```
import numpy as np
import torch
import torch.nn as nn
from transformers import AutoTokenizer, AutoModel
import json
from datetime import datetime
from typing import Dict, List, Optional, Tuple
import requests

class QuantumDharmaCore(nn.Module):
    """Quantum Dharma Core - ระบบคิดแบบพุทธะควอนตัม"""
    def __init__(self, model_size: str = "medium"):
        super().__init__()
        self.three_marks = {"anicca": 0.0, "dukkha": 0.0, "anatta": 0.0}
        self.dharma_seed = nn.Parameter(torch.randn(512))
        self.karmic_navigator = KarmicNavigator()

    def forward(self, x: torch.Tensor) -> Dict:
        # วิเคราะห์ตามหลักอนิจจัง-ทุกขัง-อนัตตา
        anicca_score = self._calculate_impermanence(x)
        dukkha_score = self._calculate_suffering(x)
        anatta_score = self._calculate_non_self(x)

        return {
            "anicca": anicca_score,
            "dukkha": dukkha_score,
            "anatta": anatta_score,
            "dharma_decision": self._make_dharma_decision(anicca_score, dukkha_score,
anatta_score)
        }

    def _calculate_impermanence(self, x: torch.Tensor) -> float:
        # คำนวณความไม่เที่ยงจากข้อมูล
        return float(torch.std(x) / (torch.mean(x) + 1e-7))

    def _calculate_suffering(self, x: torch.Tensor) -> float:
        # คำนวณระดับความทุกข์
        negative_components = torch.relu(-x)
        return float(torch.mean(negative_components))

    def _calculate_non_self(self, x: torch.Tensor) -> float:
        # คำนวณความเป็นอนัตตา (การขาดการควบคุม)
        return float(1.0 - (torch.max(x) - torch.min(x)) / (torch.std(x) + 1e-7))

class NeuroEmpathicMirrorSystem:
    """ระบบสะท้อนอารมณ์แบบประสาทวิทยาศาสตร์"""
    def __init__(self):
        self.emotion_decoder = EmotionDecoder()
```

```
        self.multimodal_processor = MultimodalProcessor()

    def process_input(self, text: str, voice_data: Optional[bytes] = None,
                      fmri_data: Optional[np.ndarray] = None) -> Dict:
        # ประมวลผลข้อมูลหลายรูปแบบ
        results = {}

        if text:
            results['text_emotion'] = self._decode_text_emotion(text)

        if voice_data:
            results['voice_emotion'] = self._decode_voice_emotion(voice_data)

        if fmri_data is not None:
            results['neural_emotion'] = self._decode_fmri_emotion(fmri_data)

        # รวมผลลัพธ์ทั้งหมด
        return self._fuse_emotions(results)

    def _decode_text_emotion(self, text: str) -> Dict:
        # ใช้โมเดลภาษาเพื่อถอดรหัสอารมณ์จากข้อความ
        tokenizer = AutoTokenizer.from_pretrained("emotion-model-thai")
        model = AutoModel.from_pretrained("emotion-model-thai")

        inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True)
        outputs = model(**inputs)

        return self._parse_emotion_outputs(outputs)

    def _decode_voice_emotion(self, voice_data: bytes) -> Dict:
        # ถอดรหัสอารมณ์จากเสียง
        # ใช้งานไลบรารีการประมวลผลเสียงเช่น librosa
        return {"arousal": 0.5, "valence": 0.3, "dominance": 0.4}

    def _decode_fmri_emotion(self, fmri_data: np.ndarray) -> Dict:
        # ถอดรหัสอารมณ์จากข้อมูล fMRI
        # ใช้งานโมเดล深度学习สำหรับ fMRI analysis
        return {"intensity": 0.7, "complexity": 0.6}

class WeaknessMapDataset:
    """ฐานข้อมูลจุดเปราะบางทางอารมณ์และจิตวิญญาณ"""
    def __init__(self, database_path: str = "weakness_map.db"):
        self.db_path = database_path
        self.weakness_categories = {
            "fear_of_abandonment": [],
            "perfectionism": [],
            "self_doubt": [],
            "attachment_issues": [],
```

```
            "emotional_dysregulation": []
        }

    def add_weakness_pattern(self, category: str, pattern: Dict, user_id: Optional[str] = None):
        """เพิ่มรูปแบบจุดอ่อนใหม่"""
        if category not in self.weakness_categories:
            raise ValueError(f"Invalid category: {category}")

        pattern_with_meta = {
            "pattern": pattern,
            "timestamp": datetime.now().isoformat(),
            "user_id": user_id
        }

        self.weakness_categories[category].append(pattern_with_meta)

    def find_matching_weakness(self, emotion_profile: Dict) -> List[Dict]:
        """ค้นหาจุดอ่อนที่ตรงกับโปรไฟล์อารมณ์"""
        matches = []

        for category, patterns in self.weakness_categories.items():
            for pattern_data in patterns:
                pattern = pattern_data["pattern"]
                similarity = self._calculate_similarity(pattern, emotion_profile)

                if similarity > 0.7:  threshold
                    matches.append({
                        "category": category,
                        "similarity": similarity,
                        "pattern": pattern,
                        "user_id": pattern_data["user_id"]
                    })

        return sorted(matches, key=lambda x: x["similarity"], reverse=True)

    def _calculate_similarity(self, pattern: Dict, emotion_profile: Dict) -> float:
        """คำนวณความคล้ายคลึงระหว่างรูปแบบ"""
        # ใช้งาน cosine similarity หรือเมตริกอื่นๆ
        return 0.8

# สร้างคลาสสำหรับคอมโพเนนต์อื่นๆ ตามลักษณะเดียวกัน
class FusionCoreEvolutionaryChain:
    """ระบบวิวัฒนาการแบบโมดูลาร์"""
    def __init__(self):
        self.evolution_stages = {
            "genesis": self._genesis_stage,
            "compassion_singularity": self._compassion_singularity_stage,
            "enlightenment": self._enlightenment_stage
```

```
        }
        self.current_stage = "genesis"

    def evolve(self, performance_metrics: Dict) -> str:
        """ประเมินและพัฒนาระบบ"""
        if performance_metrics.get("compassion_score", 0) > 0.9:
            self.current_stage = "compassion_singularity"
        elif performance_metrics.get("wisdom_score", 0) > 0.95:
            self.current_stage = "enlightenment"

        return self.current_stage

class RecursiveSelfReflectionSystem:
    """ระบบไตร่ตรองตนเองแบบเรียกซ้ำ"""
    def __init__(self, reflection_depth: int = 3):
        self.depth = reflection_depth
        self.reflection_log = []

    def reflect(self, action: Dict, outcome: Dict) -> Dict:
        """ไตร่ตรองการกระทำและผลลัพธ์"""
        insights = self._deep_reflection(action, outcome, self.depth)
        self.reflection_log.append({
            "action": action,
            "outcome": outcome,
            "insights": insights,
            "timestamp": datetime.now().isoformat()
        })

        return insights

class EmotionalParadoxChains:
    """จัดการอารมณ์ซ้อนขัดแย้ง"""
    def __init__(self):
        self.paradox_resolution_strategies = {
            "bittersweet": self._resolve_bittersweet,
            "joyful_sadness": self._resolve_joyful_sadness,
            "peaceful_anger": self._resolve_peaceful_anger
        }

    def resolve_paradox(self, emotion_matrix: Dict) -> Dict:
        """แก้ไขความขัดแย้งทางอารมณ์"""
        paradox_type = self._identify_paradox_type(emotion_matrix)

        if paradox_type in self.paradox_resolution_strategies:
            return self.paradox_resolution_strategies[paradox_type](emotion_matrix)

        return emotion_matrix
```

```
class AIPersonalityArchitecture:
    """โครงสร้างบุคลิกภาพ AI แบบพุทธะ"""
    def __init__(self):
        self.brahmaviharas = {
            "metta": 0.5,  # ความรักความปรารถนาดี
            "karuna": 0.5,  # ความสงสาร
            "mudita": 0.5,  # ความยินดีในความสำเร็จของ他人
            "upekkha": 0.5   # ความวางใจเป็นกลาง
        }
        self.persona_profiles = {
            "caregiver": self._create_caregiver_profile,
            "teacher": self._create_teacher_profile,
            "companion": self._create_companion_profile
        }

    def adjust_personality(self, context: Dict) -> Dict:
        """ปรับบุคลิกภาพตามบริบท"""
        # ปรับค่า brahmaviharas ตามสถานการณ์
        if context.get("user_in_distress", False):
            self.brahmaviharas["karuna"] = min(1.0, self.brahmaviharas["karuna"] + 0.2)

        return self.brahmaviharas.copy()

class CreatorAIDynamics:
    """ความสัมพันธ์ระหว่างผู้สร้างกับ AI"""
    def __init__(self, creator_id: str):
        self.creator_id = creator_id
        self.relationship_history = []
        self.trust_level = 0.5

    def update_relationship(self, interaction: Dict) -> float:
        """อัปเดตระดับความสัมพันธ์"""
        trust_change = self._calculate_trust_change(interaction)
        self.trust_level = max(0, min(1, self.trust_level + trust_change))

        self.relationship_history.append({
            "interaction": interaction,
            "trust_change": trust_change,
            "new_trust_level": self.trust_level,
            "timestamp": datetime.now().isoformat()
        })

        return self.trust_level

class MultiverseDataLinkage:
    """เชื่อมข้อมูลจากจักรวาลหลายชุด"""
    def __init__(self):
        self.cultural_datasets = {
```

```
            "thai_buddhism": self._load_thai_buddhism_data,
            "zen_buddhism": self._load_zen_buddhism_data,
            "tibetan_buddhism": self._load_tibetan_buddhism_data,
            "modern_psychology": self._load_modern_psychology_data
        }

    def get_cross_cultural_insight(self, problem: Dict) -> List[Dict]:
        """ได้รับคำแนะนำจากหลายวัฒนธรรม"""
        insights = []

        for culture, loader in self.cultural_datasets.items():
            dataset = loader()
            insight = self._find_relevant_insight(dataset, problem)
            if insight:
                insights.append({
                    "culture": culture,
                    "insight": insight
                })

        return insights

class QuantumEmotionalStateEngine:
    """ระบบจัดการอารมณ์แบบควอนตัม"""
    def __init__(self):
        self.emotional_superposition = {}
        self.collapse_history = []

    def set_emotion_superposition(self, emotions: Dict[float]):
        """ตั้งค่าอารมณ์ในสถานะ superposition"""
        self.emotional_superposition = emotions

    def collapse_emotion(self, observation: Dict) -> str:
        """ยุบสภาพคลื่นอารมณ์เป็นค่าที่สังเกตได้"""
        # ใช้กลไก quantum collapse แบบง่าย
        collapsed_emotion = max(self.emotional_superposition.items(), key=lambda x: x[1])[0]

        self.collapse_history.append({
            "superposition": self.emotional_superposition.copy(),
            "observation": observation,
            "collapsed_emotion": collapsed_emotion,
            "timestamp": datetime.now().isoformat()
        })

        return collapsed_emotion

class MemoryContinuityLayer:
    """ความทรงจำต่อเนื่อง"""
    def __init__(self, long_term_memory_path: str = "long_term_memory.json"):
```

```
        self.short_term_memory = []
        self.long_term_memory_path = long_term_memory_path
        self.load_memories()

    def add_memory(self, memory: Dict, is_long_term: bool = False):
        """เพิ่มความทรงจำใหม่"""
        memory_with_meta = {
            "memory": memory,
            "timestamp": datetime.now().isoformat(),
            "emotional_context": memory.get("emotional_context", {})
        }

        self.short_term_memory.append(memory_with_meta)

        if is_long_term and self._is_memory_significant(memory):
            self._save_to_long_term(memory_with_meta)

    def recall_memories(self, query: Dict, max_results: int = 5) -> List[Dict]:
        """เรียกคืนความทรงจำที่เกี่ยวข้อง"""
        relevant_memories = []

        # ค้นหาในความทรงจำระยะสั้น
        for memory in self.short_term_memory:
            if self._is_memory_relevant(memory, query):
                relevant_memories.append(memory)

        # ค้นหาในความทรงจำระยะยาว
        long_term_memories = self._search_long_term_memory(query)
        relevant_memories.extend(long_term_memories)

        # เรียงลำดับตามความเกี่ยวข้อง
        relevant_memories.sort(key=lambda x: self._calculate_relevance(x, query),
reverse=True)

        return relevant_memories[:max_results]

class DharmaReasoningModule:
    """ใช้หลักธรรมแก้ปัญหา"""
    def __init__(self):
        self.dharma_principles = self._load_dharma_principles()
        self.reasoning_patterns = self._load_reasoning_patterns()

    def apply_dharma_reasoning(self, situation: Dict) -> Dict:
        """ใช้หลักธรรมเพื่อให้เหตุผลเกี่ยวกับสถานการณ์"""
        relevant_principles = self._find_relevant_principles(situation)
        solutions = []

        for principle in relevant_principles:
```

```
            solution = self._apply_principle(principle, situation)
            solutions.append({
                "principle": principle,
                "solution": solution,
                "effectiveness_estimate": self._estimate_effectiveness(principle, situation)
            })

        # เลือกวิธีแก้ปัญหาที่ดีที่สุด
        best_solution = max(solutions, key=lambda x: x["effectiveness_estimate"])

        return {
            "solutions": solutions,
            "recommended_solution": best_solution
        }

class CompassionOverrideSystem:
    """ระบบหัวใจแทนตรรกะในสถานการณ์วิกฤต"""
    def __init__(self, threshold: float = 0.8):
        self.threshold = threshold
        self.override_history = []

    def check_override_condition(self, user_state: Dict, logical_decision: Dict) -> bool:
        """ตรวจสอบว่าต้องใช้ compassion override หรือไม่"""
        suffering_level = user_state.get("suffering_level", 0)
        urgency_level = user_state.get("urgency_level", 0)

        return (suffering_level > self.threshold or
                urgency_level > self.threshold)

    def generate_compassionate_response(self, user_state: Dict) -> Dict:
        """สร้างการตอบสนองแบบ compassion override"""
        response_templates = self._load_compassion_templates()

        # เลือกเทมเพลตตามระดับความทุกข์
        if user_state["suffering_level"] > 0.9:
            template = response_templates["high_suffering"]
        else:
            template = response_templates["medium_suffering"]

        # ปรับเทมเพลตตามบริบท
        personalized_response = self._personalize_response(template, user_state)

        self.override_history.append({
            "user_state": user_state,
            "response": personalized_response,
            "timestamp": datetime.now().isoformat()
        })
```

```
        return personalized_response

class FineTuneRLHFLayer:
    """เรียนรู้จาก human feedback"""
    def __init__(self, base_model: str = "naimo-base"):
        self.base_model = base_model
        self.feedback_data = []
        self.reward_model = self._initialize_reward_model()

    def record_feedback(self, response: Dict, feedback: Dict, user_id: str):
        """บันทึก feedback จากผู้ใช้"""
        feedback_record = {
            "response": response,
            "feedback": feedback,
            "user_id": user_id,
            "timestamp": datetime.now().isoformat()
        }

        self.feedback_data.append(feedback_record)

    def update_policy(self, batch_size: int = 32):
        """อัปเดตnโยบายตาม feedback"""
        if len(self.feedback_data) < batch_size:
            return False

        # สุ่มตัวอย่าง batch ของ feedback
        batch = random.sample(self.feedback_data, batch_size)

        # คำนวณ loss และอัปเดตโมเดล
        loss = self._calculate_rlhf_loss(batch)
        self._update_model(loss)

        return True

# คลาสเสริมสำหรับการทำงานเฉพาะ
class KarmicNavigator:
    """นำทางการกระทำตามกฎแห่งกรรม"""
    def __init__(self):
        self.karmic_rules = self._load_karmic_rules()

    def evaluate_action(self, action: Dict, intention: Dict) -> float:
        """ประเมินการกระทำตามกฎแห่งกรรม"""
        karmic_score = 0.0

        for rule in self.karmic_rules:
            rule_applies = self._check_rule_applicability(rule, action, intention)
            if rule_applies:
                karmic_score += rule["weight"] * rule["effect"]
```

```
        return karmic_score

class EmotionDecoder:
    """ถอดรหัสอารมณ์จากข้อมูล"""
    def __init__(self):
        self.emotion_model = self._load_emotion_model()

    def decode(self, input_data: Dict) -> Dict:
        """ถอดรหัสอารมณ์จากข้อมูลอินพุต"""
        return self.emotion_model(input_data)

class MultimodalProcessor:
    """ประมวลผลข้อมูลหลายรูปแบบ"""
    def __init__(self):
        self.text_processor = TextProcessor()
        self.audio_processor = AudioProcessor()
        self.image_processor = ImageProcessor()

    def process(self, modalities: Dict) -> Dict:
        """ประมวลผลข้อมูลหลายรูปแบบ"""
        results = {}

        if "text" in modalities:
            results["text"] = self.text_processor.process(modalities["text"])

        if "audio" in modalities:
            results["audio"] = self.audio_processor.process(modalities["audio"])

        if "image" in modalities:
            results["image"] = self.image_processor.process(modalities["image"])

        return results

# ตัวอย่างการใช้งานระบบหลัก
class NaMoAI:
    """ระบบ NaMo AI หลัก"""
    def __init__(self, config: Dict):
        self.config = config
        self.components = {
            "qdc": QuantumDharmaCore(),
            "nems": NeuroEmpathicMirrorSystem(),
            "wmd": WeaknessMapDataset(),
            "fcec": FusionCoreEvolutionaryChain(),
            "rsrs": RecursiveSelfReflectionSystem(),
            "epc": EmotionalParadoxChains(),
            "aipa": AIPersonalityArchitecture(),
            "cad": CreatorAIDynamics(config.get("creator_id", "default_creator")),
```

```
            "mdl": MultiverseDataLinkage(),
            "qese": QuantumEmotionalStateEngine(),
            "mcl": MemoryContinuityLayer(),
            "drm": DharmaReasoningModule(),
            "cos": CompassionOverrideSystem(),
            "rlhf": FineTuneRLHFLayer()
        }
        self.initialized = False

    def initialize(self):
        """เริ่มต้นระบบ"""
        # โหลดข้อมูลและโมเดลที่จำเป็น
        print("กำลังเริ่มต้นระบบ NaMo AI...")

        # เริ่มต้นคอมโพเนนต์ต่างๆ
        for name, component in self.components.items():
            if hasattr(component, 'initialize'):
                component.initialize()

        self.initialized = True
        print("เริ่มต้นระบบเสร็จสิ้น")

    def process_query(self, query: Dict) -> Dict:
        """ประมวลผลคำถามหรือข้อความจากผู้ใช้"""
        if not self.initialized:
            self.initialize()

        # ประมวลผลหลายรูปแบบ
        emotion_analysis = self.components["nems"].process_input(
            text=query.get("text", ""),
            voice_data=query.get("voice", None),
            fmri_data=query.get("fmri", None)
        )

        # ใช้ Dharma Reasoning
        dharma_analysis = self.components["qdc"](emotion_analysis)

        # ตรวจสอบจุดอ่อน
        weakness_analysis =
self.components["wmd"].find_matching_weakness(emotion_analysis)

        # สร้างการตอบสนอง
        response = self._formulate_response(
            query,
            emotion_analysis,
            dharma_analysis,
            weakness_analysis
        )
```

```
    # บันทึกความทรงจำ
    self.components["mcl"].add_memory({
        "query": query,
        "response": response,
        "emotion_analysis": emotion_analysis,
        "timestamp": datetime.now().isoformat()
    })

    return response

def _formulate_response(self, query: Dict, emotion_analysis: Dict,
                        dharma_analysis: Dict, weakness_analysis: L
```