```
"""
ERES Institute for New Age Cybernetics ~ PlayNAC "KERNEL" Codebase V7.2
Empirical Realtime Education System × Human-Centered Skill Development
Platform

SIMPLIFIED VISION: "Design technology systems that incentivize human
development
instead of human exploitation"

Key Changes from V7.1:
- EarnedPath: Binary skill progression with verifiable credentials
- BiometricAuth: Simple proof-of-human validation (replacing BioPoW)
- 7 Core Development Areas (simplified from 17x7 matrix)
- ExpertAdvisor: Advisory guidance system (replacing GAIA governance)
- PeerReviewEngine: Community validation (replacing JAS consensus)
- MediaProcessor: Creative feedback engine (streamlined)
- Removed: Token economics, complex formulas, mystical elements
"""

import os
import time
import json
import hashlib
import sqlite3
import logging
from abc import ABC, abstractmethod
from dataclasses import dataclass, field
from typing import Any, Dict, List, Optional, Tuple, Union
from enum import Enum
import threading

#
============================================================================
===
# CORE CONFIGURATION & MODELS
#
============================================================================
===

class ConfigManager:
    """Simplified configuration manager"""
    def __init__(self, env_file: str = ".env"):
        self.env_file = env_file
        self.config = {}
        self.load()

    def load(self) -> None:
        """Load configuration from environment file"""
        if os.path.exists(self.env_file):
            with open(self.env_file) as f:
                for line in f:
                    if '=' in line and not line.strip().startswith('#'):
                        key, value = line.strip().split('=', 1)
                        self.config[key] = value

        # Set defaults
        self.config.setdefault('DATABASE_PATH', 'playnac_v71.db')
        self.config.setdefault('BIOMETRIC_THRESHOLD', '0.7')
        self.config.setdefault('PEER_REVIEW_THRESHOLD', '0.6')

    def get(self, key: str, default: Any = None) -> Any:
        return self.config.get(key, default)

# Core Development Areas (simplified from 17x7 matrix)
CORE_AREAS = [
    'Technical Skills',
    'Communication',
    'Problem-Solving',
    'Collaboration',
    'Ethics',
    'Creativity',
    'Leadership'
]

@dataclass
class SkillCredential:
    """Blockchain-verified skill credential"""
    skill_id: str
    user_id: str
    core_area: str
    timestamp: float
    proof_hash: str
    peer_validations: List[str] = field(default_factory=list)
    portfolio_evidence: Optional[str] = None

@dataclass
class Project:
    """User project for skill demonstration"""
    project_id: str
    user_id: str
    core_area: str
    title: str
    description: str
    submission_data: Dict[str, Any]
    timestamp: float
    status: str = "submitted"  # submitted, reviewing, approved, rejected

class SkillState(Enum):
    LOCKED = "locked"
    AVAILABLE = "available"
    COMPLETED = "completed"

#
==========================================================================
===
# STORAGE & PERSISTENCE
#
==========================================================================
===

class StorageAdapter:
    """SQLite-based storage for simplified kernel"""

    def __init__(self, db_path: str = 'playnac_v71.db'):
        self.conn = sqlite3.connect(db_path, check_same_thread=False)
        self.lock = threading.Lock()
        self._init_schema()

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

            # Skills and credentials
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS skills (
                    skill_id TEXT PRIMARY KEY,
                    name TEXT NOT NULL,
                    core_area TEXT NOT NULL,
                    prerequisites TEXT,
                    description TEXT
                )
            ''')

            cursor.execute('''
                CREATE TABLE IF NOT EXISTS user_skills (
                    user_id TEXT,
                    skill_id TEXT,
                    status TEXT,
                    completion_date REAL,
                    credential_hash TEXT,
                    PRIMARY KEY (user_id, skill_id)
                )
            ''')

            # Projects and portfolio
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS projects (
                    project_id TEXT PRIMARY KEY,
                    user_id TEXT,
                    core_area TEXT,
                    title TEXT,
                    description TEXT,
                    submission_data TEXT,
                    timestamp REAL,
                    status TEXT
                )
            ''')

            # Peer reviews
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS peer_reviews (
                    review_id TEXT PRIMARY KEY,
                    project_id TEXT,
                    reviewer_id TEXT,
                    score REAL,
                    feedback TEXT,
                    timestamp REAL
                )
            ''')

            # Expert advisors
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS expert_advisors (
                    advisor_id TEXT PRIMARY KEY,
                    name TEXT,
                    core_area TEXT,
                    credentials TEXT,
                    status TEXT
                )
            ''')

            self.conn.commit()

    def save_project(self, project: Project) -> None:
        """Save project to database"""
        with self.lock:
            cursor = self.conn.cursor()
```

```
            cursor.execute('''
                INSERT OR REPLACE INTO projects
                (project_id, user_id, core_area, title, description,
                 submission_data, timestamp, status)
                VALUES (?, ?, ?, ?, ?, ?, ?, ?)
            ''', (
                project.project_id, project.user_id, project.core_area,
                project.title, project.description,
                json.dumps(project.submission_data),
                project.timestamp, project.status
            ))
            self.conn.commit()

    def get_user_skills(self, user_id: str) -> Dict[str, str]:
        """Get user's skill completion status"""
        with self.lock:
            cursor = self.conn.cursor()
            cursor.execute('''
                SELECT skill_id, status FROM user_skills WHERE user_id = ?
            ''', (user_id,))
            return dict(cursor.fetchall())

    def save_peer_review(self, review_id: str, project_id: str,
                        reviewer_id: str, score: float, feedback: str) -> 
None:
        """Save peer review"""
        with self.lock:
            cursor = self.conn.cursor()
            cursor.execute('''
                INSERT INTO peer_reviews
                (review_id, project_id, reviewer_id, score, feedback, 
timestamp)
                VALUES (?, ?, ?, ?, ?, ?)
            ''', (review_id, project_id, reviewer_id, score, feedback, 
time.time()))
            self.conn.commit()

# 
===========================================================================
===
# EARNEDPATH - BINARY SKILL PROGRESSION
# 
===========================================================================
===

class EarnedPathEngine:
    """Binary skill progression & credential issuance"""
    def __init__(self, storage: StorageAdapter):
        self.storage = storage
        self.skills: Dict[str, bool] = {}
        self.skill_dependencies = self._load_skill_dependencies()

    def _load_skill_dependencies(self) -> Dict[str, List[str]]:
        """Load skill dependency graph"""
        # Example dependency structure
        return {
            'web_dev_basics': [],
            'javascript_fundamentals': ['web_dev_basics'],
            'react_development': ['javascript_fundamentals'],
            'full_stack_project': ['react_development'],
            'public_speaking': [],
            'technical_writing': ['public_speaking'],
            'team_leadership': ['public_speaking', 'technical_writing']
        }

    def check_prerequisites(self, skill_id: str, user_id: str) -> bool:
        """Check if user has completed prerequisite skills"""
        prerequisites = self.skill_dependencies.get(skill_id, [])
        user_skills = self.storage.get_user_skills(user_id)

        return all(
            user_skills.get(prereq) == SkillState.COMPLETED.value
            for prereq in prerequisites
        )

    def unlock_skill(self, skill_id: str, user_id: str,
                    prereqs_override: bool = False) -> bool:
        """Unlock skill for user if prerequisites are met"""
        if not prereqs_override and not self.check_prerequisites(skill_id,
user_id):
            return False

        # Update skill status to available
        with self.storage.lock:
            cursor = self.storage.conn.cursor()
            cursor.execute('''
                INSERT OR REPLACE INTO user_skills (user_id, skill_id,
status)
                VALUES (?, ?, ?)
            ''', (user_id, skill_id, SkillState.AVAILABLE.value))
            self.storage.conn.commit()

        return True
    def complete_skill(self, skill_id: str, user_id: str,
                       evidence_hash: str) -> bool:
        """Mark skill as completed and issue credential"""
        # Verify the user has the skill available
        user_skills = self.storage.get_user_skills(user_id)
        if user_skills.get(skill_id) != SkillState.AVAILABLE.value:
            return False

        # Generate credential
        credential = self._generate_credential(skill_id, user_id,
evidence_hash)

        # Update skill status
        with self.storage.lock:
            cursor = self.storage.conn.cursor()
            cursor.execute('''
                UPDATE user_skills
                SET status = ?, completion_date = ?, credential_hash = ?
                WHERE user_id = ? AND skill_id = ?
            ''', (
                SkillState.COMPLETED.value, time.time(),
                credential.proof_hash, user_id, skill_id
            ))
            self.storage.conn.commit()

        return True

    def _generate_credential(self, skill_id: str, user_id: str,
                             evidence_hash: str) -> SkillCredential:
        """Generate blockchain-verifiable credential"""
        timestamp = time.time()

        # Create proof hash combining skill, user, evidence, and timestamp
        proof_data = f"{skill_id}:{user_id}:{evidence_hash}:{timestamp}"
        proof_hash = hashlib.sha256(proof_data.encode()).hexdigest()

        return SkillCredential(
            skill_id=skill_id,
            user_id=user_id,
            core_area=self._get_skill_core_area(skill_id),
            timestamp=timestamp,
            proof_hash=proof_hash,
            portfolio_evidence=evidence_hash
        )

    def _get_skill_core_area(self, skill_id: str) -> str:
        """Map skill to core development area"""
        skill_area_map = {
            'web_dev_basics': 'Technical Skills',
            'javascript_fundamentals': 'Technical Skills',
            'react_development': 'Technical Skills',
            'full_stack_project': 'Technical Skills',
            'public_speaking': 'Communication',
            'technical_writing': 'Communication',
            'team_leadership': 'Leadership'
        }
        return skill_area_map.get(skill_id, 'Technical Skills')

#
============================================================================
===
# BIOMETRIC AUTHENTICATION - PROOF OF HUMAN
#
============================================================================
===

class BiometricAuth:
    """Simple proof-of-human: heartbeat, voice, or basic biometric
check"""

    def __init__(self, threshold: float = 0.7):
        self.threshold = threshold
        self.verification_cache = {}

    def verify_heartbeat(self, sample: bytes, user_id: str) -> bool:
        """Verify heartbeat pattern for proof-of-human"""
        # Stub implementation - in production would analyze:
        # - Heart rate variability
        # - Pattern consistency
        # - Liveness detection

        # Simple validation based on sample characteristics
        if len(sample) < 100:  # Minimum sample size
            return False

        # Basic entropy check to ensure non-synthetic data
        sample_entropy = self._calculate_entropy(sample)
        is_valid = sample_entropy > self.threshold

        # Cache verification for session
        if is_valid:
            self.verification_cache[user_id] = time.time()
        return is_valid

    def verify_voice(self, sample: bytes, user_id: str) -> bool:
        """Verify voice pattern for proof-of-human"""
        # Stub implementation - in production would analyze:
        # - Voice print characteristics
        # - Natural speech patterns
        # - Anti-spoofing measures

        if len(sample) < 1000:  # Minimum voice sample
            return False

        # Basic validation
        sample_entropy = self._calculate_entropy(sample)
        is_valid = sample_entropy > self.threshold * 0.8  # Slightly lower
threshold for voice

        if is_valid:
            self.verification_cache[user_id] = time.time()

        return is_valid

    def verify(self, sample: bytes, user_id: str, method: str =
'heartbeat') -> bool:
        """Generic verification method"""
        if method == 'heartbeat':
            return self.verify_heartbeat(sample, user_id)
        elif method == 'voice':
            return self.verify_voice(sample, user_id)
        else:
            # Fallback to basic entropy check
            return self._calculate_entropy(sample) > self.threshold

    def is_verified(self, user_id: str, max_age: float = 3600.0) -> bool:
        """Check if user has recent verification"""
        if user_id not in self.verification_cache:
            return False

        verification_time = self.verification_cache[user_id]
        return (time.time() - verification_time) < max_age

    def _calculate_entropy(self, data: bytes) -> float:
        """Calculate Shannon entropy of byte sequence"""
        if not data:
            return 0.0

        # Count byte frequencies
        frequencies = {}
        for byte in data:
            frequencies[byte] = frequencies.get(byte, 0) + 1

        # Calculate entropy
        length = len(data)
        entropy = 0.0
        for count in frequencies.values():
            probability = count / length
            if probability > 0:
                entropy -= probability * (probability.bit_length() - 1)

        return min(1.0, entropy / 8.0)  # Normalize to [0,1]

#
==========================================================================
===
# EXPERT ADVISOR SYSTEM
#
==========================================================================
===

class ExpertAdvisor:
    """Advisory system for curriculum guidance and expert
recommendations"""

    def __init__(self, storage: StorageAdapter):
        self.storage = storage
        self.advisors: Dict[str, List[str]] = self._load_advisors()

    def _load_advisors(self) -> Dict[str, List[str]]:
        """Load expert advisors by core area"""
        return {
            'Technical Skills': ['senior_developer_1', 'tech_lead_2'],
            'Communication': ['communication_coach_1', 'presenter_2'],
            'Leadership': ['team_lead_1', 'manager_2'],
            'Problem-Solving': ['architect_1', 'consultant_2'],
            'Ethics': ['ethicist_1', 'philosopher_2'],
            'Creativity': ['designer_1', 'artist_2'],
            'Collaboration': ['facilitator_1', 'team_coach_2']
        }

    def get_curriculum_recommendation(self, core_area: str,
                                      user_id: str) -> List[Dict[str, Any]]:
        """Get curriculum recommendations from experts"""
        if core_area not in self.advisors:
            return []
        # Get user's current skill level
        user_skills = self.storage.get_user_skills(user_id)
        completed_skills = [
            skill for skill, status in user_skills.items()
            if status == SkillState.COMPLETED.value
        ]

        # Generate recommendations based on progression
        recommendations = []

        if core_area == 'Technical Skills':
            if not completed_skills:
                recommendations.append({
                    'skill_id': 'web_dev_basics',
                    'title': 'Web Development Fundamentals',
                    'description': 'Learn HTML, CSS, and basic web
concepts',
                    'estimated_hours': 40,
                    'advisor': self.advisors[core_area][0]
                })
            elif 'web_dev_basics' in completed_skills:
                recommendations.append({
                    'skill_id': 'javascript_fundamentals',
                    'title': 'JavaScript Programming',
                    'description': 'Master JavaScript fundamentals and
ES6+',
                    'estimated_hours': 60,
                    'advisor': self.advisors[core_area][1]
                })

        return recommendations

    def get_expert_feedback(self, project_id: str,
                          advisor_id: str) -> Optional[Dict[str, Any]]:
        """Get expert feedback on a project"""
        # Stub implementation - in production would:
        # - Route to actual expert advisor
        # - Provide structured feedback
        # - Include improvement suggestions

        return {
            'advisor_id': advisor_id,
            'feedback': 'Strong technical implementation. Consider adding
error handling.',
            'strengths': ['Clean code structure', 'Good documentation'],
            'improvements': ['Add unit tests', 'Improve error handling'],
            'next_steps': ['Learn testing frameworks', 'Study error 
patterns'],
            'timestamp': time.time()
        }

# 
=============================================================================
===
# PEER REVIEW ENGINE
# 
=============================================================================
===

class PeerReviewEngine:
    """Community-driven project validation through peer review"""
    
    def __init__(self, storage: StorageAdapter, threshold: float = 0.6):
        self.storage = storage
        self.threshold = threshold
        self.min_reviews = 3
    
    def submit_project(self, project: Project) -> bool:
        """Submit project for peer review"""
        self.storage.save_project(project)
        return True
    
    def submit_review(self, project_id: str, reviewer_id: str, 
                     score: float, feedback: str) -> bool:
        """Submit peer review for a project"""
        if not (0.0 <= score <= 1.0):
            return False
        
        review_id = f"{project_id}_{reviewer_id}_{int(time.time())}"
        self.storage.save_peer_review(
            review_id, project_id, reviewer_id, score, feedback
        )
        
        # Check if project now meets approval threshold
        self._update_project_status(project_id)
        return True
    
    def get_project_rating(self, project_id: str) -> Dict[str, Any]:
        """Get current rating and status for project"""
        with self.storage.lock:
            cursor = self.storage.conn.cursor()
            cursor.execute('''
                SELECT score, feedback FROM peer_reviews
                WHERE project_id = ?
            ''', (project_id,))

            reviews = cursor.fetchall()

            if not reviews:
                return {
                    'average_score': 0.0,
                    'review_count': 0,
                    'status': 'pending',
                    'feedback': []
                }

            scores = [review[0] for review in reviews]
            feedback = [review[1] for review in reviews if review[1]]

            average_score = sum(scores) / len(scores)
            review_count = len(reviews)

            # Determine status
            if review_count < self.min_reviews:
                status = 'reviewing'
            elif average_score >= self.threshold:
                status = 'approved'
            else:
                status = 'needs_improvement'

            return {
                'average_score': average_score,
                'review_count': review_count,
                'status': status,
                'feedback': feedback
            }

    def _update_project_status(self, project_id: str) -> None:
        """Update project status based on peer reviews"""
        rating = self.get_project_rating(project_id)

        new_status = rating['status']
        if rating['review_count'] >= self.min_reviews:
            if rating['average_score'] >= self.threshold:
                new_status = 'approved'
            else:
                new_status = 'rejected'

        with self.storage.lock:
            cursor = self.storage.conn.cursor()
            cursor.execute('''
                UPDATE projects SET status = ? WHERE project_id = ?
            ''', (new_status, project_id))
            self.storage.conn.commit()

#
============================================================================
===
# MEDIA PROCESSOR - CREATIVE FEEDBACK
#
============================================================================
===

class MediaProcessor:
    """Automated feedback engine for creative submissions"""

    def __init__(self, quality_threshold: float = 0.5):
        self.quality_threshold = quality_threshold

    def assess_submission(self, submission_data: Dict[str, Any]) -> 
Dict[str, Any]:
        """Assess quality and provide feedback for creative submission"""

        assessment = {
            'quality_score': 0.0,
            'feedback': '',
            'strengths': [],
            'improvements': [],
            'meets_threshold': False
        }

        submission_type = submission_data.get('type', 'unknown')

        if submission_type == 'code':
            assessment = self._assess_code(submission_data)
        elif submission_type == 'design':
            assessment = self._assess_design(submission_data)
        elif submission_type == 'writing':
            assessment = self._assess_writing(submission_data)
        else:
            assessment['feedback'] = 'Unknown submission type'

        assessment['meets_threshold'] = assessment['quality_score'] >= 
self.quality_threshold
        return assessment

    def _assess_code(self, data: Dict[str, Any]) -> Dict[str, Any]:
        """Assess code submission quality"""
        code = data.get('content', '')

        # Basic code quality metrics
        line_count = len(code.split('\n'))
        has_comments = '#' in code or '//' in code or '/*' in code
        has_functions = 'def ' in code or 'function' in code
        has_error_handling = 'try' in code or 'catch' in code

        quality_score = 0.0
        strengths = []
        improvements = []

        # Scoring
        if line_count > 10:
            quality_score += 0.2
            strengths.append('Substantial code implementation')

        if has_comments:
            quality_score += 0.2
            strengths.append('Good documentation')
        else:
            improvements.append('Add comments to explain code logic')

        if has_functions:
            quality_score += 0.3
            strengths.append('Good code organization')

        if has_error_handling:
            quality_score += 0.3
            strengths.append('Includes error handling')
        else:
            improvements.append('Add error handling for robustness')

        feedback = f"Code analysis complete. {len(strengths)} strengths
identified."

        return {
            'quality_score': min(1.0, quality_score),
            'feedback': feedback,
            'strengths': strengths,
            'improvements': improvements
        }

    def _assess_design(self, data: Dict[str, Any]) -> Dict[str, Any]:
        """Assess design submission quality"""
        # Stub implementation for design assessment
        return {
            'quality_score': 0.7,
            'feedback': 'Design shows creativity and good composition',
            'strengths': ['Creative use of color', 'Clear visual
hierarchy'],
            'improvements': ['Consider accessibility guidelines']
        }

    def _assess_writing(self, data: Dict[str, Any]) -> Dict[str, Any]:
        """Assess writing submission quality"""
        content = data.get('content', '')
        word_count = len(content.split())

        quality_score = 0.4  # Base score
        strengths = []
        improvements = []

        if word_count > 100:
            quality_score += 0.3
            strengths.append('Adequate length and detail')

        if word_count > 500:
            quality_score += 0.2
            strengths.append('Comprehensive coverage')

        # Basic grammar check (very simple)
        sentences = content.split('.')
        if len(sentences) > 3:
            quality_score += 0.1
            strengths.append('Good sentence structure')

        feedback = f"Writing assessment: {word_count} words analyzed"

        return {
            'quality_score': min(1.0, quality_score),
            'feedback': feedback,
            'strengths': strengths,
            'improvements': improvements
        }

#
==========================================================================
===
# INTENT PARSER - SIMPLIFIED
#
==========================================================================
===
class IntentParser:
    """Maps user text to intent, parameters, and core development area"""

    def __init__(self):
        self.core_areas = CORE_AREAS

    def parse(self, text: str) -> Tuple[str, Dict[str, Any], str]:
        """Parse user intent and map to core area"""
        text_lower = text.lower()

        # Determine intent
        if any(word in text_lower for word in ['learn', 'study',
'skill']):
            intent = 'learn_skill'
            params = {'area': self._extract_core_area(text)}
        elif any(word in text_lower for word in ['submit', 'project',
'portfolio']):
            intent = 'submit_project'
            params = {'type': self._extract_project_type(text)}
        elif any(word in text_lower for word in ['review', 'feedback',
'rate']):
            intent = 'review_project'
            params = {}
        elif any(word in text_lower for word in ['status', 'progress',
'show']):
            intent = 'get_status'
            params = {}
        elif any(word in text_lower for word in ['help', 'guide',
'recommend']):
            intent = 'get_guidance'
            params = {'area': self._extract_core_area(text)}
        else:
            intent = 'unknown'
            params = {}

        core_area = self._extract_core_area(text)
        return intent, params, core_area

    def _extract_core_area(self, text: str) -> str:
        """Extract core development area from text"""
        text_lower = text.lower()

        for area in self.core_areas:
            if area.lower().replace('-', ' ') in text_lower:
                return area
        # Check for keywords that map to areas
        if any(word in text_lower for word in ['code', 'programming',
'development']):
            return 'Technical Skills'
        elif any(word in text_lower for word in ['speak', 'present',
'communicate']):
            return 'Communication'
        elif any(word in text_lower for word in ['solve', 'problem',
'debug']):
            return 'Problem-Solving'
        elif any(word in text_lower for word in ['team', 'collaborate',
'work together']):
            return 'Collaboration'
        elif any(word in text_lower for word in ['lead', 'manage',
'guide']):
            return 'Leadership'
        elif any(word in text_lower for word in ['create', 'design',
'art']):
            return 'Creativity'
        elif any(word in text_lower for word in ['ethics', 'moral',
'right']):
            return 'Ethics'

        return self.core_areas[0]  # Default to Technical Skills

    def _extract_project_type(self, text: str) -> str:
        """Extract project type from text"""
        text_lower = text.lower()

        if any(word in text_lower for word in ['code', 'program', 'app']):
            return 'code'
        elif any(word in text_lower for word in ['design', 'visual',
'ui']):
            return 'design'
        elif any(word in text_lower for word in ['write', 'essay',
'document']):
            return 'writing'
        else:
            return 'general'

#
============================================================================
===
# MAIN KERNEL ORCHESTRATOR
#
============================================================================
===
class PlayNACKernel:
    """
    Simplified PlayNAC Kernel: Human-verified skill development platform

    Core flow:
    1. User authenticates with biometric proof-of-human
    2. Intent parser determines user goals
    3. Route to appropriate module (skill learning, project submission,
peer review)
    4. Expert advisors provide guidance
    5. Community validates quality through peer review
    6. Verifiable credentials issued for completed skills
    """

    def __init__(self, config: ConfigManager):
        self.config = config
        self.storage = StorageAdapter(config.get('DATABASE_PATH'))

        # Initialize modules
        self.earned_path = EarnedPathEngine(self.storage)
        self.biometric_auth = BiometricAuth(
            float(config.get('BIOMETRIC_THRESHOLD', 0.7))
        )
        self.expert_advisor = ExpertAdvisor(self.storage)
        self.peer_review = PeerReviewEngine(
            self.storage,
            float(config.get('PEER_REVIEW_THRESHOLD', 0.6))
        )
        self.media_processor = MediaProcessor()
        self.intent_parser = IntentParser()

        # Session management
        self.active_sessions = {}

    def handle_request(self, user_id: str, message: str,
                       biometric_sample: Optional[bytes] = None) ->
Dict[str, Any]:
        """
        Main request handler

        Args:
            user_id: User identifier
            message: User's text input
            biometric_sample: Optional biometric data for authentication

        Returns:
```

```
    Response dictionary with status and data
"""

# Step 1: Authenticate user if biometric sample provided
if biometric_sample:
    if not self.biometric_auth.verify(biometric_sample, user_id):
        return {
            'status': 'error',
            'message': 'Human verification failed',
            'authenticated': False
        }

# Check if user has recent authentication
if not self.biometric_auth.is_verified(user_id):
    return {
        'status': 'error',
        'message': 'Authentication required',
        'authenticated': False
    }

# Step 2: Parse intent
intent, params, core_area = self.intent_parser.parse(message)

# Step 3: Route to appropriate handler
try:
    if intent == 'learn_skill':
        response = self._handle_learn_skill(user_id, params, core
```