('portfolio_value',0):,.0f}\n\n"
            f"MARKET:\n{json.dumps(md,indent=2)}\n\n"
            f"TECHNICAL:\n{json.dumps(ta,indent=2)}\n\n"
            f"RISK:\n{json.dumps(ra,indent=2)}\n\n"
            f"RECOMMENDATION:\n{json.dumps(rec,indent=2)}"
        )),
    ]
    resp = _llm_retry(llm, messages, "synthesiser")
    return {
```

```
            "final_response": resp.content,
            "current_step":   "synthesis_complete",
        }


# ═══════════════════════════════════════════════════════════════════════
#  ▓▓▓▓  SECTION 11: EVOLUTION ENGINE  ▓▓▓▓
#  Strategy mutation · Self-evaluation · Reinforcement learning loop
# ═══════════════════════════════════════════════════════════════════════

@dataclass
class TradeOutcome:
    """Records the result of a closed trade for Evolution Engine learning."""
    recommendation_id: str
    symbol:            str
    action:            str
    entry_price:       float
    exit_price:        float
    pnl_pct:           float
    agent_states:      Dict[str, CognitiveState] = field(default_factory=dict)
    timestamp:         str = field(default_factory=lambda: datetime.now(timezone.


class EvolutionEngine:
    """
    Closes the learning loop: trade outcomes → agent personality updates.

    Flow:
      1. Trade closes → record TradeOutcome
      2. Engine distributes outcome to every CognitiveAgent that voted
      3. Agents update personality traits via learn_from_outcome()
      4. Reputation scores adjust via ARC-P win/loss tracking
      5. Strategy registry records which setup patterns work over time

    The engine also runs self-evaluation: it periodically compares each
    agent's win rate against its reputation score and flags inconsistencies
    (e.g. high reputation but dropping win rate → reputation decay).
    """

    def __init__(self, swarm: List[CognitiveAgent] = None):
        self.swarm          = swarm or AGENT_SWARM
        self.outcomes:       List[TradeOutcome] = []
        self.strategy_stats: Dict[str, Dict]    = {}

    def record_outcome(self, outcome: TradeOutcome):
        """Feed a closed trade back to all agents that were involved."""
        self.outcomes.append(outcome)
```

```
        # Update all agents
        for agent in self.swarm:
            agent.learn_from_outcome(outcome.pnl_pct)

        # Strategy registry
        strat = outcome.action
        if strat not in self.strategy_stats:
            self.strategy_stats[strat] = {"wins": 0, "losses": 0, "total_pnl": 0.
        if outcome.pnl_pct > 0:
            self.strategy_stats[strat]["wins"] += 1
        else:
            self.strategy_stats[strat]["losses"] += 1
        self.strategy_stats[strat]["total_pnl"] += outcome.pnl_pct

    def self_evaluate(self) -> str:
        """
        Compare agent win rates against reputation scores.
        Flag drift. Return a diagnostic report.
        """
        lines = ["[EVOLUTION ENGINE] Self-Evaluation Report", "=" * 50]
        for agent in self.swarm:
            wr  = agent.win_rate()
            rep = agent.reputation_score
            drift = abs(wr - (rep / 2.0))   # expect rep ≈ 2×win_rate
            flag  = "⚠ DRIFT" if drift > 0.2 else "✓ OK"
            lines.append(
                f"  {agent.name:20s} WR={wr:.0%} Rep={rep:.2f} Drift={drift:.2f}
            )
        lines.append("")
        lines.append("Strategy Performance:")
        for strat, stats in self.strategy_stats.items():
            total  = stats["wins"] + stats["losses"]
            rate   = stats["wins"] / total if total > 0 else 0
            avg    = stats["total_pnl"] / total if total > 0 else 0
            lines.append(
                f"  {strat:20s} WR={rate:.0%} AvgPnL={avg:+.1f}% N={total}"
            )
        return "\n".join(lines)

    def mutate_strategy(self, agent_name: str, direction: str = "auto") -> str:
        """
        Sandboxed mutation: adjust one agent's personality based on recent perfor
        direction: 'aggressive' | 'conservative' | 'auto'
        """
        agent = AGENT_MAP.get(agent_name)
        if not agent:
```

```
            return f"Agent {agent_name} not found."

        wr = agent.win_rate()
        if direction == "auto":
            direction = "aggressive" if wr > 0.6 else "conservative"

        p = agent.personality
        if direction == "aggressive":
            p.aggression = min(0.95, p.aggression + 0.05)
            p.patience   = max(0.05, p.patience   - 0.03)
            return f"[EVOLUTION] {agent_name} → MORE AGGRESSIVE (WR={wr:.0%})"
        else:
            p.aggression = max(0.05, p.aggression - 0.05)
            p.fear       = min(0.95, p.fear       + 0.03)
            return f"[EVOLUTION] {agent_name} → MORE CONSERVATIVE (WR={wr:.0%})"


# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓▓  SECTION 12: SYSTEM STATUS + CAPABILITIES REPORT  ▓▓▓▓
# ═══════════════════════════════════════════════════════════════════════════

def full_system_status(stats: dict = None) -> str:
    """
    Complete system status report with agent commentary.
    Pass in live stats dict for real numbers.
    """
    stats = stats or {}
    pending  = stats.get("pending", 0)
    approved = stats.get("approved", 0)
    rejected = stats.get("rejected", 0)
    win_rate = stats.get("win_rate", 0.0)

    swarm_lines = []
    for a in AGENT_SWARM:
        swarm_lines.append(
            f"  {a.color_code:8s} | {a.name:20s} | {a.role:38s} | "
            f"Rep={a.reputation_score:.2f} | "
            f"Agg={a.personality.aggression:.1f} "
            f"Pat={a.personality.patience:.1f} "
            f"Grd={a.personality.greed:.1f} "
            f"Fear={a.personality.fear:.1f}"
        )

    return f"""
╔═══════════════════════════════════════════════════════════════════════════╗
║   HYDRA SENTINEL-X CORPORATION — INTELLIGENCE SYSTEM STATUS v3.0           ║
║   Authority: President & C.E.O. Daryell McFarland                         ║
```

╚═══╝

{HydraRawMode.greeting()}

── LIVE QUEUE ──
  Pending: {pending} | Approved: {approved} | Rejected: {rejected}
  Win Rate on Executed Trades: {win_rate:.1f}%

── GOVERNANCE KERNEL (IMMUTABLE) ──
  {KERNEL.report()}

── 7-AGENT COGNITIVE SWARM ───
  COLOR | NAME | ROLE | MET
{'─'*95}
{chr(10).join(swarm_lines)}

── DUAL-PROCESS COGNITION ──
  [SYSTEM 1 — FAST]
    Immediate pattern recognition driven by personality traits.
    Computes decision score from: aggression, patience, greed, fear, contrarian.
    RSI response, volume spike reaction, momentum chasing — all personality-gated
    Latency: <10ms (pure Python, no LLM call).

  [SYSTEM 2 — SLOW]
    Memory retrieval + deliberate reasoning.
    Reads past trade history from vector memory (Qdrant).
    Can OVERRIDE System 1 if traumatic memories contradict the signal.
    Contrarian agents may flip even on strong consensus (with probability).
    Latency: ~200ms (memory lookup + scoring).

── ADVERSARIAL CONSENSUS PROTOCOL (ARC-P) ──
  Wolfpack Detection: Flag suspicious unanimous directional signals (>85% conf)
  Byzantine Tolerance: Cull lowest-reputation {KERNEL.BYZANTINE_TOLERANCE*100:.0
  Reputation Voting: Vote weight = confidence × reputation_score.
  Validity Gate: Consensus must carry >2/3 of total reputation mass.
  Fallback: If Byzantine gate fails → auto-HOLD.

── NEURAL CONTEXT PROTOCOL (NCP) ───
  Every market_context is SHA-256 signed before being passed to the swarm.
  Priority levels: CRITICAL (>8% move) · HIGH (>4%) · MEDIUM (>2%) · LOW.
  Tamper detection: agents receive verified hash; mismatch = context rejected.

── GRAPH OF THOUGHTS LAYER ───
  STINKMEANER + SAMUEL generate competing hypotheses in parallel (~5s).
  GRANDDAD applies 4 adversarial attacks per hypothesis:
    • FLASH_CRASH (15-25% instant drop)
    • VOLUME_SPOOF (fake volume evaporation)

• MACRO_SHOCK (Fed/exchange event)
• LIQUIDITY_DRAIN (stop-loss slippage)
Survival threshold: >{KERNEL.GOT_SURVIVAL_THRESHOLD*100:.0f}% confidence post-a
CLAYTON + JULIUS aggregate survivors into conservative final recommendation.
Max retry generations: {KERNEL.GOT_MAX_GENERATIONS} (then → auto-HOLD).

── MULTIMODAL PERCEPTION ENGINE ──

Pure-Python technical indicators (no API key required):
RSI(14), MACD(12/26/9), Bollinger Bands(20,2), EMA(20/50/200),
ATR(14), Support/Resistance pivots, Candlestick pattern recognition,
RSI divergence detection (bullish/bearish), Volume spike detection.

── EVOLUTION ENGINE ──

Every closed trade feeds back to agents via learn_from_outcome().
Winning trades: boost aggression if dominant trait.
Losing trades: elevate fear, reduce aggression.
Reputation adjustments affect ARC-P vote weight in next deliberation.
Strategy registry: tracks win rate and avg PnL per strategy type.
Sandboxed mutation: auto-adjust aggression/patience based on rolling WR.

── HYDRA RAW MODE PERSONALITY ──

Tone: {HydraRawMode.TONE}
Filter: {HydraRawMode.FILTER.upper()}
Style: {HydraRawMode.STYLE}
All agent broadcasts and system announcements use Raw Mode voice.

── CAPABILITIES SUMMARY ──

✓ Live price feeds via Binance WebSocket (no API key)
✓ CoinGecko API — prices, market cap, Fear & Greed
✓ Real Anthropic LLM calls (Haiku for routing, Sonnet for analysis, Opus for sy
✓ LangGraph multi-agent orchestration with streaming
✓ SQLite approval audit trail — immutable, tamper-evident
✓ MEV-aware execution quotes via CDP SDK
✓ Qdrant vector memory for cross-session recall
✓ Prometheus + Grafana observability
✓ Human-in-the-loop approval dashboard (4-hour TTL)
✓ HYDRA Governance Token — on-chain parameter voting (Base Mainnet)

── WHAT EVERY AGENT SAYS ABOUT IMPROVEMENTS ──────────────────────────────

[STINKMEANER]
"Give me Binance L2 order book depth. I can see the bid walls forming
before the breakout happens. Right now I am flying blind on execution.
Also raise the position cap. Please. PLEASE."
[KERNEL overrides the last request. Request denied. As always.]

[SAMUEL]

"On-chain flow data would cut my uncertainty by 40%.
Exchange deposit/withdrawal ratios, UTXO age bands, miner flows.
The Graph API is free. There is no excuse not to have it."

[WHALE_WATCHER]
"I need wallet clustering. I can see LARGE moves but I cannot tell
if it is one whale repositioning or fifty retail traders panic-buying.
The difference is everything."

[VOLATILITY_DEMON]
"IV data. Options markets price in the volatility regime before
spot markets react. If I had Deribit IV data I would call regime
shifts 6-12 hours earlier. That is real edge."

[GRANDDAD]
"Nothing. Add nothing. More signals = more manipulation surface.
The four attacks I run are sufficient. If anything, I want a fifth:
REGULATORY_SHOCK. Because it will happen again."

[VOICE_OF_REASON]
"Automated rebalancing alerts. I flag concentration risk but I
cannot act on it without human approval. That delay costs alpha.
A push notification to Telegram when concentration exceeds 5%
would close this gap immediately."

[MOMENTUM_MASTER]
"Regime detection. Bull market vs bear market vs sideways.
My trend-following strategy is built for trending markets.
In ranging markets I am generating noise. If the system could
auto-disable me in ranging regimes, overall accuracy would improve."

NEXT STEPS (Strategy Advisor consensus)

1. Deploy to VPS (8GB RAM, 4 CPU, Ubuntu 22.04) — run ./scripts/deploy.sh
2. Set ANTHROPIC_API_KEY and COINGECKO_API_KEY
3. Connect Binance WebSocket for live ticker + L2 order book
4. Deploy HYDRA token to Base Sepolia for governance testing
5. Integrate The Graph for on-chain whale flow data
6. Add Telegram bot for mobile approval notifications
7. Run paper trades for 30 days — validate win rate before live capital
8. Enable Evolution Engine feedback loop after 20+ closed trades

FINAL WORD FROM GRANDDAD

"The system works. Start on paper. Validate the win rate.
The only thing more dangerous than a bad system is a confident
system executing bad trades quickly. Slow down. Verify. THEN trust it.
And for the last time — watch the liquidity."

```
{HydraRawMode.motivate()}
"""


# ═══════════════════════════════════════════════════════════════════════════
#  ▓▓▓▓  ENTRY POINT — Standalone demo  ▓▓▓▓
# ═══════════════════════════════════════════════════════════════════════════

if __name__ == "__main__":
    import pprint

    print("=" * 80)
    print("  HYDRA SENTINEL-X — INTELLIGENCE MODULE v3.0 — STANDALONE DEMO")
    print("=" * 80)
    print(KERNEL.report())
    print()

    # ── Build a market context ──────────────────────────────────────────
    market_ctx = {
        "symbol":          "BTC/USD",
        "price":           65000.0,
        "price_change_pct": 3.5,
        "rsi":             68.0,
        "volume_spike":    True,
        "trend":           "BULLISH",
        "atr_pct":         2.8,
        "whale_activity":  False,
        "volatility":      "NORMAL",
    }

    # ── Mock memory ─────────────────────────────────────────────────────
    class MockMemory:
        def recall(self, tags, min_relevance=0.3):
            return [
                {"relevance": 0.8, "memory": {
                    "txt": "BTC fake breakout → -8% in 2h, stopped out hard",
                    "sentiment": -0.9, "tags": tags,
                }},
                {"relevance": 0.6, "memory": {
                    "txt": "Similar volume spike preceded +12% bull run",
                    "sentiment": 0.7, "tags": tags,
                }},
            ]

    # ── Run council deliberation ────────────────────────────────────────
    council = AgentCouncil(memory_system=MockMemory(), use_full_swarm=True)
    result  = council.deliberate(market_ctx)
```

```
report  = council.explain_decision(result)

print(report)

print()
print(f"✅ EXECUTE: {result['consensus_decision']} "
      f"with {result['final_confidence']:.0%} confidence")
print(f"   Byzantine valid: {result['is_valid_byzantine']}")
print(f"   Wolfpack:        {result['wolfpack_detected']}")
print()

# ── Evolution Engine demo ──────────────────────────────────────────
engine = EvolutionEngine()
engine.record_outcome(TradeOutcome(
    recommendation_id="demo-001",
    symbol="BTC",
    action="buy",
    entry_price=63000,
    exit_price=68000,
    pnl_pct=7.9,
))
print(engine.self_evaluate())
print()

# ── Multimodal Perception Engine demo ──────────────────────────────
import math as _m
prices = [50000 + 1000*_m.sin(i/5) for i in range(60)]
analysis = MultimodalPerceptionEngine.full_analysis(prices)
print("Perception Engine output:")
pprint.pprint({k: v for k, v in analysis.items() if k != "bollinger"})
print()

# ── System status ──────────────────────────────────────────────────
print(full_system_status({"pending": 2, "approved": 14, "rejected": 3, "win_r
```