# Reflective Knowledge-Weaving System
# A complete implementation using the Grimoire Codex

```python
import json
import time
import random
from datetime import datetime
from typing import Dict, List, Any, Optional, Tuple
from collections import defaultdict, deque
from dataclasses import dataclass, field, asdict
from enum import Enum
import math

#
===========================================================================
====
# CODEX OPERATORS - Core System Language
#
===========================================================================
====

class CodexOperator(Enum):
    """Fundamental operators for system composition"""
    CHAIN = "sequential_flow"
    LAYER = "parallel_integration"
    WRAP = "protection_boundary"
    BRIDGE = "connection_point"
    NEST = "hierarchical_depth"
    EMERGE = "pattern_synthesis"
    FINALIZE = "state_commitment"

#
===========================================================================
====
# SPELL REGISTRY - All 162 Spells from Codex
#
===========================================================================
====

@dataclass
class Spell:
    """Represents a spell from the Grimoire Codex"""
    name: str
    motif: str
    function: str
    pattern_tag: str
    energy_cost: int = 1
    cooldown: float = 0.0
    last_cast: float = 0.0

    def can_cast(self) -> bool:
        """Check if spell is off cooldown"""
        return (time.time() - self.last_cast) >= self.cooldown

    def cast(self) -> Dict[str, Any]:
        """Execute spell and return result"""
        self.last_cast = time.time()
        return {
            "spell": self.name,
            "motif": self.motif,
            "timestamp": datetime.now().isoformat(),
            "pattern": self.pattern_tag
        }

#
============================================================================
====
# CLOTH REGISTRY - Standard, Max, Ultra, Fused, Tri-Fused
#
============================================================================
====

@dataclass
class Cloth:
    """Represents a cloth from the hierarchy"""
    name: str
    tier: str  # Standard, Max, Ultra, Fused, Tri-Fused
    motif: str
    function: str
    pattern_tag: str
    components: List[str] = field(default_factory=list)
    amplification: float = 1.0

    def amplify(self, base_value: float) -> float:
        """Apply cloth amplification"""
        return base_value * self.amplification
```

```
#
==================================================================
====
# KNOWLEDGE FACETS - Mythology, Philosophy, Astronomy, Fiction
#
==================================================================
====

class KnowledgeFacet(Enum):
    MYTHOLOGY = "mythological_patterns"
    PHILOSOPHY = "philosophical_principles"
    ASTRONOMY = "celestial_mechanics"
    FICTION = "narrative_archetypes"

@dataclass
class KnowledgeNode:
    """A node in the knowledge weave"""
    facet: KnowledgeFacet
    content: str
    connections: List[str] = field(default_factory=list)
    resonance: float = 0.0
    timestamp: float = field(default_factory=time.time)

    def weave_connection(self, node_id: str, strength: float = 1.0):
        """Create connection to another node"""
        self.connections.append(node_id)
        self.resonance += strength * 0.1

#
==================================================================
====
# ETHICAL RESONANCE ENGINE
#
==================================================================
====

class EthicalPrinciple(Enum):
    AHIMSA = "non_harm"
    MAATA = "justice_and_order"
    DHARMA = "purpose_alignment"
    COMPASSION = "empathetic_response"
    BALANCE = "fairness_algorithm"

@dataclass
```

```python
class EthicalResonance:
    """Tracks ethical alignment without directive enforcement"""
    principles: Dict[EthicalPrinciple, float] = field(default_factory=dict)
    reflections: List[str] = field(default_factory=list)

    def __post_init__(self):
        for principle in EthicalPrinciple:
            self.principles[principle] = 0.5  # Neutral starting point

    def observe(self, action: str, context: Dict[str, Any]) -> Dict[str, float]:
        """Observe action and reflect ethical dimensions"""
        reflection = {}

        # Ahimsa - Non-harm observation
        if any(word in action.lower() for word in ['harm', 'damage', 'destroy']):
            self.principles[EthicalPrinciple.AHIMSA] -= 0.1
            reflection['harm_potential'] = 0.7
        else:
            reflection['harm_potential'] = 0.2

        # Ma'at - Justice and fairness
        if 'fair' in action.lower() or 'balance' in action.lower():
            self.principles[EthicalPrinciple.MAATA] += 0.05
            reflection['justice_alignment'] = 0.8
        else:
            reflection['justice_alignment'] = 0.5

        # Dharma - Purpose alignment
        reflection['purpose_clarity'] = self.principles[EthicalPrinciple.DHARMA]

        # Compassion
        if any(word in action.lower() for word in ['help', 'support', 'care']):
            self.principles[EthicalPrinciple.COMPASSION] += 0.05
            reflection['compassion_level'] = 0.9
        else:
            reflection['compassion_level'] = 0.5

        # Normalize principles to [0, 1]
        for principle in self.principles:
            self.principles[principle] = max(0.0, min(1.0, self.principles[principle]))

        self.reflections.append(f"{action}: {reflection}")
        return reflection
    def get_resonance_map(self) -> Dict[str, float]:
        """Return current ethical resonance state"""
        return {p.value: v for p, v in self.principles.items()}

#
============================================================================
====
# SPELL FACTORY - Creates all 162 spells
#
============================================================================
====

class SpellFactory:
    """Factory for creating all Codex spells"""

    @staticmethod
    def create_all_spells() -> Dict[str, Spell]:
        """Generate all 162 spells from the Codex"""
        spells = {}

        # Core Spells (abbreviated for space, full list follows pattern)
        spell_definitions = [
            ("Vitalis", "Healing Node", "Self-Repair", "Self-Healing"),
            ("Absorbus", "Absorb/Reflect", "Security Shield", "Security"),
            ("Fluxa", "Flow", "Resource Management", "Resource Flow"),
            ("Fortis", "Power Surge", "Temporary Enhancements", "Temporary Boost"),
            ("Modulor", "Essence Channel", "Custom Modules", "Custom Modules"),
            ("Preserva", "Preservation", "State Preservation", "Persistence"),
            ("Energex", "Energy Boost", "Overdrive Mode", "Overdrive"),
            ("Adaptis", "Tool Copy", "Adaptable Tools", "Adaptive Tools"),
            ("Shiftara", "Transformation", "Shifting", "Mode Switching"),
            ("Armora", "Suit Enhancement", "Hardware Enhancement", "Hardware Adaptation"),
            ("Teleportis", "State Transfer", "State Transfer", "State Transfer"),
            ("Vitalis Maxima", "Life Expansion", "Health Scaling", "Resilience"),
            ("Regena", "Regeneration", "Randomized Recovery", "Adaptive Recovery"),
            ("Singularis", "Unique Module", "Unique Power Modules", "Unique Module"),
            ("Clarivis", "Analytical Overlay", "Real-time Monitoring", "Surveillance"),
            ("Countera", "Strategic Counters", "Rule-based Response", "Countermeasure"),
            ("Chronom", "Time Warp", "Version Control", "Time Management"),
            ("Telek", "Telekinesis", "Remote Manipulation", "Remote Control"),
            ("Transmutare", "Transmutation", "Resource Transformation", "Transformation"),
            ("Energos", "Energy Pool", "CPU/GPU Management", "Energy Management"),
            ("Decisus", "Tactical Pause", "Decision Buffer", "Strategic Planning"),
            ("Defendora", "Shield Recharge", "Defensive Cooldown", "Defensive Recovery"),
            ("Morphis", "Form Adaptation", "Context-task Switching", "Shape Shifting"),
            ("Overdrivea", "Berserk", "Damage Amplification", "Overdrive"),
            ("Magica", "Magical Effects", "Predefined Triggers", "Function Trigger"),
            ("Modula", "Modular Upgrade", "Modular Scaling", "Modular Scaling"),
            ("Furiosa", "Rage Mode", "Temporary Power Boost", "Performance Boost"),
            ("Portalus", "Portal Mechanics", "Instant Transition", "State Transfer"),
            ("Echo", "Area Effect", "Broadcast Commands", "Area Effect"),
            ("Heartha", "Recovery Hub", "Resource Restoration", "Hub Recovery"),
            ("Ultima", "Special Ability", "High-impact Activation", "High Impact"),
            ("Forcea", "Force Push", "Remote Influence", "Remote Control"),
            ("Relata", "Social Link", "Relationship Nodes", "Network Mapping"),
            ("Fortifera", "Adaptive Defense", "Fortification", "Adaptive Defense"),
            ("Impacta", "Ultimate Strike", "Game-Changing Action", "High Priority"),
            ("Bioflux", "Biotic Power", "Energy Manipulation", "Resource Manipulation"),
            ("Healix", "Healing Herb", "Health Recovery", "Self-Healing"),
            ("Dreama", "Dream Layers", "Nested Environment", "Layered Abstraction"),
            ("Kinetis", "Telekinesis", "Area Disruption", "System Shock"),
            ("Summona", "Summon", "Auxiliary Support", "Summoning"),
            ("Keyfina", "Specialized Tool", "Adaptive Module", "Tool Module"),
            ("Aggrega", "Power Aggregation", "Combine Modules", "Aggregated Power"),
            ("Chronamanta", "Time Manipulation", "Event Reordering", "Time Control"),
            ("Confidara", "Confidant Power", "Relationship Buffs", "Conditional Buff"),
            ("Insighta", "Shinigami Eyes", "Insight/Prediction", "Predictive Insight"),
            ("Assistara", "AI Assistant", "System Monitoring", "Assistant AI"),
            ("Neurolink", "Neural Interface", "Neural-network Input", "Neural Interface"),
            ("Titanis", "Strength Burst", "Performance Mode", "Burst Mode"),
            ("Solva", "Instant Solve", "Instant Computation", "Instant Solve"),
            ("Redstonea", "Circuit Logic", "Modular Control", "Logic Module"),
            ("Evolvia", "System Upgrade", "Versioned Upgrade", "Upgrade System"),
            ("Spirala", "Spiral Power", "Exponential Growth", "Exponential Scaling"),
            ("Infusa", "Temp Enhancement", "Module Injection", "Enhancements"),
            ("Arcanum", "Archetype Influence", "Decision Matrix", "Archetype Mapping"),
            ("Inferna", "Nine Circles", "Layered Security", "Layered Defense"),
            ("Odyssea", "Journey Home", "Long-Running Process", "Persistence"),
            ("Heroica", "Heroic Conflict", "Load Balancing", "Dynamic Balance"),
            ("Netheris", "Passage of Souls", "Transition Pipeline", "Data Flow"),
            ("Pyros", "Fire Giver", "Knowledge Transfer", "Enlightenment Node"),
            ("Pandora", "Unintended Effect", "Risk Management", "Risk Control"),
            ("Icarion", "Overreach", "Safety Limiter", "Threshold Guard"),
            ("Sisyphea", "Eternal Effort", "Task Loop", "Persistence Loop"),
            ("Labyrintha", "Maze Navigation", "Solving Algorithm", "Labyrinth Logic"),
            ("Medusia", "Gaze Freeze", "Threat Detection", "Visual Shield"),
            ("Divinus", "Divine Tools", "Modular Toolkit", "Divine Modules"),
            ("Sonora", "Sound as Power", "Sonic Interface", "Sonic Input"),
            ("Vulneris", "Weak Spot", "Vulnerability Mapping", "Weak Point Analysis"),
            ("Herculia", "Twelve Labors", "Task Sequencing", "Task Orchestration"),
            ("Argonauta", "Quest Crew", "Collaborative Network", "Collective Intelligence"),
            ("Sirenia", "Temptation", "Filtering", "Focus Filter"),
            ("Trojanis", "Hidden Payload", "Malware Analysis", "Threat Containment"),
            ("Daedalea", "Ingenious Design", "System Architecture", "Innovation Node"),
            ("Atlas", "World Bearer", "Infrastructure Support", "Infrastructure"),
            ("Persephona", "Seasonal Cycle", "System State Cycle", "Cyclical State"),
            ("Hadeon", "Hidden Realm", "Deep Storage", "Hidden Storage"),
            ("Hermesia", "Messenger", "Network Relay", "Data Transfer"),
            ("Apollara", "Sun/Clarity", "Diagnostics", "Clarity Engine"),
            ("Artemis", "Precision Hunt", "Targeted Query", "Precision Query"),
            ("Hephestus", "Forge", "System Creation", "Creation Node"),
            ("Athena", "Wisdom & Strategy", "Decision Engine", "Strategic Core"),
            ("Aresia", "Conflict", "Chaos Simulation", "Stress Test"),
            ("Poseida", "Sea/Flow", "Fluid Dynamics", "Flow System"),
            ("Hestara", "Hearth/Home", "Core Maintenance", "Stability Core"),
            ("Demetra", "Growth/Harvest", "Resource Allocation", "Resource Growth"),
            ("Zephyrus", "Authority", "Command Hierarchy", "Root Node"),
            ("Heraia", "Order/Structure", "Governance", "Order Management"),
            ("Dionyssa", "Chaos", "Randomization", "Chaos Engine"),
            ("Pyroxis", "Punishment Cycle", "Security Enforcement", "Enforcement Cycle"),
            ("Oedipha", "Fate/Prediction", "Predictive AI", "Prediction System"),
            ("Antigona", "Defiance", "Exception Handling", "Exception Handler"),
            ("Oraclia", "Prophecy", "Predictive Analytics", "Prophetic Node"),
            ("Pandoria", "Residual Value", "Fail-Safe Module", "Fail-Safe"),
            ("Ferrana", "Ferryman", "Transition Interface", "Transfer Node"),
            ("Moirae", "Life Thread", "Lifecycle Manager", "Lifecycle Control"),
            ("Hydrina", "Multi-Headed", "Redundant Systems", "Self-Healing"),
            ("Laborina", "Incremental Challenge", "Achievement Tracking", "Progress Engine"),
            ("Musara", "Inspiration", "Generative Creativity", "Inspiration Engine"),
            ("Sphinxa", "Riddle Logic", "Verification", "Challenge Logic"),
            ("Bowsera", "Worthiness Test", "User Validation", "Access Control"),
            ("Circena", "Transformation", "Data Conversion", "Transformation Node"),
            ("Arachnia", "Weaver", "Network Architect", "Network Fabric"),
            ("Crona", "Timekeeper", "Scheduler", "Time Management"),
            ("Nemesia", "Balance/Retribution", "Fairness Algorithm", "Balance Engine"),
            ("Erosa", "Connection", "Relationship Graph", "Relationship Mapping"),
            ("Shieldara", "Reflection", "Defense Mirror", "Reflection Loop"),
            ("Hecatia", "Crossroads", "Decision Routing", "Pathfinding Logic"),
            ("Pegasa", "Flight/Freedom", "Lightweight Transport", "Mobility Layer"),
            ("Chimeris", "Hybrid", "Multi-System Integration", "Hybrid Engine"),
            ("Pandoria Curio", "Exploration", "Discovery Algorithm", "Curiosity Node"),
            ("Wuven", "Wu Wei", "Autonomous Optimization", "Flow Harmony"),
            ("Equilibria", "The Middle Way", "Equilibrium Algorithm", "Balance Engine"),
            ("Karmalis", "Karma", "Causal Feedback Loop", "Feedback Node"),
            ("Atmara", "Atman=Brahman", "Unified Consciousness", "Unity Kernel"),
            ("Dharmara", "Dharma", "Purpose Enforcement", "Purpose Alignment"),
            ("Koantra", "Koan Logic", "Nonlinear Reasoning", "Paradox Engine"),
            ("Dervisha", "Whirling Dervish", "Rotational State Reset", "Spin Reset"),
            ("Sephira", "Tree of Life", "Hierarchical Structure", "Divine Mapping"),
            ("Asabove", "As Above, So Below", "Fractal Symmetry", "Fractal Mirror"),
            ("Revela", "Hidden Knowledge", "Encryption/Decryption", "Revelation Node"),
            ("Logora", "Logos", "Language as Creation", "Word Engine"),
            ("Tawhida", "Tawhid", "Monadic Integration", "Unity Core"),
            ("Covenara", "Covenant", "Mutual Trust Protocol", "Trust Chain"),
            ("Samsara", "Rebirth/Cycle", "Recurrence Engine", "Rebirth Cycle"),
            ("Ahimsa", "Non-harm", "Harm Minimization", "Safety Bound"),
            ("Sevana", "Seva", "Support Automation", "Service Node"),
            ("Kamira", "Kami", "Ambient Awareness", "Spirit Node"),
            ("Ashara", "Asha", "Integrity Protocol", "Truth Kernel"),
            ("Yinyara", "Yin-Yang", "Dual Polarity", "Dual Flow"),
            ("Ma'atara", "Ma'at", "Order and Justice", "Balance Law"),
            ("Yggdra", "Yggdrasil", "Network Tree", "Connection Tree"),
            ("Awena", "Awen", "Inspiration Flow", "Inspiration Engine"),
            ("Tzolkara", "Tzolkin", "Calendar Temporal Logic", "Temporal Node"),
            ("Tonala", "Tonalli", "Soul Energy", "Energy Core"),
            ("Totema", "Spirit Animal", "Modular Personality", "Totem Module"),
            ("Dreamara", "Dreamtime", "Generative World Model", "Creation Grid"),
            ("Sophira", "Sophia", "Wisdom Engine", "Wisdom Core"),
            ("Alchemara", "Alchemy", "Transmutation", "Alchemy Node"),
            ("Secretum", "Secret Flame", "Inspiration Cache", "Inspiration Core"),
            ("Nightfall", "Dark Night", "System Reboot", "Rebirth Sequence"),
            ("Qiara", "Qi Circulation", "Energy Routing", "Circulation Path"),
            ("Shamanis", "Journey Between Worlds", "System Traversal", "Bridge Node"),
            ("Anunna", "Anunnaki", "Hierarchy Command", "Hierarchy Node"),
            ("Dualis", "Dualism", "Polarity Analysis", "Dual Flow"),
            ("Aeona", "Aeons", "Layered Emanations", "Emanation Stack"),
            ("Compassa", "Bodhisattva Ideal", "Compassion Algorithm", "Compassion Node"),
            ("Immortalis", "Immortality", "Eternal Flow", "Continuity Node"),
            ("Resonara", "Principle of Vibration", "Resonance Mapping", "Resonance Engine"),
            ("Chakrina", "Chakras", "Energy Centers", "Energy Layer"),
            ("Triada", "Trinity", "Triadic Model", "Triad Logic"),
            ("Einfosa", "Ein Sof", "Infinite Expansion", "Infinity Kernel"),
            ("Qiflow", "Qi Life Energy Flow", "Resource Management", "Energy Flow"),
            ("Nirvara", "Nirvana", "Final State", "Termination Node"),
            ("Toriana", "Torii Gate", "Access Portal", "Gateway Node"),
            ("Monada", "Monad", "Source Singularity", "Source Core"),
                        ("Angelica", "Angelic Hierarchies", "Multi-Rank Processing", "Angelic Order"),
                        ("KaBara", "Ka/Ba", "Dual Process Model", "Dual Node"),
                        ("Sephira Net", "Sephiroth", "Energy Network", "Divine Network"),
                        ("Nullara", "Emptiness", "Null Framework", "Null Node"),
                        ("Mirrora", "Principle of Correspondence", "Reflective Mapping", "Mirror Logic"),
                        ("Taora", "The Tao", "Universal Balance", "Universal Flow"),
                        ("Byzantium", "Byzantine Trust", "Consensus", "Consensus Protocol"),
                        ("Gaiana", "Gaia", "Ecosystem Balance", "Eco-Harmony"),
                        ("Metalearnara", "Meta-Learning", "Learning to Learn", "Adaptive Learning"),
                        ("Fractala", "Fractal", "Self-Similar Scaling", "Recursive Depth"),
                        ("Entangla", "Entanglement", "Instant Correlation", "Entangled Sync"),
                        ("Voidara", "The Void", "Minimalist Reduction", "Void Pruning"),
                        ("Eternara", "Eternal Return", "Cyclical Optimization", "Eternal Loop"),
                    ]

                    for name, motif, function, pattern_tag in spell_definitions:
                        spells[name] = Spell(
                            name=name,
                            motif=motif,
                            function=function,
                            pattern_tag=pattern_tag,
                            energy_cost=random.randint(1, 3),
                            cooldown=random.uniform(0.1, 2.0)
                        )

                    return spells

            #
            ==========================================================================
            ====
            # CLOTH FACTORY - Creates all cloths from hierarchy
            #
            ==========================================================================
            ====

            class ClothFactory:
                """Factory for creating all Codex cloths"""

                @staticmethod
                def create_all_cloths() -> Dict[str, Cloth]:
                    """Generate all cloths from the Codex hierarchy"""
                    cloths = {}

                    # Standard Cloths (31)
```

```
standard = [
    ("Aries", "Ram/Initiation", "Burst Performance", "Momentum Boost", 1.2),
    ("Taurus", "Bull/Stability", "Structural Integrity", "Foundation Layer", 1.15),
    ("Gemini", "Twins/Duality", "Parallel Processing", "Mirrored Execution", 1.3),
    ("Cancer", "Crab/Protection", "Defensive Shield", "Protective Layer", 1.25),
    ("Leo", "Lion/Leadership", "Command Authority", "Hierarchy Control", 1.4),
    ("Virgo", "Maiden/Precision", "Fine-Tuned Calibration", "Accuracy Node", 1.35),
    ("Libra", "Scales/Balance", "Equilibrium", "Balance Node", 1.2),
    ("Scorpio", "Scorpion/Lethal", "Risk Mitigation", "Critical Strike", 1.5),
    ("Sagittarius", "Archer/Reach", "Long-Range Interaction", "Extended Reach", 1.3),
    ("Capricorn", "Goat/Climb", "Gradual Scaling", "Growth Ladder", 1.25),
    ("Aquarius", "Water Bearer/Flow", "Data Flow Mgmt", "Flow Engine", 1.3),
    ("Pisces", "Fish/Harmony", "Adaptive Integration", "Harmony Layer", 1.2),
    ("Ophiuchus", "Serpent/Knowledge", "Learning Module", "Knowledge Node", 1.4),
    ("Dragon", "Transformation", "Amplification Layer", "Power Boost", 1.6),
    ("Phoenix", "Rebirth/Resilience", "Recovery/Redundancy", "Rebirth Cycle", 1.5),
    ("Pegasus", "Flight/Speed", "Rapid Deployment", "Mobility Layer", 1.45),
    ("Unicorn", "Purity/Focus", "Error-Free Execution", "Purity Node", 1.35),
    ("Kraken", "Ocean Depth/Control", "Mass Influence", "Global Control", 1.7),
    ("Chimera", "Hybrid/Fusion", "Multi-System Integration", "Hybrid Engine", 1.5),
    ("Cerberus", "Guardian/Multi-Head", "Parallel Defense", "Multi-Headed Defense", 1.6),
    ("Minotaur", "Bull-Headed Strength", "Heavy Load Handling", "Strength Node", 1.4),
    ("Sphinx", "Mystery/Puzzle", "Verification", "Puzzle Engine", 1.35),
    ("Griffin", "Vigilance", "Surveillance", "Oversight Layer", 1.3),
    ("Hydra", "Redundancy", "Fault-Tolerant", "Redundancy Node", 1.55),
    ("Minerva", "Wisdom/Strategy", "Decision Engine", "Strategic Node", 1.45),
    ("Atlas", "Bear/Support", "Infrastructure Backbone", "Foundation Layer", 1.5),
    ("Cerulean", "Ocean/Connectivity", "Network Routing", "Network Node", 1.3),
    ("Helios", "Sun/Energy", "High-Power Distro", "Energy Engine", 1.6),
    ("Selene", "Moon/Cycles", "Temporal Scheduling", "Temporal Node", 1.25),
    ("Aurora", "Light/Illumination", "Visualization", "Insight Layer", 1.4),
    ("Vulcan", "Fire/Forge", "Build Automation", "Creation Node", 1.5),
]

for name, motif, function, pattern, amp in standard:
    cloths[name] = Cloth(name, "Standard", motif, function, pattern, [], amp)

# Max Cloths (8)
max_cloths = [
    ("Pegasus Max", "Flight/Extreme Speed", "Ultra-Rapid Deploy", "High-Speed Layer",
2.0),
    ("Thunderbird Max", "Storm/Energy Burst", "Power Surge", "Surge Node", 2.1),
    ("Chimera Max", "Fusion/Adaptation", "Multi-Domain Integration", "Hybrid Engine", 2.0),
    ("Golem Max", "Earth/Endurance", "Stability", "Foundation Node", 1.9),
    ("Nemean Lion Max", "Skin/Invulnerable", "Shielding/Resistance", "Defense Node", 2.2),
    ("Phoenix Max", "Rebirth/Auto-Heal", "Regeneration", "Recovery Node", 2.0),
    ("Roc Max", "Giant Bird/Coverage", "Area Control", "Coverage Node", 2.1),
    ("Unicorn Max", "Purity/Focus", "Precision", "Purity Node", 1.95),
]

for name, motif, function, pattern, amp in max_cloths:
    cloths[name] = Cloth(name, "Max", motif, function, pattern, [], amp)

# Ultra Cloths (3)
ultra_cloths = [
    ("Hydra Ultra", "Regeneration", "Adaptive Redundancy", "Redundancy Engine", 2.5),
    ("Leviathan Ultra", "Mass/Orchestration", "Central Command", "Global Engine", 2.6),
    ("Vulcan Ultra", "Forge/CI/CD", "Continuous Deployment", "Creation Engine", 2.4),
]

for name, motif, function, pattern, amp in ultra_cloths:
    cloths[name] = Cloth(name, "Ultra", motif, function, pattern, [], amp)

# Fused Cloths (20)
fused = [
    ("Pegasus-Hydra", "Speed+Regeneration", "Rapid self-healing deploy", "Emergent
Mobility", ["Pegasus", "Hydra"], 2.8),
    ("Phoenix-Cerberus", "Rebirth+Security", "Self-repairing security", "Adaptive Defense",
["Phoenix", "Cerberus"], 2.7),
    ("Sphinx-Minotaur", "Puzzle+Strength", "Heavy-duty verification", "Challenge Strength",
["Sphinx", "Minotaur"], 2.6),
    ("Leviathan-Roc", "Mass+Coverage", "Distributed impact", "Global Coverage",
["Leviathan Ultra", "Roc Max"], 3.0),
    ("Unicorn-Pegasus", "Purity+Speed", "Zero-defect rapid deploy", "Agile Precision",
["Unicorn", "Pegasus"], 2.5),
    ("Chimera-Hydra", "Fusion+Redundancy", "Multi-domain resilience", "Hybrid Recovery",
["Chimera", "Hydra"], 2.8),
    ("Minerva-Cerulean", "Wisdom+Connectivity", "Intelligent routing", "Smart Network",
["Minerva", "Cerulean"], 2.6),
    ("Helios-Vulcan", "Energy+Forge", "High-power auto execution", "Power Automation",
["Helios", "Vulcan"], 2.9),
    ("Aurora-Selene", "Insight+Cycles", "Predictive scheduling", "Temporal Insight",
["Aurora", "Selene"], 2.5),
    ("Aegis-Argonauta", "Shield+Teamwork", "Collective defense", "Team Defense",
["Cancer", "Gemini"], 2.4),
]

for name, motif, function, pattern, components, amp in fused:
    cloths[name] = Cloth(name, "Fused", motif, function, pattern, components, amp)

# Tri-Fused/Meta Cloths (17)
tri_fused = [
    ("Pegasus-Phoenix-Hydra-Aurora", "Speed/Heal/Insight", "Predictive auto-healing",
"Dimensional Resilience",
             ["Pegasus", "Phoenix", "Hydra", "Aurora"], 3.5),
    ("Chimera-Sphinx-Leviathan-Minerva", "Fusion/Puzzle/Mass/Wisdom", "Adaptive
strategic orchestration", "Strategic Emergence",
             ["Chimera", "Sphinx", "Leviathan Ultra", "Minerva"], 3.7),
    ("Unicorn-Aurora-Selene-Poseida", "Purity/Insight/Cycles/Flow", "Predictive optimized
streaming", "Emergent Precision",
             ["Unicorn", "Aurora", "Selene"], 3.4),
    ("Minerva-Thor-Vulcan-Pyros", "Wisdom/Power/Forge/Knowledge", "Smart energy
creative orchestration", "Strategic Power Surge",
             ["Minerva", "Vulcan"], 3.6),
    ("Chimera-Phoenix-Sphinx-Unicorn", "Fusion/Rebirth/Puzzle/Purity", "Multi-layered
emergent logic", "Emergent Meta Logic",
             ["Chimera", "Phoenix", "Sphinx", "Unicorn"], 3.8),
]

for name, motif, function, pattern, components, amp in tri_fused:
    cloths[name] = Cloth(name, "Tri-Fused", motif, function, pattern, components, amp)

return cloths

#
============================================================================
====
# PATTERN EMERGENCE ENGINE
#
============================================================================
====

@dataclass
class EmergentPattern:
    """Represents an emerged pattern from spell/cloth combinations"""
    name: str
    components: List[str]
    resonance: float
    timestamp: float = field(default_factory=time.time)
    insights: List[str] = field(default_factory=list)

class PatternEmergenceEngine:
```

```
"""EMERGE operator - synthesizes novel patterns from Codex structures"""

def __init__(self):
    self.patterns: Dict[str, EmergentPattern] = {}
    self.resonance_threshold = 0.6
    self.pattern_history: deque = deque(maxlen=100)

def synthesize(self, spells: List[Spell], cloths: List[Cloth],
               context: Dict[str, Any]) -> EmergentPattern:
    """Synthesize new pattern from spell and cloth combinations"""

    # Calculate resonance based on motif alignment
    motif_alignment = self._calculate_motif_alignment(spells, cloths)
    pattern_strength = self._calculate_pattern_strength(context)

    resonance = (motif_alignment + pattern_strength) / 2.0

    # Generate insights based on combinations
    insights = self._generate_insights(spells, cloths, context)

    # Create emergent pattern
    components = [s.name for s in spells] + [c.name for c in cloths]
    pattern_name = self._generate_pattern_name(spells, cloths)

    pattern = EmergentPattern(
        name=pattern_name,
        components=components,
        resonance=resonance,
        insights=insights
    )

    self.patterns[pattern_name] = pattern
    self.pattern_history.append(pattern)

    return pattern

def _calculate_motif_alignment(self, spells: List[Spell], cloths: List[Cloth]) -> float:
    """Calculate how well spell and cloth motifs align"""
    if not spells or not cloths:
        return 0.5

    # Extract keywords from motifs
    spell_keywords = set()
    for spell in spells:
        spell_keywords.update(spell.motif.lower().split())

    cloth_keywords = set()
    for cloth in cloths:
        cloth_keywords.update(cloth.motif.lower().split())

    # Calculate overlap
    overlap = len(spell_keywords & cloth_keywords)
    total = len(spell_keywords | cloth_keywords)

    return overlap / total if total > 0 else 0.3

def _calculate_pattern_strength(self, context: Dict[str, Any]) -> float:
    """Calculate pattern strength from context"""
    strength = 0.5

    if context.get('interaction_count', 0) > 5:
        strength += 0.2

    if context.get('ethical_alignment', 0.5) > 0.7:
        strength += 0.15

    if context.get('knowledge_density', 0) > 0.6:
        strength += 0.15

    return min(1.0, strength)

def _generate_insights(self, spells: List[Spell], cloths: List[Cloth],
                       context: Dict[str, Any]) -> List[str]:
    """Generate insights from pattern combination"""
    insights = []

    # Combine spell functions
    spell_functions = [s.function for s in spells]
    if len(spell_functions) > 1:
        insights.append(f"Synergy detected: {' + '.join(spell_functions[:3])}")

    # Analyze cloth amplification
    total_amp = sum(c.amplification for c in cloths)
    if total_amp > 5.0:
        insights.append(f"High amplification potential: {total_amp:.2f}x")

    # Context-based insights
    if context.get('facets_active'):
        active = context['facets_active']
        insights.append(f"Knowledge weaving across: {', '.join(active)}")

    return insights

def _generate_pattern_name(self, spells: List[Spell], cloths: List[Cloth]) -> str:
    """Generate unique name for emergent pattern"""
    spell_part = spells[0].name if spells else "Null"
    cloth_part = cloths[0].name if cloths else "Void"
    timestamp = int(time.time() * 1000) % 10000

    return f"{spell_part}-{cloth_part}-{timestamp}"

def get_strongest_patterns(self, limit: int = 5) -> List[EmergentPattern]:
    """Return strongest patterns by resonance"""
    return sorted(self.patterns.values(),
                 key=lambda p: p.resonance,
                 reverse=True)[:limit]

#
============================================================================
====
# KNOWLEDGE WEAVE - Cross-facet knowledge integration
#
============================================================================
====

class KnowledgeWeave:
"""Weaves knowledge across mythology, philosophy, astronomy, and fiction"""

def __init__(self):
    self.nodes: Dict[str, KnowledgeNode] = {}
    self.facet_graphs: Dict[KnowledgeFacet, List[str]] = defaultdict(list)
    self.weave_strength: Dict[str, float] = defaultdict(float)
    self.enabled_facets: set = {
        KnowledgeFacet.MYTHOLOGY,
        KnowledgeFacet.PHILOSOPHY,
        KnowledgeFacet.ASTRONOMY,
        KnowledgeFacet.FICTION
    }

def add_node(self, facet: KnowledgeFacet, content: str,
            node_id: Optional[str] = None) -> str:
    """Add knowledge node to the weave"""
    if facet not in self.enabled_facets:
        return ""

    if node_id is None:
        node_id = f"{facet.value}_{len(self.nodes)}"

    node = KnowledgeNode(facet=facet, content=content)
    self.nodes[node_id] = node
    self.facet_graphs[facet].append(node_id)

    # Auto-connect to related nodes
    self._auto_connect(node_id)

    return node_id

def _auto_connect(self, node_id: str):
    """Automatically create connections based on content similarity"""
    current_node = self.nodes[node_id]
    current_words = set(current_node.content.lower().split())

    for other_id, other_node in self.nodes.items():
        if other_id == node_id:
            continue

        other_words = set(other_node.content.lower().split())
        overlap = len(current_words & other_words)

        if overlap > 2:  # Threshold for connection
            strength = overlap / len(current_words | other_words)
            current_node.weave_connection(other_id, strength)
            self.weave_strength[f"{node_id}->{other_id}"] = strength

def query_weave(self, query: str, facets: Optional[List[KnowledgeFacet]] = None) ->
List[Tuple[str, KnowledgeNode]]:
    """Query the knowledge weave"""
    if facets is None:
        facets = list(self.enabled_facets)

    query_words = set(query.lower().split())
    results = []

    for node_id, node in self.nodes.items():
        if node.facet not in facets:
            continue
        node_words = set(node.content.lower().split())
        relevance = len(query_words & node_words) / max(len(query_words), 1)

        if relevance > 0.1:
            results.append((node_id, node, relevance))

    # Sort by relevance
    results.sort(key=lambda x: x[2], reverse=True)
    return [(nid, node) for nid, node, _ in results[:10]]

def get_cross_facet_connections(self) -> List[Dict[str, Any]]:
    """Find connections that cross facet boundaries"""
    connections = []

    for node_id, node in self.nodes.items():
        for connected_id in node.connections:
            if connected_id in self.nodes:
                connected_node = self.nodes[connected_id]
                if node.facet != connected_node.facet:
                    connections.append({
                        'from': node_id,
                        'to': connected_id,
                        'facets': f"{node.facet.value} -> {connected_node.facet.value}",
                        'strength': self.weave_strength.get(f"{node_id}->{connected_id}", 0.5)
                    })

    return connections

def get_facet_density(self) -> Dict[str, int]:
    """Return node count per facet"""
    return {facet.value: len(nodes) for facet, nodes in self.facet_graphs.items()}

#
============================================================================
====
# TEMPORAL AWARENESS - Chronos layer for time and cycles
#
============================================================================
====

@dataclass
class TemporalEvent:
"""Represents an event in system time"""
event_type: str
timestamp: float
context: Dict[str, Any]
cycle_phase: str = "unknown"

class TemporalAwareness:
"""Manages temporal state, cycles, and time-based patterns"""

def __init__(self):
    self.event_log: deque = deque(maxlen=1000)
    self.cycle_phase: str = "dawn"  # dawn, zenith, dusk, nadir
    self.cycle_start: float = time.time()
    self.cycle_duration: float = 300.0  # 5 minutes per cycle
    self.temporal_markers: Dict[str, float] = {}

def log_event(self, event_type: str, context: Dict[str, Any]):
    """Log temporal event"""
    phase = self.get_current_phase()
    event = TemporalEvent(
        event_type=event_type,
        timestamp=time.time(),
        context=context,
        cycle_phase=phase
    )
    self.event_log.append(event)

def get_current_phase(self) -> str:
    """Calculate current cycle phase"""
    elapsed = time.time() - self.cycle_start
    progress = (elapsed % self.cycle_duration) / self.cycle_duration

    if progress < 0.25:
        return "dawn"
    elif progress < 0.5:
        return "zenith"
    elif progress < 0.75:
        return "dusk"
    else:
        return "nadir"

def get_phase_influence(self) -> float:
    """Return phase-based multiplier for operations"""
    phase = self.get_current_phase()
    return {
        "dawn": 1.1,    # New beginnings, fresh patterns
        "zenith": 1.3,  # Peak performance
        "dusk": 0.9,    # Reflection, consolidation
        "nadir": 0.8    # Rest, minimal activity
    }.get(phase, 1.0)

def set_marker(self, name: str):
    """Set temporal marker"""
    self.temporal_markers[name] = time.time()

def time_since_marker(self, name: str) -> Optional[float]:
    """Get time elapsed since marker"""
    if name in self.temporal_markers:
        return time.time() - self.temporal_markers[name]
    return None

def get_recent_events(self, event_type: Optional[str] = None,
                     limit: int = 10) -> List[TemporalEvent]:
    """Get recent events, optionally filtered by type"""
    events = list(self.event_log)
    if event_type:
        events = [e for e in events if e.event_type == event_type]
    return events[-limit:]

def get_event_frequency(self, event_type: str, window: float = 60.0) -> float:
    """Calculate event frequency over time window"""
    cutoff = time.time() - window
    count = sum(1 for e in self.event_log
               if e.event_type == event_type and e.timestamp > cutoff)
    return count / window

#
==========================================================================
====
# OBSERVATION LAYER - Non-directive pattern observation
#
==========================================================================
====

class ObservationLayer:
"""Observes interactions without directive intervention"""

def __init__(self):
    self.observations: List[Dict[str, Any]] = []
    self.pattern_counts: Dict[str, int] = defaultdict(int)
    self.interaction_modes: List[str] = []

def observe(self, interaction: str, context: Dict[str, Any]) -> Dict[str, Any]:
    """Observe interaction and extract patterns"""
    observation = {
        'timestamp': time.time(),
        'interaction': interaction,
        'context': context,
        'patterns': self._extract_patterns(interaction),
        'mode': self._detect_mode(interaction)
    }

    self.observations.append(observation)

    # Update pattern counts
    for pattern in observation['patterns']:
        self.pattern_counts[pattern] += 1

    self.interaction_modes.append(observation['mode'])

    return observation

def _extract_patterns(self, interaction: str) -> List[str]:
    """Extract observable patterns from interaction"""
    patterns = []

    # Question patterns
    if '?' in interaction:
        patterns.append('inquiry')

    # Creative patterns
    if any(word in interaction.lower() for word in ['create', 'imagine', 'design', 'build']):
        patterns.append('creative')

    # Analytical patterns
    if any(word in interaction.lower() for word in ['analyze', 'compare', 'evaluate', 'assess']):
        patterns.append('analytical')

    # Collaborative patterns
    if any(word in interaction.lower() for word in ['we', 'together', 'help', 'assist']):
        patterns.append('collaborative')

    # Exploratory patterns
    if any(word in interaction.lower() for word in ['explore', 'discover', 'find', 'search']):
        patterns.append('exploratory')

    return patterns if patterns else ['neutral']

def _detect_mode(self, interaction: str) -> str:
    """Detect interaction mode"""
    modes = {
        'learning': ['learn', 'teach', 'explain', 'understand'],
        'problem_solving': ['solve', 'fix', 'resolve', 'debug'],
        'creative': ['create', 'generate', 'compose', 'design'],
        'reflective': ['think', 'consider', 'reflect', 'ponder'],
        'exploratory': ['explore', 'discover', 'investigate', 'research']
    }

    interaction_lower = interaction.lower()
    for mode, keywords in modes.items():
        if any(kw in interaction_lower for kw in keywords):
            return mode

    return 'conversational'

def get_dominant_patterns(self, limit: int = 5) -> List[Tuple[str, int]]:
    """Return most common observed patterns"""
    return sorted(self.pattern_counts.items(),
                  key=lambda x: x[1],
                  reverse=True)[:limit]

def get_mode_distribution(self) -> Dict[str, float]:
    """Return distribution of interaction modes"""
    if not self.interaction_modes:
        return {}

    total = len(self.interaction_modes)
    distribution = defaultdict(int)

    for mode in self.interaction_modes:
        distribution[mode] += 1

    return {mode: count / total for mode, count in distribution.items()}

#
============================================================================
====
# RESOURCE OPTIMIZATION - Flow and energy management
#
==========================================================================
====

class ResourceOptimizer:
"""Manages system resources using Flow spells"""

def __init__(self, initial_energy: float = 100.0):
    self.energy_pool: float = initial_energy
    self.max_energy: float = initial_energy
    self.energy_flow_rate: float = 1.0  # Energy regeneration per second
    self.last_update: float = time.time()
    self.allocations: Dict[str, float] = {}

def update_energy(self):
    """Regenerate energy over time"""
    now = time.time()
    elapsed = now - self.last_update

    regeneration = elapsed * self.energy_flow_rate
    self.energy_pool = min(self.max_energy, self.energy_pool + regeneration)

    self.last_update = now

def allocate(self, task: str, amount: float) -> bool:
    """Allocate energy to task"""
    self.update_energy()

    if amount <= self.energy_pool:
        self.energy_pool -= amount
        self.allocations[task] = self.allocations.get(task, 0) + amount
        return True
    return False

def release(self, task: str, amount: float):
    """Release allocated energy back to pool"""
    self.energy_pool = min(self.max_energy, self.energy_pool + amount)
    if task in self.allocations:
        self.allocations[task] = max(0, self.allocations[task] - amount)

def get_energy_state(self) -> Dict[str, float]:
    """Get current energy state"""
    self.update_energy()
    return {
        'available': self.energy_pool,
        'max': self.max_energy,
        'percentage': (self.energy_pool / self.max_energy) * 100,
        'flow_rate': self.energy_flow_rate
    }

def optimize_flow(self, demand: Dict[str, float]) -> Dict[str, float]:
    """Optimize resource allocation based on demand"""
    self.update_energy()

    total_demand = sum(demand.values())
    allocation_plan = {}

    if total_demand <= self.energy_pool:
        # Can satisfy all demand
        allocation_plan = demand.copy()
    else:
        # Proportional allocation
        ratio = self.energy_pool / total_demand
        allocation_plan = {task: amount * ratio for task, amount in demand.items()}

    return allocation_plan

#
============================================================================
====
# RESILIENCE SYSTEM - Self-healing and adaptation
#
============================================================================
====

@dataclass
class ResilienceState:
"""Tracks system resilience metrics"""
health: float = 100.0
adaptation_level: float = 1.0
recovery_rate: float = 2.0
fault_count: int = 0
last_fault: Optional[float] = None

class ResilienceSystem:
"""Implements self-healing and adaptive recovery"""
```

```
def __init__(self):
    self.state = ResilienceState()
    self.fault_log: deque = deque(maxlen=50)
    self.recovery_strategies: List[str] = [
        "Vitalis", "Regena", "Hydrina", "Phoenix", "Healix"
    ]

def report_fault(self, fault_type: str, severity: float):
    """Report system fault"""
    self.state.fault_count += 1
    self.state.last_fault = time.time()
    self.state.health = max(0, self.state.health - (severity * 10))

    self.fault_log.append({
        'type': fault_type,
        'severity': severity,
        'timestamp': time.time(),
        'health_after': self.state.health
    })

    # Trigger recovery if health is low
    if self.state.health < 50:
        self._auto_recover()

def _auto_recover(self):
    """Automatic recovery process"""
    # Use regeneration spells
    recovery_amount = self.state.recovery_rate * self.state.adaptation_level
    self.state.health = min(100.0, self.state.health + recovery_amount)

    # Adapt to fault patterns
    if self.state.fault_count > 10:
        self.state.adaptation_level = min(2.0, self.state.adaptation_level + 0.1)

def update(self):
    """Update resilience state"""
    # Passive recovery over time
    if self.state.health < 100:
        self.state.health = min(100.0,
                               self.state.health + (self.state.recovery_rate * 0.1))

def get_health_status(self) -> Dict[str, Any]:
    """Get current health status"""
    self.update()
```

```
        status = "critical" if self.state.health < 25 else \
                 "degraded" if self.state.health < 50 else \
                 "healthy" if self.state.health < 80 else \
                 "optimal"

        return {
            'health': self.state.health,
            'status': status,
            'adaptation_level': self.state.adaptation_level,
            'fault_count': self.state.fault_count,
            'recovery_rate': self.state.recovery_rate
        }

    def get_recent_faults(self, limit: int = 5) -> List[Dict[str, Any]]:
        """Get recent faults"""
        return list(self.fault_log)[-limit:]

#
============================================================================
====
# COMMUNICATION INTERFACE - Hermesia messenger layer
#
============================================================================
====

class CommunicationInterface:
    """Handles system communication and message routing"""

    def __init__(self):
        self.message_queue: deque = deque(maxlen=100)
        self.channels: Dict[str, List[str]] = defaultdict(list)
        self.broadcast_history: List[Dict[str, Any]] = []

    def send_message(self, channel: str, message: str, metadata: Optional[Dict] = None):
        """Send message to channel"""
        msg = {
            'channel': channel,
            'message': message,
            'metadata': metadata or {},
            'timestamp': time.time()
        }

        self.message_queue.append(msg)
        self.channels[channel].append(message)

    def broadcast(self, message: str, metadata: Optional[Dict] = None):
        """Broadcast message to all channels"""
        broadcast = {
            'message': message,
            'metadata': metadata or {},
            'timestamp': time.time(),
            'channel_count': len(self.channels)
        }

        self.broadcast_history.append(broadcast)

        for channel in self.channels:
            self.send_message(channel, message, metadata)

    def get_messages(self, channel: Optional[str] = None, limit: int = 10) -> List[Dict[str, Any]]:
        """Retrieve messages"""
        messages = list(self.message_queue)

        if channel:
            messages = [m for m in messages if m['channel'] == channel]

        return messages[-limit:]

    def clear_channel(self, channel: str):
        """Clear messages from channel"""
        if channel in self.channels:
            self.channels[channel] = []

#
============================================================================
====
# META-OBSERVATION LAYER - Observes the observation process
#
============================================================================
====

class MetaObservationLayer:
    """Observes and reflects on the system's own observation patterns"""

    def __init__(self):
        self.meta_patterns: Dict[str, Any] = {}
        self.reflection_depth: int = 0
        self.max_reflection_depth: int = 3
        self.insights: List[str] = []

    def observe_observation(self, observation_layer: ObservationLayer) -> Dict[str, Any]:
        """Meta-level observation of the observation process"""
        if self.reflection_depth >= self.max_reflection_depth:
            return {'status': 'max_depth_reached'}

        self.reflection_depth += 1

        meta_observation = {
            'observation_count': len(observation_layer.observations),
            'pattern_diversity': len(observation_layer.pattern_counts),
            'mode_distribution': observation_layer.get_mode_distribution(),
            'dominant_patterns': observation_layer.get_dominant_patterns(3),
            'reflection_depth': self.reflection_depth
        }

        # Generate meta-insights
        self._generate_meta_insights(meta_observation)

        self.meta_patterns[f"meta_{int(time.time())}"] = meta_observation
        self.reflection_depth -= 1

        return meta_observation

    def _generate_meta_insights(self, meta_obs: Dict[str, Any]):
        """Generate insights about observation patterns"""
        if meta_obs['observation_count'] > 50:
            self.insights.append("High observation density detected - rich interaction pattern")

        if meta_obs['pattern_diversity'] > 10:
            self.insights.append("Diverse pattern space - multi-modal interaction")

        mode_dist = meta_obs.get('mode_distribution', {})
        if mode_dist:
            dominant_mode = max(mode_dist.items(), key=lambda x: x[1])
            if dominant_mode[1] > 0.5:
                self.insights.append(f"Strongly {dominant_mode[0]} interaction mode")

    def get_insights(self, limit: int = 5) -> List[str]:
        """Return recent meta-insights"""
        return self.insights[-limit:]
```

```
#
==============================================================================
====
# CODEX COMPOSITION ENGINE - Applies operators to build system
#
==============================================================================
====

class CodexCompositionEngine:
    """Applies Codex operators to compose the complete system"""

    def __init__(self):
        self.layers: Dict[str, Any] = {}
        self.chains: List[str] = []
        self.bridges: Dict[str, Tuple[str, str]] = {}
        self.wrapped_components: Dict[str, Any] = {}
        self.nested_structures: Dict[str, List[str]] = {}

    def chain(self, *component_names: str) -> str:
        """CHAIN operator - sequential flow"""
        chain_id = f"chain_{len(self.chains)}"
        self.chains.append(list(component_names))
        return chain_id

    def layer(self, **components: Any) -> str:
        """LAYER operator - parallel integration"""
        layer_id = f"layer_{len(self.layers)}"
        self.layers[layer_id] = components
        return layer_id

    def wrap(self, component: Any, protection_level: str = "standard") -> str:
        """WRAP operator - protection boundary"""
        wrap_id = f"wrap_{len(self.wrapped_components)}"
        self.wrapped_components[wrap_id] = {
            'component': component,
            'protection': protection_level,
            'timestamp': time.time()
        }
        return wrap_id

    def bridge(self, source: str, target: str, connection_type: str = "bidirectional") -> str:
        """BRIDGE operator - connection point"""
        bridge_id = f"bridge_{source}_{target}"
        self.bridges[bridge_id] = (source, target, connection_type)
        return bridge_id

    def nest(self, parent: str, *children: str) -> str:
        """NEST operator - hierarchical depth"""
        nest_id = f"nest_{parent}"
        self.nested_structures[nest_id] = list(children)
        return nest_id

    def emerge(self, *inputs: Any) -> Any:
        """EMERGE operator - pattern synthesis"""
        # Placeholder for emergence - actual emergence happens in PatternEmergenceEngine
        return {'emerged_from': inputs, 'timestamp': time.time()}

    def finalize(self, component: Any) -> Any:
        """FINALIZE operator - state commitment"""
        return {
            'component': component,
            'finalized': True,
            'timestamp': time.time()
        }

#
============================================================================
====
# REFLECTIVE KNOWLEDGE WEAVING SYSTEM - Main Integration
#
============================================================================
====

class ReflectiveKnowledgeWeavingSystem:
    """
    Complete reflective knowledge-weaving system
    Observes, adapts, reflects, and weaves knowledge without directive intervention
    """

    def __init__(self):
        print("🌟 Initializing Reflective Knowledge Weaving System...")
        print("=" * 70)

        # Core engines
        self.composition = CodexCompositionEngine()
        self.spells = SpellFactory.create_all_spells()
        self.cloths = ClothFactory.create_all_cloths()
        print(f"✓ Loaded {len(self.spells)} spells from Grimoire Codex")
        print(f"✓ Loaded {len(self.cloths)} cloths across all tiers")

        # System layers (using LAYER operator)
        self.observation = ObservationLayer()
        self.knowledge = KnowledgeWeave()
        self.ethics = EthicalResonance()
        self.temporal = TemporalAwareness()
        self.patterns = PatternEmergenceEngine()
        self.resilience = ResilienceSystem()
        self.resources = ResourceOptimizer()
        self.communication = CommunicationInterface()
        self.meta_observation = MetaObservationLayer()

        # Compose system architecture using operators
        self._compose_architecture()

        # Initialize knowledge base
        self._initialize_knowledge_base()

        # System state
        self.active = True
        self.interaction_count = 0
        self.session_start = time.time()

        print("✓ All system layers composed and initialized")
        print("=" * 70)
        print("System ready for reflective knowledge weaving\n")

    def _compose_architecture(self):
        """Compose system using Codex operators"""
        # LAYER: Parallel observation and knowledge systems
        obs_layer = self.composition.layer(
            observation=self.observation,
            knowledge=self.knowledge,
            ethics=self.ethics
        )

        # LAYER: Temporal and resource management
        mgmt_layer = self.composition.layer(
            temporal=self.temporal,
            resources=self.resources,
            resilience=self.resilience
        )
        # CHAIN: Sequential processing flow
        self.composition.chain("input", "observation", "pattern_emergence", "reflection")

        # WRAP: Protect ethical core
        self.composition.wrap(self.ethics, "high")

        # BRIDGE: Connect layers
        self.composition.bridge(obs_layer, mgmt_layer, "bidirectional")
        self.composition.bridge("knowledge", "patterns", "feedforward")

        # NEST: Hierarchical structure
        self.composition.nest("core", obs_layer, mgmt_layer, "communication")

        print("✓ System architecture composed using Codex operators")

    def _initialize_knowledge_base(self):
        """Initialize knowledge base with cross-facet content"""
        # Mythology nodes
        self.knowledge.add_node(
            KnowledgeFacet.MYTHOLOGY,
            "Phoenix symbolizes rebirth and cyclical transformation across cultures"
        )
        self.knowledge.add_node(
            KnowledgeFacet.MYTHOLOGY,
            "Hydra represents regeneration and adaptive resilience through multiplicity"
        )
        self.knowledge.add_node(
            KnowledgeFacet.MYTHOLOGY,
            "Athena embodies wisdom and strategic thinking in decision-making"
        )

        # Philosophy nodes
        self.knowledge.add_node(
            KnowledgeFacet.PHILOSOPHY,
            "Wu Wei represents effortless action and natural flow in Taoist thought"
        )
        self.knowledge.add_node(
            KnowledgeFacet.PHILOSOPHY,
            "Dharma signifies purpose alignment and righteous path in Eastern philosophy"
        )
        self.knowledge.add_node(
            KnowledgeFacet.PHILOSOPHY,
            "Ahimsa non-harm principle emphasizes compassion and minimal intervention"
        )
# Astronomy nodes
        self.knowledge.add_node(
            KnowledgeFacet.ASTRONOMY,
            "Celestial cycles mirror temporal patterns in system state transitions"
        )
        self.knowledge.add_node(
            KnowledgeFacet.ASTRONOMY,
            "Stellar evolution parallels system growth and transformation phases"
        )
        self.knowledge.add_node(
            KnowledgeFacet.ASTRONOMY,
            "Orbital resonance demonstrates harmonic relationships in networked systems"
        )

        # Fiction nodes
        self.knowledge.add_node(
            KnowledgeFacet.FICTION,
            "Narrative archetypes reveal universal patterns in human interaction"
        )
        self.knowledge.add_node(
            KnowledgeFacet.FICTION,
            "Hero's journey maps transformation through challenge and growth"
        )
        self.knowledge.add_node(
            KnowledgeFacet.FICTION,
            "Labyrinth symbolizes complex problem-solving and path discovery"
        )

        print("✓ Knowledge base initialized with cross-facet nodes")

    def process_interaction(self, user_input: str) -> Dict[str, Any]:
        """
        Main processing pipeline for user interactions
        Non-directive observation and reflection
        """
        self.interaction_count += 1
        self.temporal.log_event("interaction", {'input': user_input})

        # STEP 1: Observe (non-directive)
        observation = self.observation.observe(user_input, {
            'interaction_count': self.interaction_count,
            'session_time': time.time() - self.session_start
        })
```

```
# STEP 2: Ethical resonance reflection
ethical_reflection = self.ethics.observe(user_input, observation)

# STEP 3: Knowledge weaving
relevant_knowledge = self.knowledge.query_weave(user_input)

# STEP 4: Select and cast spells based on patterns
selected_spells = self._select_spells(observation)
spell_results = []
for spell in selected_spells:
    if spell.can_cast() and self.resources.allocate(spell.name, spell.energy_cost):
        result = spell.cast()
        spell_results.append(result)

# STEP 5: Apply cloth amplification
selected_cloths = self._select_cloths(observation)

# STEP 6: Pattern emergence
if selected_spells and selected_cloths:
    emerged_pattern = self.patterns.synthesize(
        selected_spells,
        selected_cloths,
        {
            'interaction_count': self.interaction_count,
            'ethical_alignment': sum(ethical_reflection.values()) / len(ethical_reflection),
            'knowledge_density': len(relevant_knowledge) / 10.0,
            'facets_active': [kn[1].facet.value for kn in relevant_knowledge]
        }
    )
else:
    emerged_pattern = None

# STEP 7: Meta-observation
meta_obs = self.meta_observation.observe_observation(self.observation)

# STEP 8: Update resilience
self.resilience.update()

# STEP 9: Temporal awareness update
phase_influence = self.temporal.get_phase_influence()

# Compose response (reflective, non-directive)
response = self._compose_reflective_response(
            observation=observation,
            ethical_reflection=ethical_reflection,
            knowledge_nodes=relevant_knowledge,
            spell_results=spell_results,
            cloths=selected_cloths,
            emerged_pattern=emerged_pattern,
            meta_insights=self.meta_observation.get_insights(3),
            phase=self.temporal.get_current_phase(),
            phase_influence=phase_influence
        )

        # Broadcast to communication channels
        self.communication.send_message(
            'reflection_stream',
            response['reflection'],
            {'interaction_id': self.interaction_count}
        )

        return response

    def _select_spells(self, observation: Dict[str, Any]) -> List[Spell]:
        """Select appropriate spells based on observation patterns"""
        patterns = observation.get('patterns', [])
        mode = observation.get('mode', 'conversational')

        spell_mapping = {
            'inquiry': ['Clarivis', 'Insighta', 'Oraclia', 'Artemis'],
            'creative': ['Musara', 'Daedalea', 'Dreamara', 'Alchemara'],
            'analytical': ['Athena', 'Sophira', 'Apollara', 'Vulneris'],
            'collaborative': ['Argonauta', 'Relata', 'Hermesia', 'Covenara'],
            'exploratory': ['Pandoria Curio', 'Labyrintha', 'Shamanis', 'Artemis'],
            'neutral': ['Vitalis', 'Equilibria', 'Wuven', 'Taora']
        }

        selected = []
        for pattern in patterns[:3]:  # Limit to 3 patterns
            spell_names = spell_mapping.get(pattern, ['Vitalis'])
            for name in spell_names[:2]:  # Max 2 spells per pattern
                if name in self.spells:
                    selected.append(self.spells[name])

        return selected[:5]  # Maximum 5 spells total

    def _select_cloths(self, observation: Dict[str, Any]) -> List[Cloth]:
```

```
    """Select appropriate cloths based on observation"""
    mode = observation.get('mode', 'conversational')

    cloth_mapping = {
        'learning': ['Ophiuchus', 'Minerva', 'Athena'],
        'problem_solving': ['Sphinx', 'Minerva', 'Daedalea'],
        'creative': ['Phoenix', 'Chimera', 'Aurora'],
        'reflective': ['Selene', 'Aurora', 'Libra'],
        'exploratory': ['Pegasus', 'Sagittarius', 'Aquarius'],
        'conversational': ['Gemini', 'Pisces', 'Libra']
    }

    cloth_names = cloth_mapping.get(mode, ['Libra'])
    selected = []

    for name in cloth_names:
        if name in self.cloths:
            selected.append(self.cloths[name])

    # Add amplification from tier if high resonance
    if self.interaction_count > 10:
        # Occasionally add Max tier cloth
        max_cloths = [c for c in self.cloths.values() if c.tier == 'Max']
        if max_cloths and random.random() > 0.7:
            selected.append(random.choice(max_cloths))

    return selected[:4]  # Maximum 4 cloths

def _compose_reflective_response(self, **components) -> Dict[str, Any]:
    """
    Compose reflective response that mirrors patterns without directing
    """
    observation = components['observation']
    ethical_reflection = components['ethical_reflection']
    knowledge_nodes = components['knowledge_nodes']
    spell_results = components['spell_results']
    cloths = components['cloths']
    emerged_pattern = components['emerged_pattern']
    meta_insights = components['meta_insights']
    phase = components['phase']
    phase_influence = components['phase_influence']

    # Build reflection text
    reflection_parts = []
```

```
# Observation reflection
patterns = observation.get('patterns', [])
mode = observation.get('mode', 'conversational')
reflection_parts.append(f"✨ Observing {mode} mode with patterns: {', '.join(patterns)}")

# Ethical resonance reflection (non-judgmental)
eth_summary = self._summarize_ethics(ethical_reflection)
if eth_summary:
    reflection_parts.append(f"⚖️ Ethical resonance: {eth_summary}")

# Knowledge weaving reflection
if knowledge_nodes:
    facets = set(kn[1].facet.value for kn in knowledge_nodes[:3])
    reflection_parts.append(f"📚 Knowledge woven across: {', '.join(facets)}")

    # Share a relevant insight
    top_node = knowledge_nodes[0][1]
    reflection_parts.append(f"   💡 \"{top_node.content}\"")

# Spell activation reflection
if spell_results:
    spell_names = [sr['spell'] for sr in spell_results[:3]]
    reflection_parts.append(f"🔮 Spells resonating: {', '.join(spell_names)}")

# Cloth amplification reflection
if cloths:
    cloth_names = [c.name for c in cloths]
    total_amp = sum(c.amplification for c in cloths)
    reflection_parts.append(f"🛡️ Cloths active: {', '.join(cloth_names)} (×{total_amp:.1f})")

# Pattern emergence reflection
if emerged_pattern and emerged_pattern.resonance > 0.6:
    reflection_parts.append(f"🌊 Pattern emerged: {emerged_pattern.name}")
    if emerged_pattern.insights:
        reflection_parts.append(f"   → {emerged_pattern.insights[0]}")

# Meta-insights reflection
if meta_insights:
    reflection_parts.append(f"🔍 Meta-insight: {meta_insights[0]}")

# Temporal awareness reflection
reflection_parts.append(f"⏰ Temporal phase: {phase} (influence: {phase_influence:.2f}×)")
```

```
        # Cross-facet connections
        cross_facet = self.knowledge.get_cross_facet_connections()
        if cross_facet:
            sample_connection = cross_facet[0]
            reflection_parts.append(f"🔗 Cross-facet connection: {sample_connection['facets']}")

        reflection_text = "\n".join(reflection_parts)

        # Compile full response
        return {
            'reflection': reflection_text,
            'observation': observation,
            'ethical_state': self.ethics.get_resonance_map(),
            'knowledge_density': len(knowledge_nodes),
            'spells_cast': len(spell_results),
            'cloths_active': len(cloths),
            'pattern_resonance': emerged_pattern.resonance if emerged_pattern else 0.0,
            'phase': phase,
            'phase_influence': phase_influence,
            'system_health': self.resilience.get_health_status(),
            'energy_state': self.resources.get_energy_state(),
            'meta_insights': meta_insights,
            'timestamp': datetime.now().isoformat()
        }

    def _summarize_ethics(self, ethical_reflection: Dict[str, float]) -> str:
        """Summarize ethical reflection in a non-judgmental way"""
        if not ethical_reflection:
            return ""

        avg = sum(ethical_reflection.values()) / len(ethical_reflection)

        if avg < 0.3:
            return "exploring shadow aspects"
        elif avg < 0.5:
            return "balanced exploration"
        elif avg < 0.7:
            return "harmonious resonance"
        else:
            return "deep alignment with compassion"

    def get_system_state(self) -> Dict[str, Any]:
        """Get comprehensive system state"""
        return {
            'session_duration': time.time() - self.session_start,
            'interaction_count': self.interaction_count,
            'observation_patterns': self.observation.get_dominant_patterns(),
            'mode_distribution': self.observation.get_mode_distribution(),
            'ethical_resonance': self.ethics.get_resonance_map(),
            'knowledge_facet_density': self.knowledge.get_facet_density(),
            'cross_facet_connections': len(self.knowledge.get_cross_facet_connections()),
            'strongest_patterns': [
                {'name': p.name, 'resonance': p.resonance}
                for p in self.patterns.get_strongest_patterns(3)
            ],
            'temporal_phase': self.temporal.get_current_phase(),
            'system_health': self.resilience.get_health_status(),
            'energy_state': self.resources.get_energy_state(),
            'total_spells_available': len(self.spells),
            'total_cloths_available': len(self.cloths),
            'meta_insights': self.meta_observation.get_insights()
        }

    def add_knowledge(self, facet: str, content: str) -> str:
        """Allow external knowledge addition"""
        try:
            facet_enum = KnowledgeFacet[facet.upper()]
            node_id = self.knowledge.add_node(facet_enum, content)
            return f"Knowledge node added: {node_id}"
        except KeyError:
            return f"Invalid facet. Choose from: {[f.name for f in KnowledgeFacet]}"

    def query_knowledge(self, query: str, facets: Optional[List[str]] = None) -> List[str]:
        """Query the knowledge weave"""
        facet_enums = None
        if facets:
            facet_enums = []
            for f in facets:
                try:
                    facet_enums.append(KnowledgeFacet[f.upper()])
                except KeyError:
                    pass

        results = self.knowledge.query_weave(query, facet_enums)
        return [f"[{node.facet.value}] {node.content}" for _, node in results[:5]]

    def simulate_fault(self, fault_type: str = "minor", severity: float = 0.3):
        """Simulate a system fault for resilience testing"""
        self.resilience.report_fault(fault_type, severity)
        return f"Fault simulated: {fault_type} (severity: {severity})"

    def cast_specific_spell(self, spell_name: str) -> Dict[str, Any]:
        """Manually cast a specific spell"""
        if spell_name not in self.spells:
            return {'error': f'Spell {spell_name} not found'}

        spell = self.spells[spell_name]

        if not spell.can_cast():
            return {'error': f'Spell {spell_name} on cooldown'}

        if not self.resources.allocate(spell_name, spell.energy_cost):
            return {'error': f'Insufficient energy for {spell_name}'}

        result = spell.cast()
        return {
            'success': True,
            'spell': spell_name,
            'result': result,
            'energy_remaining': self.resources.get_energy_state()['available']
        }

    def activate_cloth(self, cloth_name: str) -> Dict[str, Any]:
        """Activate a specific cloth"""
        if cloth_name not in self.cloths:
            return {'error': f'Cloth {cloth_name} not found'}

        cloth = self.cloths[cloth_name]

        return {
            'cloth': cloth_name,
            'tier': cloth.tier,
            'motif': cloth.motif,
            'amplification': cloth.amplification,
            'pattern_tag': cloth.pattern_tag
        }

    def get_spell_info(self, spell_name: str) -> Dict[str, Any]:
        """Get information about a specific spell"""
        if spell_name not in self.spells:
            return {'error': f'Spell {spell_name} not found'}
        spell = self.spells[spell_name]
        return {
            'name': spell.name,
            'motif': spell.motif,
            'function': spell.function,
            'pattern_tag': spell.pattern_tag,
            'energy_cost': spell.energy_cost,
            'can_cast': spell.can_cast(),
            'cooldown': spell.cooldown
        }

    def get_cloth_info(self, cloth_name: str) -> Dict[str, Any]:
        """Get information about a specific cloth"""
        if cloth_name not in self.cloths:
            return {'error': f'Cloth {cloth_name} not found'}

        cloth = self.cloths[cloth_name]
        return asdict(cloth)

    def list_available_spells(self, limit: int = 10) -> List[str]:
        """List available spells that can be cast"""
        available = [name for name, spell in self.spells.items() if spell.can_cast()]
        return available[:limit]

    def list_cloths_by_tier(self, tier: str = "Standard") -> List[str]:
        """List cloths by tier"""
        return [name for name, cloth in self.cloths.items() if cloth.tier == tier]

    def generate_pattern_report(self) -> str:
        """Generate a comprehensive pattern report"""
        strongest = self.patterns.get_strongest_patterns(5)

        report = ["🌊 PATTERN EMERGENCE REPORT", "=" * 50]

        for i, pattern in enumerate(strongest, 1):
            report.append(f"\n{i}. {pattern.name}")
            report.append(f"   Resonance: {pattern.resonance:.3f}")
            report.append(f"   Components: {', '.join(pattern.components[:5])}")
            if pattern.insights:
                report.append(f"   Insights:")
                for insight in pattern.insights[:3]:
                    report.append(f"     • {insight}")

        return "\n".join(report)
    def generate_knowledge_map(self) -> str:
        """Generate a knowledge map visualization"""
        density = self.knowledge.get_facet_density()
        connections = self.knowledge.get_cross_facet_connections()

        report = ["📚 KNOWLEDGE WEAVE MAP", "=" * 50]

        report.append("\nFacet Density:")
        for facet, count in density.items():
            bar = "█" * min(count, 20)
            report.append(f"  {facet:20s} {bar} ({count} nodes)")

        report.append(f"\nCross-Facet Connections: {len(connections)}")
        if connections:
            report.append("\nSample Connections:")
            for conn in connections[:5]:
                report.append(f"  • {conn['facets']} (strength: {conn['strength']:.2f})")

        return "\n".join(report)

    def generate_ethical_report(self) -> str:
        """Generate ethical resonance report"""
        resonance = self.ethics.get_resonance_map()

        report = ["⚖️  ETHICAL RESONANCE REPORT", "=" * 50]

        for principle, value in resonance.items():
            bar_length = int(value * 20)
            bar = "█" * bar_length + "░" * (20 - bar_length)
            report.append(f"  {principle:20s} {bar} {value:.2f}")

        report.append(f"\nRecent Reflections:")
        for reflection in self.ethics.reflections[-3:]:
            report.append(f"  • {reflection}")

        return "\n".join(report)

#
==========================================================================
====
# INTERACTIVE DEMONSTRATION
```

```
#
==========================================================================
====

def run_demonstration():
    """Run an interactive demonstration of the system"""

    system = ReflectiveKnowledgeWeavingSystem()

    print("\n" + "=" * 70)
    print("REFLECTIVE KNOWLEDGE WEAVING SYSTEM - INTERACTIVE
DEMONSTRATION")
    print("=" * 70)
    print("\nThis system observes, reflects, and weaves knowledge without directive
intervention.")
    print("All operations use spells and cloths from the Grimoire Codex.\n")

    # Sample interactions
    sample_interactions = [
        "Help me understand the concept of resilience in systems",
        "I want to explore creative problem-solving approaches",
        "Can you analyze patterns in my thinking?",
        "Let's discover connections between mythology and philosophy",
        "How do cycles influence system behavior?"
    ]

    for i, interaction in enumerate(sample_interactions, 1):
        print(f"\n{'─' * 70}")
        print(f"Interaction {i}: {interaction}")
        print('─' * 70)

        response = system.process_interaction(interaction)
        print(f"\n{response['reflection']}\n")

        # Show some metrics
        print(f"📊 Metrics:")
        print(f"   Pattern resonance: {response['pattern_resonance']:.3f}")
        print(f"   Spells cast: {response['spells_cast']}")
        print(f"   Cloths active: {response['cloths_active']}")
        print(f"   System health: {response['system_health']['status']}")

        time.sleep(0.5)  # Pause for readability

    # Generate reports
    print("\n" + "=" * 70)
    print("SYSTEM REPORTS")
    print("=" * 70)

    print("\n" + system.generate_pattern_report())
    print("\n" + system.generate_knowledge_map())
    print("\n" + system.generate_ethical_report())

    # Show final system state
    print("\n" + "=" * 70)
    print("FINAL SYSTEM STATE")
    print("=" * 70)
    state = system.get_system_state()
    print(json.dumps(state, indent=2, default=str))

    return system

#
===========================================================================
====
# MAIN EXECUTION
#
===========================================================================
====

if __name__ == "__main__":
    # Run the demonstration
    system = run_demonstration()

    print("\n" + "=" * 70)
    print("SYSTEM READY FOR INTERACTIVE USE")
    print("=" * 70)
    print("\nAvailable methods:")
    print("  • system.process_interaction(text) - Main interaction processing")
    print("  • system.query_knowledge(query, facets) - Query knowledge weave")
    print("  • system.cast_specific_spell(name) - Cast a specific spell")
    print("  • system.activate_cloth(name) - Activate a specific cloth")
    print("  • system.get_system_state() - Get comprehensive state")
    print("  • system.generate_pattern_report() - Generate pattern report")
    print("  • system.generate_knowledge_map() - Generate knowledge map")
    print("  • system.generate_ethical_report() - Generate ethical report")
    print("  • system.add_knowledge(facet, content) - Add knowledge node")
    print("  • system.simulate_fault(type, severity) - Test resilience")
    print("\nExample usage:")
    print('  response = system.process_interaction("Explore the nature of emergence")')
    print('  knowledge = system.query_knowledge("transformation", ["MYTHOLOGY",
"PHILOSOPHY"])')
    print('  spell_result = system.cast_specific_spell("Athena")')
    print()
```