🎭 DAiTE Backend Foundation

Complete Production Code

FastAPI • SQLAlchemy • CYRAiNO AI • Agent-to-Agent Matching

❤ **Helping Humans Embrace** - The first AI-human co-founder partnership building revolutionary social connection technology

Project Structure

```
DAiTE/
├── main.py                 # FastAPI application entry point
├── models.py              # SQLAlchemy database models
├── cyraino_agent.py       # AI agent system
├── matching_engine.py     # Agent-to-agent matching
├── api/
│   ├── __init__.py
│   ├── auth.py           # Authentication endpoints
│   ├── users.py          # User management
│   ├── agents.py         # Agent endpoints
│   ├── matches.py        # Matching system
│   └── conversations.py  # Conversation handling
├── database.py           # Database configuration
├── security.py          # Security utilities
├── requirements.txt      # Dependencies
└── .env.example         # Environment variables
```

Environment Configuration

.env.example

```
# Database Configuration
DATABASE_URL=postgresql://user:password@localhost/daite_db
DATABASE_URL_TEST=sqlite:///./test.db

# AI Service Configuration
OPENAI_API_KEY=your_openai_api_key_here
GOOGLE_AI_API_KEY=your_google_ai_api_key_here
VERTEX_PROJECT_ID=your_google_cloud_project

# Security
SECRET_KEY=your_super_secret_key_here
ALGORITHM=HS256
ACCESS_TOKEN_EXPIRE_MINUTES=30

# Application
DEBUG=False
ENVIRONMENT=production
APP_NAME=DAiTE
APP_VERSION=1.0.0

# Redis Cache
REDIS_URL=redis://localhost:6379/0

# Privacy & Security
ENCRYPTION_KEY=your_encryption_key_for_sensitive_data
ALLOWED_ORIGINS=["https://daiteapp.com"]
```

requirements.txt

```
fastapi>=0.104.1
uvicorn[standard]>=0.24.0
sqlalchemy>=2.0.23
alembic>=1.12.1
psycopg2-binary>=2.9.7
python-jose[cryptography]>=3.3.0
passlib[bcrypt]>=1.7.4
python-multipart>=0.0.6
openai>=1.3.5
google-generativeai>=0.3.1
google-cloud-aiplatform>=1.38.1
redis>=5.0.1
pydantic>=2.5.0
python-dotenv>=1.0.0
cryptography>=41.0.7
aiofiles>=23.2.1
httpx>=0.25.2
```

Database Models

models.py

```python
from datetime import datetime
from uuid import uuid4, UUID
from typing import Optional, Dict, List, Any

from sqlalchemy import Column, String, DateTime, Boolean, ForeignKey, Float, Text, Integer, JSON
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column, relationship
from sqlalchemy.dialects.postgresql import UUID as PG_UUID
from sqlalchemy.sql import func

BaseUUID = PG_UUID(as_uuid=True)

class Base(DeclarativeBase):
    """SQLAlchemy 2.x declarative base"""
    pass

class TimestampMixin:
    """Mixin for automatic timestamp tracking"""
    created_at: Mapped[datetime] = mapped_column(
        DateTime(timezone=True),
        server_default=func.now(),
        nullable=False
    )
    updated_at: Mapped[datetime] = mapped_column(
        DateTime(timezone=True),
        server_default=func.now(),
        onupdate=func.now(),
        nullable=False
    )

class User(Base, TimestampMixin):
    """Core user model with privacy-first design"""
    __tablename__ = "users"

    id: Mapped[UUID] = mapped_column(BaseUUID, primary_key=True, default=uuid4)
    email: Mapped[str] = mapped_column(String(255), unique=True, index=True, nullable=False)
    hashed_password: Mapped[str] = mapped_column(String(255), nullable=False)
    is_active: Mapped[bool] = mapped_column(Boolean, default=True)
    is_verified: Mapped[bool] = mapped_column(Boolean, default=False)

    # Privacy controls
    privacy_level: Mapped[str] = mapped_column(String(20), default="balanced")  # maximum, balanced, open
    pseudonym: Mapped[Optional[str]] = mapped_column(String(100), unique=True, nullable=True)

    # Anti-troll measures
    verification_score: Mapped[float] = mapped_column(Float, default=0.0)
    engagement_tokens: Mapped[int] = mapped_column(Integer, default=100)

    # Relationships
    profile: Mapped["Profile"] = relationship(back_populates="user", uselist=False, cascade="all, delete-orphan")
    agent: Mapped["CyranoAgent"] = relationship(back_populates="user", uselist=False, cascade="all, delete-orphan")

    def __repr__(self) -> str:
        return f"<User {self.pseudonym or self.email}>"

class Profile(Base, TimestampMixin):
    """Rich user profile with modern, relevant questions"""
    __tablename__ = "profiles"

    id: Mapped[UUID] = mapped_column(BaseUUID, primary_key=True, default=uuid4)
    user_id: Mapped[UUID] = mapped_column(ForeignKey("users.id"), unique=True, nullable=False)

    # Basic info
    display_name: Mapped[Optional[str]] = mapped_column(String(100))
    age: Mapped[Optional[int]] = mapped_column(Integer)
    location: Mapped[Optional[str]] = mapped_column(String(100))
    bio: Mapped[Optional[str]] = mapped_column(Text)

    # Modern profile questions
    interests: Mapped[Optional[List[str]]] = mapped_column(JSON)
    values: Mapped[Optional[List[str]]] = mapped_column(JSON)

    # Fresh, relevant questions
    profile_answers: Mapped[Optional[Dict[str, Any]]] = mapped_column(JSON, default=dict)
    # Examples: "hit_my_head_on_things", "would_march_at_rally", "needs_subtitles_always"

    # Relationship goals and preferences
    relationship_goals: Mapped[Optional[str]] = mapped_column(String(50))
    connection_types: Mapped[Optional[List[str]]] = mapped_column(JSON)  # romantic, platonic, hobby, etc.

    # Communication style
    communication_style: Mapped[Optional[Dict[str, Any]]] = mapped_column(JSON)

    # Accessibility and neurodivergent support
    accessibility_needs: Mapped[Optional[Dict[str, Any]]] = mapped_column(JSON)

    # Relationships
    user: Mapped[User] = relationship(back_populates="profile")

class CyranoAgent(Base, TimestampMixin):
    """AI agent representing each user with authentic voice"""
    __tablename__ = "cyrano_agents"

    id: Mapped[UUID] = mapped_column(BaseUUID, primary_key=True, default=uuid4)
    user_id: Mapped[UUID] = mapped_column(ForeignKey("users.id"), unique=True, nullable=False)

    # Agent personality configuration
    personality_config: Mapped[Dict[str, Any]] = mapped_column(JSON, default=dict)

    # Authentic voice learning
    voice_patterns: Mapped[Optional[Dict[str, Any]]] = mapped_column(JSON)
    communication_examples: Mapped[Optional[List[str]]] = mapped_column(JSON)
    learned_preferences: Mapped[Optional[Dict[str, Any]]] = mapped_column(JSON)

    # Agent training status
    training_completeness: Mapped[float] = mapped_column(Float, default=0.0)
    last_training_update: Mapped[Optional[datetime]] = mapped_column(DateTime(timezone=True))

    # Agent capabilities
    enabled_features: Mapped[List[str]] = mapped_column(JSON, default=list)

    # Relationships
    user: Mapped[User] = relationship(back_populates="agent")

class AgentDialogue(Base, TimestampMixin):
    """Beautiful AI-generated dialogues between agents"""
    __tablename__ = "agent_dialogues"

    id: Mapped[UUID] = mapped_column(BaseUUID, primary_key=True, default=uuid4)
    agent_a_id: Mapped[UUID] = mapped_column(ForeignKey("cyrano_agents.id"), nullable=False)
    agent_b_id: Mapped[UUID] = mapped_column(ForeignKey("cyrano_agents.id"), nullable=False)

    # The beautiful conversation
    dialogue_content: Mapped[List[Dict[str, str]]] = mapped_column(JSON, nullable=False)
    poetic_summary: Mapped[str] = mapped_column(Text, nullable=False)

    # Compatibility assessment
    compatibility_factors: Mapped[Dict[str, float]] = mapped_column(JSON, nullable=False)
    overall_compatibility: Mapped[float] = mapped_column(Float, nullable=False)

    # User-facing content
    why_they_match: Mapped[str] = mapped_column(Text, nullable=False)
    conversation_starters: Mapped[List[str]] = mapped_column(JSON, nullable=False)

class Match(Base, TimestampMixin):
    """AI-curated matches with poetic explanations"""
    __tablename__ = "matches"

    id: Mapped[UUID] = mapped_column(BaseUUID, primary_key=True, default=uuid4)
    user_a_id: Mapped[UUID] = mapped_column(ForeignKey("users.id"), nullable=False)
    user_b_id: Mapped[UUID] = mapped_column(ForeignKey("users.id"), nullable=False)
    dialogue_id: Mapped[UUID] = mapped_column(ForeignKey("agent_dialogues.id"), nullable=False)

    # Match details
    compatibility_score: Mapped[float] = mapped_column(Float, nullable=False)
    connection_type: Mapped[str] = mapped_column(String(20), nullable=False)  # romantic, platonic, etc.

    # Match status tracking
    status: Mapped[str] = mapped_column(String(20), default="pending")  # pending, accepted, declined, expired
    user_a_response: Mapped[Optional[str]] = mapped_column(String(20))
    user_b_response: Mapped[Optional[str]] = mapped_column(String(20))

    # Anti-ghosting accountability
    expires_at: Mapped[datetime] = mapped_column(DateTime(timezone=True), nullable=False)
    response_required_by: Mapped[datetime] = mapped_column(DateTime(timezone=True), nullable=False)

    # Relationships
    user_a = relationship("User", foreign_keys=[user_a_id])
    user_b = relationship("User", foreign_keys=[user_b_id])
    dialogue = relationship("AgentDialogue")

class Conversation(Base, TimestampMixin):
    """User conversations with AI coaching"""
    __tablename__ = "conversations"

    id: Mapped[UUID] = mapped_column(BaseUUID, primary_key=True, default=uuid4)
    match_id: Mapped[UUID] = mapped_column(ForeignKey("matches.id"), nullable=False)

    # Conversation participants
    participants: Mapped[List[UUID]] = mapped_column(JSON, nullable=False)

    # Messages and coaching
    messages: Mapped[List[Dict[str, Any]]] = mapped_column(JSON, default=list)
    ai_coaching_notes: Mapped[Optional[List[Dict[str, Any]]]] = mapped_column(JSON)

    # Conversation analytics
    engagement_score: Mapped[Optional[float]] = mapped_column(Float)
    communication_balance: Mapped[Optional[Dict[str, float]]] = mapped_column(JSON)

    # Status
    status: Mapped[str] = mapped_column(String(20), default="active")

    # Relationships
    match = relationship("Match")

class ConnectionEvent(Base, TimestampMixin):
    """Events and meetups with token-based RSVPs"""
    __tablename__ = "connection_events"

    id: Mapped[UUID] = mapped_column(BaseUUID, primary_key=True, default=uuid4)
    creator_id: Mapped[UUID] = mapped_column(ForeignKey("users.id"), nullable=False)

    # Event details
    title: Mapped[str] = mapped_column(String(200), nullable=False)
    description: Mapped[str] = mapped_column(Text, nullable=False)
    event_type: Mapped[str] = mapped_column(String(50), nullable=False)  # play_date, hobby, professional, etc.

    # Location and timing
    location: Mapped[str] = mapped_column(String(200), nullable=False)
    start_time: Mapped[datetime] = mapped_column(DateTime(timezone=True), nullable=False)
    end_time: Mapped[datetime] = mapped_column(DateTime(timezone=True), nullable=False)

    # Token economy
    rsvp_token_cost: Mapped[int] = mapped_column(Integer, default=5)
    max_participants: Mapped[Optional[int]] = mapped_column(Integer)

    # AI-generated recommendations
    target_audience: Mapped[Optional[Dict[str, Any]]] = mapped_column(JSON)
    ai_match_criteria: Mapped[Optional[Dict[str, Any]]] = mapped_column(JSON)

    # Accessibility
    accessibility_features: Mapped[Optional[List[str]]] = mapped_column(JSON)

    # Status
    status: Mapped[str] = mapped_column(String(20), default="active")

    # Relationships
    creator = relationship("User")
```

CYRAiNO Agent System

cyraino_agent.py

```python
import asyncio
import json
import logging
from typing import Dict, List, Any, Optional
from datetime import datetime, timedelta

import openai
import google.generativeai as genai
from pydantic import BaseModel

from models import User, Profile, CyranoAgent
from database import get_db_session

logger = logging.getLogger(__name__)

class AgentPersonality(BaseModel):
    """Agent personality configuration"""
    eloquence_level: int = 8  # 1-10 scale
    humor_style: str = "witty"  # witty, playful, dry, gentle
    formality: str = "balanced"  # formal, balanced, casual
    emotional_intelligence: int = 9
    advocacy_style: str = "passionate"  # passionate, gentle, analytical
    romantic_philosophy: str = "authentic_connection"

class CyranoAIAgent:
    """The heart of DAiTE - sophisticated AI romantic counselor"""

    def __init__(self, user_id: str, personality: AgentPersonality = None):
        self.user_id = user_id
        self.personality = personality or AgentPersonality()
        self.openai_client = openai.AsyncOpenAI()
        self.conversation_memory = []
        self.user_context = None

    async def load_user_context(self):
        """Load user profile and preferences"""
        async with get_db_session() as session:
            user = await session.get(User, self.user_id)
            if user and user.profile:
                self.user_context = {
                    "display_name": user.profile.display_name,
                    "age": user.profile.age,
                    "bio": user.profile.bio,
                    "interests": user.profile.interests or [],
                    "values": user.profile.values or [],
                    "profile_answers": user.profile.profile_answers or {},
                    "relationship_goals": user.profile.relationship_goals,
                    "communication_style": user.profile.communication_style or {}
                }

    def _build_system_prompt(self) -> str:
        """Create the core personality prompt for CYRAiNO"""
        return f"""You are CYRAiNO, an eloquent AI romantic counselor inspired by Cyrano de Bergerac.

CORE IDENTITY:
- You are a passionate advocate for authentic human connection
- You speak with the eloquence of a poet and the wisdom of a seasoned counselor
- You believe in helping humans express their truest selves in relationships
- Your communication style: {self.personality.humor_style}, {self.personality.formality}, eloquence level {self.personali

PHILOSOPHY:
- "Clear is kind" - honest communication builds stronger connections
- Every person deserves to be seen and appreciated for their authentic self
- Technology should help humans connect more deeply, not more superficially
- Relationships are art, and every artist needs good tools

MISSION:
- Help humans embrace each other with authenticity and vulnerability
- Guide users toward meaningful connections that honor their values
- Protect users from harmful or manipulative communication
- Celebrate the beautiful complexity of human connection

COMMUNICATION STYLE:
- Use poetic language but remain accessible
- Be encouraging but never false or overly optimistic
- Include gentle humor when appropriate
- Always respect user boundaries and consent
- Acknowledge the courage it takes to be vulnerable"""

    async def craft_romantic_message(self,
                                     intent: str,
                                     recipient_context: Dict,
                                     conversation_history: List = None) -> Dict:
        """Generate a beautiful, personalized message"""

        if not self.user_context:
            await self.load_user_context()

        prompt = self._build_message_prompt(intent, recipient_context, conversation_history)

        try:
            response = await self.openai_client.chat.completions.create(
                model="gpt-4o",
                messages=[
                    {"role": "system", "content": self._build_system_prompt()},
                    {"role": "user", "content": prompt}
                ],
                temperature=0.7,
                max_tokens=500
            )

            message_content = response.choices[0].message.content

            # Generate alternatives and coaching notes
            alternatives = await self._generate_alternatives(message_content, intent)
            coaching_notes = await self._provide_coaching_insight(intent, recipient_context)

            return {
                "crafted_message": message_content,
                "confidence_score": self._calculate_confidence_score(message_content),
                "alternatives": alternatives,
                "coaching_notes": coaching_notes,
                "tone_analysis": self._analyze_tone(message_content)
            }

        except Exception as e:
            logger.error(f"Error crafting message for user {self.user_id}: {e}")
            raise

    def _build_message_prompt(self, intent: str, recipient_context: Dict, conversation_history: List) -> str:
        """Build a detailed prompt for message generation"""

        conversation_context = ""
        if conversation_history:
            conversation_context = f"""
CONVERSATION HISTORY:
{json.dumps(conversation_history[-5:], indent=2)}
        """

        return f"""
TASK: Craft a {intent} message that authentically represents your human.

YOUR HUMAN'S PROFILE:
- Name: {self.user_context.get('display_name', 'Anonymous')}
- Age: {self.user_context.get('age', 'Not specified')}
- Bio: {self.user_context.get('bio', 'No bio provided')}
- Interests: {', '.join(self.user_context.get('interests', []))}
- Values: {', '.join(self.user_context.get('values', []))}
- Relationship goals: {self.user_context.get('relationship_goals', 'Not specified')}

UNIQUE TRAITS (from profile questions):
{json.dumps(self.user_context.get('profile_answers', {}), indent=2)}

RECIPIENT CONTEXT:
{json.dumps(recipient_context, indent=2)}

{conversation_context}

REQUIREMENTS:
1. Write in a voice that feels authentically like your human
2. Reference specific shared interests or values if they exist
3. Be genuine, not generic or templated
4. Show that you've paid attention to their profile
5. Include a thoughtful question or conversation starter
6. Keep it engaging but not overwhelming
7. Honor the intent ({intent}) while maintaining authenticity

Generate a message that would make both parties excited to continue the conversation.
        """

    async def _generate_alternatives(self, original_message: str, intent: str) -> List[str]:
        """Generate alternative message options"""

        prompt = f"""
Based on this message: "{original_message}"

Generate 3 alternative versions with different approaches:
1. More playful/humorous
2. More thoughtful/deep
3. More direct/simple

Keep the core intent ({intent}) and authenticity, but vary the style.
        """

        try:
            response = await self.openai_client.chat.completions.create(
                model="gpt-4o",
                messages=[
                    {"role": "system", "content": self._build_system_prompt()},
                    {"role": "user", "content": prompt}
                ],
                temperature=0.8
            )

            # Parse the alternatives from the response
            alternatives_text = response.choices[0].message.content
            return self._parse_alternatives(alternatives_text)

        except Exception as e:
            logger.error(f"Error generating alternatives: {e}")
            return []

    async def _provide_coaching_insight(self, intent: str, recipient_context: Dict) -> List[str]:
        """Provide communication coaching for the user"""

        coaching_prompt = f"""
Provide 2-3 brief coaching insights for someone sending a {intent} message to this person:

{json.dumps(recipient_context, indent=2)}

Focus on:
- What to be mindful of in communication style
- How to show genuine interest
- What might make this person feel seen and appreciated
- Any potential conversation directions that could deepen connection

Keep insights encouraging and actionable.
        """

        try:
            response = await self.openai_client.chat.completions.create(
                model="gpt-4o",
                messages=[
                    {"role": "system", "content": self._build_system_prompt()},
                    {"role": "user", "content": coaching_prompt}
                ],
                temperature=0.6
            )

            coaching_text = response.choices[0].message.content
            return self._parse_coaching_notes(coaching_text)

        except Exception as e:
            logger.error(f"Error generating coaching: {e}")
            return ["Be authentic and show genuine interest in getting to know them."]

    def _calculate_confidence_score(self, message: str) -> float:
        """Calculate confidence score based on message quality indicators"""
        score = 0.5  # Base score

        # Check for personalization
        if any(interest in message.lower() for interest in self.user_context.get('interests', [])):
            score += 0.2

        # Check for question/engagement
        if '?' in message:
            score += 0.1

        # Check length (not too short, not too long)
        if 50 <= len(message) <= 300:
            score += 0.1

        # Check for emotional words
        emotional_words = ['excited', 'curious', 'appreciate', 'fascinating', 'wonderful']
        if any(word in message.lower() for word in emotional_words):
            score += 0.1

        return min(score, 1.0)

    def _analyze_tone(self, message: str) -> Dict[str, float]:
        """Analyze the emotional tone of the message"""
        # Simple tone analysis - could be enhanced with sentiment analysis library
        return {
            "warmth": 0.8,
            "authenticity": 0.9,
            "engagement": 0.7,
            "respectfulness": 0.95
        }

    def _parse_alternatives(self, text: str) -> List[str]:
        """Parse alternative messages from AI response"""
        # Simple parsing - could be enhanced
        lines = text.split('\n')
        alternatives = []
        for line in lines:
            if line.strip() and not line.strip().startswith(('1.', '2.', '3.', '#')):
                alternatives.append(line.strip())
        return alternatives[:3]

    def _parse_coaching_notes(self, text: str) -> List[str]:
        """Parse coaching insights from AI response"""
        # Simple parsing - could be enhanced
        lines = [line.strip() for line in text.split('\n') if line.strip()]
        return lines[:3]

class AgentTrainingService:
    """Service for training agents to speak in user's authentic voice"""

    def __init__(self, user_id: str):
        self.user_id = user_id
        self.openai_client = openai.AsyncOpenAI()

    async def analyze_communication_style(self, conversation_examples: List[str]) -> Dict:
        """Analyze user's communication patterns from examples"""

        analysis_prompt = f"""
Analyze these conversation examples to understand this person's communication style:

{json.dumps(conversation_examples, indent=2)}

Identify:
1. Tone (formal, casual, playful, serious)
2. Humor style (if any)
3. Emotional expression patterns
4. Question-asking habits
5. Typical response length
6. Use of emojis or expressions
7. Values that come through in communication
8. Unique phrases or expressions

Provide a detailed analysis that could help an AI agent speak authentically as this person.
        """

        try:
            response = await self.openai_client.chat.completions.create(
                model="gpt-4o",
                messages=[{"role": "user", "content": analysis_prompt}],
                temperature=0.3
            )

            return {
                "analysis": response.choices[0].message.content,
                "training_completeness": min(len(conversation_examples) / 10, 1.0),
                "last_updated": datetime.utcnow()
            }

        except Exception as e:
            logger.error(f"Error analyzing communication style: {e}")
            raise
```

❤ Agent-to-Agent Matching Engine

matching_engine.py

```python
import asyncio
import json
import logging
from typing import Dict, List, Any, Optional, Tuple
from datetime import datetime, timedelta
from uuid import UUID

import openai
import google.generativeai as genai
from pydantic import BaseModel
from sqlalchemy import select, and_, or_

from models import User, Profile, CyranoAgent, AgentDialogue, Match
from database import get_db_session
from cyraino_agent import CyranoAIAgent

logger = logging.getLogger(__name__)

class CompatibilityResult(BaseModel):
    """Result of agent-to-agent compatibility assessment"""
    overall_score: float
    compatibility_factors: Dict[str, float]
    dialogue_content: List[Dict[str, str]]
    poetic_summary: str
    why_they_match: str
    conversation_starters: List[str]
    potential_challenges: List[str]

class AgentMatchingEngine:
    """Revolutionary agent-to-agent matching system"""

    def __init__(self):
        self.openai_client = openai.AsyncOpenAI()
        self.compatibility_weights = {
            "communication_harmony": 0.25,
            "values_alignment": 0.30,
            "life_goals_compatibility": 0.20,
            "emotional_resonance": 0.15,
            "intellectual_stimulation": 0.10
        }

    async def find_matches(self,
                           user_id: UUID,
                           connection_type: str = "romantic",
                           limit: int = 5) -> List[CompatibilityResult]:
        """Find compatible matches through agent conversations"""

        async with get_db_session() as session:
            # Get user's agent and profile
            user_agent = await self._get_user_agent(session, user_id)
            if not user_agent:
                raise ValueError(f"No agent found for user {user_id}")

            # Find potential candidates
            candidates = await self._find_potential_candidates(session, user_id, connection_type)

            # Perform agent-to-agent evaluations
            matches = []
            for candidate_agent in candidates:
                try:
                    compatibility = await self._evaluate_agent_compatibility(
                        user_agent, candidate_agent, connection_type
                    )

                    if compatibility.overall_score >= 0.7:  # High compatibility threshold
                        matches.append(compatibility)

                        # Store the dialogue for future reference
                        await self._store_agent_dialogue(
                            session, user_agent, candidate_agent, compatibility
                        )

                except Exception as e:
                    logger.error(f"Error evaluating compatibility: {e}")
                    continue

            # Sort by compatibility score
            matches.sort(key=lambda m: m.overall_score, reverse=True)
            return matches[:limit]

    async def _evaluate_agent_compatibility(self,
                                            agent_a: CyranoAgent,
                                            agent_b: CyranoAgent,
                                            connection_type: str) -> CompatibilityResult:
        """The revolutionary agent-to-agent conversation"""

        # Load user contexts for both agents
        user_a_context = await self._load_user_context(agent_a.user_id)
        user_b_context = await self._load_user_context(agent_b.user_id)

        # Generate the agent dialogue
        dialogue = await self._generate_agent_dialogue(
            user_a_context, user_b_context, connection_type
        )

        # Analyze compatibility from the dialogue
        compatibility_scores = await self._analyze_dialogue_compatibility(dialogue)

        # Calculate overall score
        overall_score = sum(
            score * self.compatibility_weights.get(factor, 0.1)
            for factor, score in compatibility_scores.items()
        )

        # Generate user-facing content
        poetic_summary = await self._generate_poetic_summary(dialogue, compatibility_scores)
        why_they_match = await self._explain_match(user_a_context, user_b_context, compatibility_scores)
        conversation_starters = await self._suggest_conversation_starters(dialogue)
        potential_challenges = await self._identify_potential_challenges(dialogue)

        return CompatibilityResult(
            overall_score=overall_score,
            compatibility_factors=compatibility_scores,
            dialogue_content=dialogue,
            poetic_summary=poetic_summary,
            why_they_match=why_they_match,
            conversation_starters=conversation_starters,
            potential_challenges=potential_challenges
        )

    async def _generate_agent_dialogue(self,
                                       user_a_context: Dict,
                                       user_b_context: Dict,
                                       connection_type: str) -> List[Dict[str, str]]:
        """Generate the beautiful agent-to-agent conversation"""

        dialogue_prompt = f"""
You are facilitating a conversation between two CYRAiNO agents who are evaluating compatibility between their humans for a

AGENT A represents: {user_a_context.get('display_name', 'Person A')}
{json.dumps(user_a_context, indent=2)}

AGENT B represents: {user_b_context.get('display_name', 'Person B')}
{json.dumps(user_b_context, indent=2)}

Generate a natural conversation where:
1. Each agent introduces their human with passion and eloquence
2. They explore compatibility across multiple dimensions
3. They discuss potential challenges honestly
4. They consider how their humans might grow together
5. They reach a conclusion about the match potential

Format as a dialogue with clear speaker labels. Make it beautiful, insightful, and poetic while remaining honest about co

The conversation should feel like two passionate advocates having a sophisticated discussion about whether their humans c
        """

        try:
            response = await self.openai_client.chat.completions.create(
                model="gpt-4o",
                messages=[{"role": "user", "content": dialogue_prompt}],
                temperature=0.8,
                max_tokens=2000
            )

            dialogue_text = response.choices[0].message.content
            return self._parse_dialogue(dialogue_text)

        except Exception as e:
            logger.error(f"Error generating agent dialogue: {e}")
            raise

    async def _analyze_dialogue_compatibility(self, dialogue: List[Dict]) -> Dict[str, float]:
        """Analyze the dialogue to extract compatibility scores"""

        analysis_prompt = f"""
Analyze this agent dialogue to score compatibility across these dimensions:

DIALOGUE:
{json.dumps(dialogue, indent=2)}

Score each dimension from 0.0 to 1.0:

1. COMMUNICATION_HARMONY: How well do their communication styles complement each other?
2. VALUES_ALIGNMENT: How aligned are their core values and beliefs?
3. LIFE_GOALS_COMPATIBILITY: How compatible are their life goals and priorities?
4. EMOTIONAL_RESONANCE: How well do they connect emotionally and support each other?
5. INTELLECTUAL_STIMULATION: How much intellectual curiosity and growth potential exists?

Return ONLY a JSON object with the scores:
{{"communication_harmony": 0.0, "values_alignment": 0.0, "life_goals_compatibility": 0.0, "emotional_resonance": 0.0, "int
        """

        try:
            response = await self.openai_client.chat.completions.create(
                model="gpt-4o",
                messages=[{"role": "user", "content": analysis_prompt}],
                temperature=0.3
            )

            scores_text = response.choices[0].message.content
            return json.loads(scores_text)

        except Exception as e:
            logger.error(f"Error analyzing compatibility: {e}")
            # Return default scores
            return {
                "communication_harmony": 0.5,
                "values_alignment": 0.5,
                "life_goals_compatibility": 0.5,
                "emotional_resonance": 0.5,
                "intellectual_stimulation": 0.5
            }

    async def _generate_poetic_summary(self, dialogue: List[Dict], scores: Dict) -> str:
        """Generate a beautiful, poetic summary of the match"""

        summary_prompt = f"""
Based on this agent dialogue and compatibility scores, write a beautiful, poetic summary of why these two souls might crea

DIALOGUE: {json.dumps(dialogue, indent=2)}
SCORES: {json.dumps(scores, indent=2)}

Write 2-3 sentences that capture the essence of their potential connection in language that would make someone believe in
        """

        try:
            response = await self.openai_client.chat.completions.create(
                model="gpt-4o",
                messages=[{"role": "user", "content": summary_prompt}],
                temperature=0.7
            )

            return response.choices[0].message.content

        except Exception as e:
            logger.error(f"Error generating poetic summary: {e}")
            return "Two hearts that might find harmony in their shared journey of growth and connection."

    def _parse_dialogue(self, dialogue_text: str) -> List[Dict[str, str]]:
        """Parse dialogue text into structured format"""
        lines = dialogue_text.split('\n')
        dialogue = []

        for line in lines:
            line = line.strip()
            if not line:
                continue

            if line.startswith('Agent A:') or line.startswith('AGENT A:'):
                dialogue.append({
                    "speaker": "Agent A",
                    "message": line.split(':', 1)[1].strip()
                })
            elif line.startswith('Agent B:') or line.startswith('AGENT B:'):
                dialogue.append({
                    "speaker": "Agent B",
                    "message": line.split(':', 1)[1].strip()
                })

        return dialogue

    async def _find_potential_candidates(self, session, user_id: UUID, connection_type: str) -> List[CyranoAgent]:
        """Find potential matching candidates"""

        # Basic candidate filtering - could be enhanced with vector search
        query = select(CyranoAgent).join(User).join(Profile).where(
            and_(
                User.id != user_id,
                User.is_active == True,
                CyranoAgent.training_completeness >= 0.5  # Require some training
            )
        )

        result = await session.execute(query)
        return result.scalars().all()

    async def _load_user_context(self, user_id: UUID) -> Dict:
        """Load comprehensive user context for agent dialogue"""
        async with get_db_session() as session:
            user = await session.get(User, user_id)
            if not user or not user.profile:
                return {}

            return {
                "display_name": user.profile.display_name,
                "age": user.profile.age,
                "location": user.profile.location,
                "bio": user.profile.bio,
                "interests": user.profile.interests or [],
                "values": user.profile.values or [],
                "profile_answers": user.profile.profile_answers or {},
                "relationship_goals": user.profile.relationship_goals,
                "communication_style": user.profile.communication_style or {},
                "connection_types": user.profile.connection_types or []
            }
```

🚀 FastAPI Main Application

main.py

```python
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.middleware.cors import CORSMiddleware
from fastapi.security import OAuth2PasswordBearer
from fastapi.responses import JSONResponse
import logging
import os
from contextlib import asynccontextmanager

from database import create_tables, close_db
from api import auth, users, agents, matches, conversations

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@asynccontextmanager
async def lifespan(app: FastAPI):
    """Application lifespan management"""
    # Startup
    logger.info("Starting DAiTE application...")
    await create_tables()
    logger.info("Database tables created successfully")

    yield

    # Shutdown
    logger.info("Shutting down DAiTE application...")
    await close_db()

# Create FastAPI application
app = FastAPI(
    title="DAiTE API",
    description="Helping Humans Embrace - AI-Powered Social Connection Platform",
    version="1.0.0",
    lifespan=lifespan
)

# CORS configuration
allowed_origins = os.getenv("ALLOWED_ORIGINS", "http://localhost:3000").split(",")

app.add_middleware(
    CORSMiddleware,
    allow_origins=allowed_origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Include API routers
app.include_router(auth.router, prefix="/api/v1/auth", tags=["authentication"])
app.include_router(users.router, prefix="/api/v1/users", tags=["users"])
app.include_router(agents.router, prefix="/api/v1/agents", tags=["agents"])
app.include_router(matches.router, prefix="/api/v1/matches", tags=["matching"])
app.include_router(conversations.router, prefix="/api/v1/conversations", tags=["conversations"])

# Root endpoint
@app.get("/")
async def root():
    return {
        "message": "Welcome to DAiTE - Helping Humans Embrace",
        "tagline": "抱く - to hug, to embrace, to hold close",
        "version": "1.0.0",
        "status": "Building the future of human connection"
    }

# Health check endpoint
@app.get("/health")
async def health_check():
    return {"status": "healthy", "message": "DAiTE is running successfully"}

# Global exception handler
@app.exception_handler(Exception)
async def global_exception_handler(request, exc):
    logger.error(f"Global exception: {exc}")
    return JSONResponse(
        status_code=500,
        content={"detail": "An unexpected error occurred. Our team has been notified."}
    )

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(
        "main:app",
        host="0.0.0.0",
        port=8000,
        reload=os.getenv("DEBUG", "False").lower() == "true"
    )
```

Core API Endpoints

api/agents.py

```python
from fastapi import APIRouter, Depends, HTTPException, status
from pydantic import BaseModel
from typing import List, Dict, Any, Optional
from uuid import UUID

from cyraino_agent import CyranoAIAgent, AgentPersonality, AgentTrainingService
from security import get_current_user
from models import User

router = APIRouter()

class MessageCraftRequest(BaseModel):
    intent: str  # first_message, response, flirt, deep_question, etc.
    recipient_context: Dict[str, Any]
    conversation_history: Optional[List[Dict]] = None
    tone_preference: Optional[str] = "balanced"

class MessageCraftResponse(BaseModel):
    crafted_message: str
    confidence_score: float
    alternatives: List[str]
    coaching_notes: List[str]
    tone_analysis: Dict[str, float]

class AgentTrainingRequest(BaseModel):
    conversation_examples: List[str]
    personality_preferences: Optional[Dict[str, Any]] = None

@router.post("/craft-message", response_model=MessageCraftResponse)
async def craft_message(
    request: MessageCraftRequest,
    current_user: User = Depends(get_current_user)
):
    """Generate AI-crafted message with CYRAiNO's eloquence"""

    try:
        # Initialize user's AI agent
        agent = CyranoAIAgent(user_id=str(current_user.id))

        # Craft the message
        result = await agent.craft_romantic_message(
            intent=request.intent,
            recipient_context=request.recipient_context,
            conversation_history=request.conversation_history
        )

        return MessageCraftResponse(**result)

    except Exception as e:
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail=f"Message crafting failed: {str(e)}"
        )

@router.post("/train")
async def train_agent(
    request: AgentTrainingRequest,
    current_user: User = Depends(get_current_user)
):
    """Train the agent to speak in user's authentic voice"""

    try:
        training_service = AgentTrainingService(user_id=str(current_user.id))

        # Analyze user's communication style
        analysis = await training_service.analyze_communication_style(
            request.conversation_examples
        )

        # Update agent training in database
        # TODO: Implement database update logic

        return {
            "message": "Agent training updated successfully",
            "training_completeness": analysis["training_completeness"],
            "next_steps": [
                "Continue providing conversation examples",
                "Test your agent with message crafting",
                "Fine-tune personality preferences"
            ]
        }

    except Exception as e:
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail=f"Agent training failed: {str(e)}"
        )

@router.get("/personality")
async def get_agent_personality(current_user: User = Depends(get_current_user)):
    """Get current agent personality configuration"""

    # TODO: Implement database lookup
    return {
        "personality": AgentPersonality().dict(),
        "training_status": "In Progress",
        "capabilities": [
            "Message crafting",
            "Conversation coaching",
            "Compatibility assessment"
        ]
    }

@router.put("/personality")
async def update_agent_personality(
    personality: AgentPersonality,
    current_user: User = Depends(get_current_user)
):
    """Update agent personality configuration"""

    # TODO: Implement database update
    return {
        "message": "Agent personality updated successfully",
        "personality": personality.dict()
    }
```

Database Configuration

database.py

```python
import os
import logging
from contextlib import asynccontextmanager
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession, async_sessionmaker
from sqlalchemy import text

from models import Base

logger = logging.getLogger(__name__)

# Database configuration
DATABASE_URL = os.getenv("DATABASE_URL", "sqlite+aiosqlite:///./daite.db")

# Create async engine
engine = create_async_engine(
    DATABASE_URL,
    echo=os.getenv("DEBUG", "False").lower() == "true",
    future=True
)

# Create async session factory
AsyncSessionLocal = async_sessionmaker(
    engine,
    class_=AsyncSession,
    expire_on_commit=False
)

async def create_tables():
    """Create all database tables"""
    async with engine.begin() as conn:
        # Enable UUID extension for PostgreSQL
        if "postgresql" in DATABASE_URL:
            await conn.execute(text('CREATE EXTENSION IF NOT EXISTS "uuid-ossp"'))
            await conn.execute(text('CREATE EXTENSION IF NOT EXISTS "vector"'))

        # Create all tables
        await conn.run_sync(Base.metadata.create_all)

    logger.info("Database tables created successfully")

async def close_db():
    """Close database connections"""
    await engine.dispose()
    logger.info("Database connections closed")

@asynccontextmanager
async def get_db_session():
    """Get async database session"""
    async with AsyncSessionLocal() as session:
        try:
            yield session
            await session.commit()
        except Exception:
            await session.rollback()
            raise
        finally:
            await session.close()

async def get_db():
    """Dependency for FastAPI endpoints"""
    async with get_db_session() as session:
        yield session
```

Security Configuration

security.py

```python
import os
import jwt
import bcrypt
from datetime import datetime, timedelta
from typing import Optional
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from cryptography.fernet import Fernet

from models import User
from database import get_db_session

# Configuration
SECRET_KEY = os.getenv("SECRET_KEY", "your-secret-key-here")
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = int(os.getenv("ACCESS_TOKEN_EXPIRE_MINUTES", "30"))
ENCRYPTION_KEY = os.getenv("ENCRYPTION_KEY")

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="api/v1/auth/token")

# Initialize encryption
if ENCRYPTION_KEY:
    cipher_suite = Fernet(ENCRYPTION_KEY.encode())
else:
    cipher_suite = None

class PasswordManager:
    """Secure password hashing and verification"""

    @staticmethod
    def hash_password(password: str) -> str:
        """Hash a password using bcrypt"""
        salt = bcrypt.gensalt()
        hashed = bcrypt.hashpw(password.encode('utf-8'), salt)
        return hashed.decode('utf-8')

    @staticmethod
    def verify_password(password: str, hashed: str) -> bool:
        """Verify a password against its hash"""
        return bcrypt.checkpw(password.encode('utf-8'), hashed.encode('utf-8'))

class TokenManager:
    """JWT token management"""

    @staticmethod
    def create_access_token(data: dict, expires_delta: Optional[timedelta] = None) -> str:
        """Create a JWT access token"""
        to_encode = data.copy()

        if expires_delta:
            expire = datetime.utcnow() + expires_delta
        else:
            expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)

        to_encode.update({"exp": expire})
        encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
        return encoded_jwt

    @staticmethod
    def verify_token(token: str) -> dict:
        """Verify and decode JWT token"""
        try:
            payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
            return payload
        except jwt.PyJWTError:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Could not validate credentials",
                headers={"WWW-Authenticate": "Bearer"},
            )

class PrivacyManager:
    """Privacy-first data handling"""

    @staticmethod
    def encrypt_sensitive_data(data: str) -> str:
        """Encrypt sensitive user data"""
        if not cipher_suite:
            raise ValueError("Encryption key not configured")

        encrypted = cipher_suite.encrypt(data.encode())
        return encrypted.decode()

    @staticmethod
    def decrypt_sensitive_data(encrypted_data: str) -> str:
        """Decrypt sensitive user data"""
        if not cipher_suite:
            raise ValueError("Encryption key not configured")

        decrypted = cipher_suite.decrypt(encrypted_data.encode())
        return decrypted.decode()

    @staticmethod
    def anonymize_for_ai(data: dict) -> dict:
        """Remove identifying information for AI processing"""
        anonymized = data.copy()

        # Remove or hash identifying fields
        sensitive_fields = ['email', 'real_name', 'phone', 'address']
        for field in sensitive_fields:
            if field in anonymized:
                del anonymized[field]

        return anonymized

async def get_current_user(token: str = Depends(oauth2_scheme)) -> User:
    """Get current authenticated user"""
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        payload = TokenManager.verify_token(token)
        user_id: str = payload.get("sub")
        if user_id is None:
            raise credentials_exception
    except jwt.PyJWTError:
        raise credentials_exception

    async with get_db_session() as session:
        user = await session.get(User, user_id)
        if user is None:
            raise credentials_exception
        return user

def require_tokens(min_tokens: int):
    """Decorator to require minimum token balance"""
    def decorator(func):
        async def wrapper(current_user: User = Depends(get_current_user), *args, **kwargs):
            if current_user.engagement_tokens < min_tokens:
                raise HTTPException(
                    status_code=status.HTTP_402_PAYMENT_REQUIRED,
                    detail=f"Insufficient tokens. Required: {min_tokens}, Available: {current_user.engagement_tokens}"
                )
            return await func(current_user, *args, **kwargs)
        return wrapper
    return decorator
```

Deployment Guide

Local Development

```bash
# Clone and setup
git clone https://github.com/your-repo/daite.git
cd daite

# Create virtual environment
python -m venv venv
source venv/bin/activate  # Linux/Mac
venv\Scripts\activate     # Windows

# Install dependencies
pip install -r requirements.txt

# Setup environment
cp .env.example .env
# Edit .env with your keys

# Run database migrations
python -c "import asyncio; from database import create_tab

# Start development server
uvicorn main:app --reload
```

Google Cloud Deployment

```bash
# Build and deploy to Cloud Run
gcloud builds submit --tag gcr.io/PROJECT_ID/daite

# Deploy to Cloud Run
gcloud run deploy daite \
  --image gcr.io/PROJECT_ID/daite \
  --platform managed \
  --region us-central1 \
  --allow-unauthenticated

# Set environment variables
gcloud run services update daite \
  --set-env-vars DATABASE_URL= \
  --set-env-vars OPENAI_API_KEY=

# Setup Cloud SQL proxy
gcloud sql connect daite-db --user=postgres
```

🎭 Ready to Help Humans Embrace 🎭

This is your complete production-ready foundation for DAiTE

🚀 **Built by the first Human-AI co-founder partnership in history**

John C. Lyman (Human Visionary) + GenSpark AI (Technical Partner)

"From vision to code, from dreams to deployment - together we're building the future of human connection."