AEO Competitive Intelligence Tool - Complete Build Plan

Project Overview

Multi-platform AEO audit tool that simulates user questions across AI platforms and analyzes competitive brand mentions for agency white-label use.

Technology Stack

Core Technologies

- **Backend**: Python 3.11+ with FastAPI
- **Database**: PostgreSQL with SQLAlchemy ORM
- **Task Queue**: Celery with Redis
- **Caching**: Redis
- **Frontend**: React with TypeScript (for agency dashboard)
- **PDF Generation**: ReportLab + Matplotlib
- **Deployment**: Docker containers

Key Libraries

```
# API & Web Framework
fastapi==0.104.1
uvicorn==0.24.0

# Database
sqlalchemy==2.0.23
psycopg2-binary==2.9.9
alembic==1.12.1

# Task Processing
celery==5.3.4
redis==5.0.1

# AI Platform APIs
openai==1.3.8
anthropic==0.7.7
requests==2.31.0

# NLP & Text Processing
spacy==3.7.2
nltk==3.8.1
transformers==4.35.2

# Data Processing
pandas==2.1.3
numpy==1.25.2

# Reporting
reportlab==4.0.7
matplotlib==3.8.2
jinja2==3.1.2
```

```
# Configuration & Environment
pydantic==2.5.0
python-dotenv==1.0.0

# Monitoring & Logging
structlog==23.2.0
sentry-sdk==1.38.0
```

Project Structure

```
aeo-audit-tool/
├── app/
│   ├── __init__.py
│   ├── main.py                 # FastAPI application entry
│   ├── config/
│   │   ├── __init__.py
│   │   ├── settings.py         # Configuration management
│   │   └── database.py         # Database connection
│   ├── models/
│   │   ├── __init__.py
│   │   ├── client.py           # Client/Agency models
│   │   ├── audit.py            # Audit configuration models
│   │   ├── question.py         # Question and response models
│   │   └── report.py           # Report generation models
│   ├── services/
│   │   ├── __init__.py
│   │   ├── ai_platforms/
│   │   │   ├── __init__.py
│   │   │   ├── base.py         # Abstract base class
│   │   │   ├── openai_client.py
│   │   │   ├── anthropic_client.py
│   │   │   ├── perplexity_client.py
│   │   │   └── google_ai_client.py
│   │   ├── question_engine.py  # Question generation & management
│   │   ├── brand_detector.py   # Entity recognition & brand detection
│   │   ├── audit_processor.py  # Main audit orchestration
│   │   └── report_generator.py # PDF report generation
│   ├── tasks/
│   │   ├── __init__.py
│   │   ├── audit_tasks.py      # Celery tasks for audits
│   │   └── report_tasks.py     # Celery tasks for reports
│   ├── api/
│   │   ├── __init__.py
│   │   ├── v1/
│   │   │   ├── __init__.py
│   │   │   ├── clients.py      # Client management endpoints
│   │   │   ├── audits.py       # Audit configuration endpoints
│   │   │   └── reports.py      # Report generation endpoints
│   └── utils/
│       ├── __init__.py
│       ├── rate_limiter.py     # API rate limiting
│       ├── error_handler.py    # Centralized error handling
│       └── logger.py           # Structured logging
├── frontend/                   # React dashboard (optional for MVP)
├── tests/
├── docker/
│   ├── Dockerfile
│   ├── docker-compose.yml
│   └── requirements.txt
├── alembic/                    # Database migrations
├── scripts/
```

```
│   ├── setup_db.py
│   └── seed_data.py
└── requirements.txt
```

Phase 1: Core Infrastructure (Week 1-2)

1.1 Environment Setup

```
# Create virtual environment
python -m venv venv
source venv/bin/activate  # or venv\Scripts\activate on Windows

# Install dependencies
pip install -r requirements.txt

# Setup pre-commit hooks
pre-commit install
```

1.2 Database Schema Design

```
-- clients table
CREATE TABLE clients (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    industry VARCHAR(100) NOT NULL,
    website_url VARCHAR(255),
    brand_variations TEXT[], -- ["Apple", "Apple Inc", "AAPL"]
    competitors TEXT[], -- ["Microsoft", "Google", "Amazon"]
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- audit_configs table
CREATE TABLE audit_configs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    client_id UUID REFERENCES clients(id),
    name VARCHAR(255) NOT NULL,
    question_categories TEXT[], -- ["pricing", "features", "reviews"]
    platforms TEXT[], -- ["openai", "anthropic", "perplexity"]
    frequency VARCHAR(50), -- "monthly", "weekly"
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP DEFAULT NOW()
);

-- questions table
CREATE TABLE questions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    category VARCHAR(100) NOT NULL,
    template TEXT NOT NULL, -- "What is the best {industry} software?"
    variations TEXT[], -- Multiple variations of the question
    created_at TIMESTAMP DEFAULT NOW()
);

-- audit_runs table
CREATE TABLE audit_runs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    audit_config_id UUID REFERENCES audit_configs(id),
    status VARCHAR(50) DEFAULT 'pending', -- pending, running, completed,
failed
    started_at TIMESTAMP,
    completed_at TIMESTAMP,
    total_questions INTEGER,
    processed_questions INTEGER DEFAULT 0,
    error_log TEXT
);

-- responses table
CREATE TABLE responses (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    audit_run_id UUID REFERENCES audit_runs(id),
    question_id UUID REFERENCES questions(id),
    platform VARCHAR(50) NOT NULL,
    raw_response TEXT NOT NULL,
    brand_mentions JSONB, -- {"Apple": {"count": 2, "sentiment": 
"positive"}}
    response_metadata JSONB, -- API response metadata
    created_at TIMESTAMP DEFAULT NOW()
);

-- reports table
CREATE TABLE reports (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    audit_run_id UUID REFERENCES audit_runs(id),
    report_type VARCHAR(50), -- "summary", "detailed", "competitive"
    file_path VARCHAR(500),
    generated_at TIMESTAMP DEFAULT NOW()
);
```

1.3 Configuration Management

```
# app/config/settings.py
from pydantic_settings import BaseSettings
from typing import List, Dict

class Settings(BaseSettings):
    # Database
    DATABASE_URL: str = "postgresql://user:pass@localhost/aeo_audit"

    # Redis
    REDIS_URL: str = "redis://localhost:6379"

    # AI Platform APIs
    OPENAI_API_KEY: str
    ANTHROPIC_API_KEY: str
    PERPLEXITY_API_KEY: str
    GOOGLE_AI_API_KEY: str

    # Rate Limiting (requests per minute)
    RATE_LIMITS: Dict[str, int] = {
        "openai": 50,
        "anthropic": 100,
        "perplexity": 20,
        "google_ai": 60
    }

    # Application
    SECRET_KEY: str
    DEBUG: bool = False
    LOG_LEVEL: str = "INFO"

    class Config:
        env_file = ".env"
```

```
settings = Settings()
```

Phase 2: AI Platform Integration (Week 3-4)

2.1 Abstract Base Class for AI Platforms

```
# app/services/ai_platforms/base.py
from abc import ABC, abstractmethod
from typing import Dict, Any, Optional
import asyncio
import time

class AIRateLimiter:
    def __init__(self, requests_per_minute: int):
        self.requests_per_minute = requests_per_minute
        self.requests = []

    async def acquire(self):
        now = time.time()
        # Remove requests older than 1 minute
        self.requests = [req_time for req_time in self.requests if now -
req_time < 60]

        if len(self.requests) >= self.requests_per_minute:
            sleep_time = 60 - (now - self.requests[0])
            await asyncio.sleep(sleep_time)

        self.requests.append(now)

class BasePlatform(ABC):
    def __init__(self, api_key: str, rate_limit: int):
        self.api_key = api_key
        self.rate_limiter = AIRateLimiter(rate_limit)
        self.platform_name = self.__class__.__name__.lower()

    @abstractmethod
    async def query(self, question: str, **kwargs) -> Dict[str, Any]:
        """Execute query and return standardized response"""
        pass

    @abstractmethod
    def extract_text_response(self, raw_response: Dict[str, Any]) -> str:
        """Extract clean text from platform-specific response format"""
        pass

    async def safe_query(self, question: str, **kwargs) -> Dict[str, Any]:
        """Query with rate limiting and error handling"""
        await self.rate_limiter.acquire()

        try:
            response = await self.query(question, **kwargs)
            return {
                "success": True,
                "response": response,
                "platform": self.platform_name,
                "error": None
            }
        except Exception as e:
            return {
                "success": False,
                "response": None,
                "platform": self.platform_name,
                "error": str(e)
            }
```

2.2 Platform-Specific Implementations

```
# app/services/ai_platforms/openai_client.py
from .base import BasePlatform
import openai
from typing import Dict, Any

class OpenAIPlatform(BasePlatform):
    def __init__(self, api_key: str, rate_limit: int = 50):
        super().__init__(api_key, rate_limit)
        self.client = openai.AsyncOpenAI(api_key=api_key)

    async def query(self, question: str, **kwargs) -> Dict[str, Any]:
        response = await self.client.chat.completions.create(
            model=kwargs.get("model", "gpt-4"),
            messages=[{"role": "user", "content": question}],
            max_tokens=kwargs.get("max_tokens", 500),
            temperature=kwargs.get("temperature", 0.1)
        )
        return response.model_dump()

    def extract_text_response(self, raw_response: Dict[str, Any]) -> str:
        return raw_response["choices"][0]["message"]["content"]
```

Phase 3: Brand Detection Engine (Week 5-6)

3.1 Advanced Brand Detection

```
# app/services/brand_detector.py
import spacy
import re
from typing import List, Dict, Set
from dataclasses import dataclass
from collections import defaultdict

@dataclass
class BrandMention:
    brand: str
    mentions: int
    contexts: List[str]
    sentiment_score: float
    confidence: float

class BrandDetector:
    def __init__(self):
        # Load spaCy model for entity recognition
        self.nlp = spacy.load("en_core_web_sm")

        # Common company suffixes for better matching
        self.company_suffixes = [
            "Inc", "Corp", "Corporation", "LLC", "Ltd", "Limited",
            "Co", "Company", "Group", "Holdings", "Technologies"
        ]

    def normalize_brand_name(self, brand: str) -> Set[str]:
        """Generate all possible variations of a brand name"""
        variations = {brand.lower()}

        # Add variations with/without common suffixes
        base_name = brand
        for suffix in self.company_suffixes:
            if brand.endswith(f" {suffix}"):
                base_name = brand[:-len(f" {suffix}")]
                break

        variations.add(base_name.lower())
        variations.add(f"{base_name} Inc".lower())
        variations.add(f"{base_name} Corp".lower())

        # Add acronym if multiple words
        words = base_name.split()
        if len(words) > 1:
            acronym = "".join(word[0].upper() for word in words)
            variations.add(acronym.lower())

        return variations

    def detect_brands(self, text: str, target_brands: List[str]) ->
Dict[str, BrandMention]:
        """Detect brand mentions in text with context and confidence"""
        doc = self.nlp(text)
        brand_mentions = defaultdict(lambda: {"count": 0, "contexts": [],
"positions": []})

        # Create normalized brand lookup
        brand_variations = {}
        for brand in target_brands:
            for variation in self.normalize_brand_name(brand):
                brand_variations[variation] = brand

        # Check named entities first
        for ent in doc.ents:
            if ent.label_ in ["ORG", "PERSON", "PRODUCT"]:
                normalized = ent.text.lower()
                if normalized in brand_variations:
                    original_brand = brand_variations[normalized]
                    context = self._extract_context(text, ent.start_char,
ent.end_char)
                    brand_mentions[original_brand]["count"] += 1

brand_mentions[original_brand]["contexts"].append(context)

brand_mentions[original_brand]["positions"].append((ent.start_char,
ent.end_char))

        # Fallback to regex matching for missed mentions
        for brand in target_brands:
            for variation in self.normalize_brand_name(brand):
                pattern = r'\b' + re.escape(variation) + r'\b'
                matches = re.finditer(pattern, text, re.IGNORECASE)

                for match in matches:
                    # Avoid double-counting
                    start, end = match.span()
                    already_found = any(
                        abs(pos[0] - start) < 10 for pos in
brand_mentions[brand]["positions"]
                    )
                    if not already_found:
                        context = self._extract_context(text, start, end)
                        brand_mentions[brand]["count"] += 1
                        brand_mentions[brand]["contexts"].append(context)
                        brand_mentions[brand]["positions"].append((start,
end))

        # Convert to BrandMention objects with sentiment
        result = {}
        for brand, data in brand_mentions.items():
            sentiment_score = self._calculate_sentiment(data["contexts"])
            confidence = self._calculate_confidence(brand,
data["contexts"])

            result[brand] = BrandMention(
                brand=brand,
                mentions=data["count"],
                contexts=data["contexts"],
                sentiment_score=sentiment_score,
                confidence=confidence
            )

        return result

    def _extract_context(self, text: str, start: int, end: int, window: int
= 100) -> str:
        """Extract context around a brand mention"""
        context_start = max(0, start - window)
        context_end = min(len(text), end + window)
        return text[context_start:context_end].strip()

    def _calculate_sentiment(self, contexts: List[str]) -> float:
        """Simple sentiment analysis of brand contexts"""
        # This is a simplified version - in production, use a proper
sentiment model
        positive_words = ["best", "excellent", "great", "good",
"recommend", "top", "leading"]
        negative_words = ["worst", "bad", "poor", "terrible", "avoid",
"problems", "issues"]

        total_score = 0
        for context in contexts:
            context_lower = context.lower()
            positive_count = sum(1 for word in positive_words if word in
context_lower)
            negative_count = sum(1 for word in negative_words if word in
context_lower)

            if positive_count > 0 or negative_count > 0:
                score = (positive_count - negative_count) / (positive_count
+ negative_count)
                total_score += score

        return total_score / len(contexts) if contexts else 0.0

    def _calculate_confidence(self, brand: str, contexts: List[str]) ->
float:
        """Calculate confidence score for brand detection"""
        # Higher confidence for:
        # - Exact case matches
        # - Mentions in business contexts
        # - Multiple mentions
        confidence = 0.5  # Base confidence

        # Boost for multiple mentions
        if len(contexts) > 1:
            confidence += 0.2

        # Boost for business context keywords
        business_keywords = ["company", "software", "product", "service",
"business"]
        for context in contexts:
            if any(keyword in context.lower() for keyword in
business_keywords):
                confidence += 0.1
                break

        return min(1.0, confidence)
```

Phase 4: Question Engine (Week 7-8)

4.1 Dynamic Question Generation

```
# app/services/question_engine.py
from typing import List, Dict, Any
import json
from dataclasses import dataclass
from enum import Enum

class QuestionCategory(Enum):
    COMPARISON = "comparison"
    RECOMMENDATION = "recommendation"
    FEATURES = "features"
    PRICING = "pricing"
    REVIEWS = "reviews"
    ALTERNATIVES = "alternatives"

@dataclass
class QuestionTemplate:
    category: QuestionCategory
    template: str
    variations: List[str]
    industry_specific: bool = False

class QuestionEngine:
    def __init__(self):
        self.base_templates = [
            QuestionTemplate(
                category=QuestionCategory.COMPARISON,
                template="What is the best {industry} software?",
                variations=[
                    "Which {industry} software is the best?",
                    "What's the top {industry} tool?",
                    "Best {industry} software in 2024?",
                    "Leading {industry} solutions?",
                    "Top-rated {industry} platforms?"
                ]
            ),
            QuestionTemplate(
                category=QuestionCategory.RECOMMENDATION,
                template="What {industry} software do you recommend?",
                variations=[
                    "Can you recommend a good {industry} tool?",
                    "What {industry} software should I use?",
                    "Which {industry} platform do you suggest?",
                    "Recommend {industry} software for small business?"
                ]
            ),
            QuestionTemplate(
                category=QuestionCategory.ALTERNATIVES,
                template="What are alternatives to {competitor}?",
                variations=[
                    "What are {competitor} competitors?",
                    "Software similar to {competitor}?",
                    "{competitor} alternatives?",
                    "Competitors of {competitor}?",
                    "Software like {competitor}?"
                ]
            ),
            QuestionTemplate(
                category=QuestionCategory.FEATURES,
                template="What features does {brand} have?",
                variations=[
                    "What can {brand} do?",
                    "{brand} capabilities?",
                    "Features of {brand}?",
                    "What does {brand} offer?",
                    "{brand} functionality?"
                ]
            ),
            QuestionTemplate(
                category=QuestionCategory.PRICING,
                template="How much does {brand} cost?",
                variations=[
                    "What is {brand} pricing?",
                    "{brand} price?",
                    "Cost of {brand}?",
                    "{brand} subscription cost?",
                    "How expensive is {brand}?"
                ]
            )
        ]

        # Industry-specific question patterns
        self.industry_patterns = {
            "CRM": [
                "What CRM integrates with Salesforce?",
                "Best CRM for lead management?",
                "Which CRM has the best mobile app?"
            ],
            "Marketing Automation": [
                "What marketing automation tool has the best email 
features?",
                "Which platform is best for drip campaigns?",
                "Best marketing automation for ecommerce?"
            ],
            "Project Management": [
                "What project management tool is best for teams?",
                "Which PM software has Gantt charts?",
                "Best project management for remote teams?"
            ]
        }

    def generate_questions(self,
                          client_brand: str,
                          competitors: List[str],
                          industry: str,
                          categories: List[QuestionCategory] = None) ->
List[Dict[str, Any]]:
        """Generate comprehensive question set for audit"""

        if categories is None:
            categories = list(QuestionCategory)

        questions = []

        # Generate from base templates
        for template in self.base_templates:
            if template.category not in categories:
                continue

            # Generate variations for each template
            for variation in template.variations:
                question_data = {
                    "category": template.category.value,
                    "template": template.template,
                    "variation": variation,
                    "industry": industry,
                    "client_brand": client_brand,
                    "competitors": competitors
                }

                # Generate actual questions
                if "{industry}" in variation:
                    questions.append({
                        **question_data,
                        "question": variation.format(industry=industry),
                        "type": "industry_general"
                    })

                if "{brand}" in variation:
                    # Generate for client brand
                    questions.append({
                        **question_data,
                        "question": variation.format(brand=client_brand),
                        "type": "brand_specific",
                        "target_brand": client_brand
                    })

                    # Generate for each competitor
                    for competitor in competitors:
                        questions.append({
                            **question_data,
                            "question": variation.format(brand=competitor),
                            "type": "competitor_specific",
                            "target_brand": competitor
                        })

                if "{competitor}" in variation:
                    for competitor in competitors:
                        questions.append({
                            **question_data,
                            "question":
variation.format(competitor=competitor),
                            "type": "alternative_seeking",
                            "target_brand": competitor
                        })
        # Add industry-specific questions
        if industry in self.industry_patterns:
            for question in self.industry_patterns[industry]:
                questions.append({
                    "category": "industry_specific",
                    "question": question,
                    "type": "industry_specific",
                    "industry": industry,
                    "client_brand": client_brand,
                    "competitors": competitors
                })

        return questions

    def prioritize_questions(self, questions: List[Dict[str, Any]],
                           max_questions: int = 100) -> List[Dict[str,
Any]]:
        """Prioritize questions based on strategic value"""

        # Priority scoring
        priority_weights = {
            "comparison": 10,      # High value - direct competitive
intelligence
            "recommendation": 9,   # High value - recommendation scenarios
            "alternatives": 8,     # High value - competitor displacement
            "features": 6,         # Medium value - feature positioning
            "pricing": 5,          # Medium value - pricing intelligence
            "industry_specific": 7 # Medium-high value - targeted insights
        }

        # Score each question
        for question in questions:
            base_score = priority_weights.get(question["category"], 5)

            # Boost score for certain question types
            if question["type"] == "industry_general":
                question["priority_score"] = base_score + 2
            elif question["type"] == "alternative_seeking":
                question["priority_score"] = base_score + 1
            else:
                question["priority_score"] = base_score

        # Sort by priority and return top questions
        sorted_questions = sorted(questions, key=lambda x:
x["priority_score"], reverse=True)
        return sorted_questions[:max_questions]
```

Phase 5: Audit Processing Engine (Week 9-10)

5.1 Main Audit Orchestrator

```
# app/services/audit_processor.py
from typing import List, Dict, Any, Optional
import asyncio
from sqlalchemy.orm import Session
from app.models.audit import AuditRun, AuditConfig
from app.services.ai_platforms.base import BasePlatform
from app.services.question_engine import QuestionEngine
from app.services.brand_detector import BrandDetector
from app.utils.logger import logger
import uuid
```

```
from datetime import datetime

class AuditProcessor:
    def __init__(self, db: Session):
        self.db = db
        self.question_engine = QuestionEngine()
        self.brand_detector = BrandDetector()
        self.platforms = {}  # Will be populated with platform instances

    def register_platform(self, name: str, platform: BasePlatform):
        """Register an AI platform for querying"""
        self.platforms[name] = platform

    async def run_audit(self, audit_config_id: uuid.UUID) -> uuid.UUID:
        """Execute complete audit process"""

        # Create audit run record
        audit_run = AuditRun(
            id=uuid.uuid4(),
            audit_config_id=audit_config_id,
            status="running",
            started_at=datetime.utcnow()
        )
        self.db.add(audit_run)
        self.db.commit()

        try:
            # Load configuration
            config = self.db.query(AuditConfig).filter(
                AuditConfig.id == audit_config_id
            ).first()

            if not config:
                raise ValueError(f"Audit config {audit_config_id} not
found")

            logger.info(f"Starting audit run {audit_run.id} for client
{config.client.name}")

            # Generate questions
            questions = self.question_engine.generate_questions(
                client_brand=config.client.name,
                competitors=config.client.competitors,
                industry=config.client.industry,
                categories=[cat for cat in config.question_categories]
            )

            # Prioritize and limit questions
            priority_questions = self.question_engine.prioritize_questions(
                questions, max_questions=200
            )

            audit_run.total_questions = len(priority_questions) *
len(config.platforms)
            self.db.commit()

            # Process questions across all platforms
            results = await self._process_questions_batch(
                audit_run.id, priority_questions, config
            )

            # Update audit run status
            audit_run.status = "completed"
            audit_run.completed_at = datetime.utcnow()
            audit_run.processed_questions = len(results)

            self.db.commit()

            logger.info(f"Completed audit run {audit_run.id}")
            return audit_run.id

        except Exception as e:
            logger.error(f"Audit run {audit_run.id} failed: {str(e)}")
            audit_run.status = "failed"
            audit_run.error_log = str(e)
            audit_run.completed_at = datetime.utcnow()
            self.db.commit()
            raise

    async def _process_questions_batch(self,
                                       audit_run_id: uuid.UUID,
                                       questions: List[Dict],
                                       config: AuditConfig) -> List[Dict]:
        """Process questions in batches across platforms"""

        all_brands = [config.client.name] + config.client.competitors
        batch_size = 10  # Process 10 questions at a time
        results = []

        for i in range(0, len(questions), batch_size):
            batch = questions[i:i + batch_size]

            # Create tasks for all platform-question combinations
            tasks = []
            for question_data in batch:
                for platform_name in config.platforms:
                    if platform_name in self.platforms:
                        task = self._process_single_question(
                            audit_run_id=audit_run_id,
                            question_data=question_data,
                            platform_name=platform_name,
                            target_brands=all_brands
                        )
                        tasks.append(task)

            # Execute batch
            batch_results = await asyncio.gather(*tasks,
return_exceptions=True)

            # Filter successful results
            successful_results = [
                result for result in batch_results
                if not isinstance(result, Exception)
            ]
            results.extend(successful_results)

            # Update progress
            audit_run = self.db.query(AuditRun).filter(
                AuditRun.id == audit_run_id
            ).first()
            audit_run.processed_questions = len(results)
            self.db.commit()

            # Small delay between batches to be respectful to APIs
            await asyncio.sleep(2)
        return results

    async def _process_single_question(self,
                                 audit_run_id: uuid.UUID,
                                 question_data: Dict,
                                 platform_name: str,
                                 target_brands: List[str]) -> Dict:
        """Process a single question on a single platform"""

        platform = self.platforms[platform_name]
        question = question_data["question"]

        try:
            # Query the platform
            response = await platform.safe_query(question)

            if not response["success"]:
                logger.warning(f"Platform {platform_name} failed for
question: {question}")
                return None

            # Extract text response
            text_response =
platform.extract_text_response(response["response"])

            # Detect brand mentions
            brand_mentions =
self.brand_detector.detect_brands(text_response, target_brands)

            # Store response in database
            from app.models.response import Response
            db_response = Response(
                id=uuid.uuid4(),
                audit_run_id=audit_run_id,
                question=question,
                question_category=question_data.get("category", "unknown"),
                platform=platform_name,
                raw_response=text_response,
                brand_mentions={
                    brand: {
                        "mentions": mention.mentions,
                        "sentiment_score": mention.sentiment_score,
                        "confidence": mention.confidence,
                        "contexts": mention.contexts[:3]  # Store top 3
contexts
                    }
                    for brand, mention in brand_mentions.items()
                },
                response_metadata=response["response"],
                created_at=datetime.utcnow()
            )

            self.db.add(db_response)
            self.db.commit()

            return {
                "question": question,
                "platform": platform_name,
                "brand_mentions": brand_mentions,
                "success": True
            }

        except Exception as e:
            logger.error(f"Error processing question '{question}' on
{platform_name}: {str(e)}")
            return None

class AuditScheduler:
    """Handles scheduled audit execution"""

    def __init__(self):
        self.running_audits = set()

    async def schedule_audit(self, audit_config_id: uuid.UUID, frequency:
str):
        """Schedule recurring audits"""
        # This would integrate with Celery for production scheduling
        pass
```

Phase 6: Report Generation (Week 11-12)

6.1 PDF Report Generator

```
# app/services/report_generator.py
from reportlab.lib.pagesizes import letter, A4
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer, Table,
TableStyle, Image, PageBreak
from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
from reportlab.lib.units import inch
from reportlab.lib import colors
from reportlab.graphics.shapes import Drawing
from reportlab.graphics.charts.barcharts import VerticalBarChart
from reportlab.graphics.charts.piecharts import Pie
import matplotlib.pyplot as plt
import pandas as pd
from typing import Dict, List, Any
import io
import base64
from datetime import datetime
import uuid

class ReportGenerator:
    def __init__(self):
        self.styles = getSampleStyleSheet()
        self.title_style = ParagraphStyle(
            'CustomTitle',
            parent=self.styles['Heading1'],
            fontSize=24,
            spaceAfter=30,
            textColor=colors.HexColor('#2C3E50'),
            alignment=1  # Center
        )

        self.heading_style = ParagraphStyle(
            'CustomHeading',
            parent=self.styles['Heading2'],
            fontSize=16,
            spaceBefore=20,
            spaceAfter=12,
            textColor=colors.HexColor('#34495E')
        )

    def generate_audit_report(self, audit_run_id: uuid.UUID, db_session) ->
str:
        """Generate comprehensive audit report"""

        # Load audit data
        audit_data = self._load_audit_data(audit_run_id, db_session)

        # Create PDF
        filename =
f"AEO_Audit_Report_{audit_run_id}_{datetime.now().strftime('%Y%m%d')}.pdf"
        filepath = f"reports/{filename}"

        doc = SimpleDocTemplate(filepath, pagesize=A4)
        story = []

        # Title Page
        story.extend(self._create_title_page(audit_data))
        story.append(PageBreak())

        # Executive Summary
        story.extend(self._create_executive_summary(audit_data))
        story.append(PageBreak())

        # Competitive Analysis
        story.extend(self._create_competitive_analysis(audit_data))
        story.append(PageBreak())

        # Platform Performance
        story.extend(self._create_platform_analysis(audit_data))
        story.append(PageBreak())

        # Content Gap Analysis
        story.extend(self._create_content_gaps(audit_data))
        story.append(PageBreak())

        # Recommendations
        story.extend(self._create_recommendations(audit_data))

        # Build PDF
        doc.build(story)

        return filepath

    def _load_audit_data(self, audit_run_id: uuid.UUID, db_session) ->
Dict[str, Any]:
        """Load and process audit data for reporting"""

        from app.models.audit import AuditRun
        from app.models.response import Response

        audit_run = db_session.query(AuditRun).filter(
            AuditRun.id == audit_run_id
        ).first()

        responses = db_session.query(Response).filter(
            Response.audit_run_id == audit_run_id
        ).all()

        # Process data for analysis
        client_name = audit_run.audit_config.client.name
        competitors = audit_run.audit_config.client.competitors
        all_brands = [client_name] + competitors

        # Calculate metrics
        platform_stats = {}
        brand_performance = {}
        question_analysis = {}

        for response in responses:
            platform = response.platform
            question_category = response.question_category

            # Platform statistics
            if platform not in platform_stats:
                platform_stats[platform] = {
                    "total_questions": 0,
                    "brand_mentions": {brand: 0 for brand in all_brands},
                    "avg_sentiment": {brand: [] for brand in all_brands}
                }

            platform_stats[platform]["total_questions"] += 1

            # Brand performance analysis
            for brand, mention_data in response.brand_mentions.items():
                if brand in all_brands:
                    platform_stats[platform]["brand_mentions"][brand] +=
mention_data["mentions"]

platform_stats[platform]["avg_sentiment"][brand].append(mention_data["senti
ment_score"])

                    # Overall brand performance
                    if brand not in brand_performance:
                        brand_performance[brand] = {
                            "total_mentions": 0,
                            "platforms": set(),
                            "sentiment_scores": [],
                            "question_categories": {}
                        }

                    brand_performance[brand]["total_mentions"] +=
mention_data["mentions"]
                    brand_performance[brand]["platforms"].add(platform)

brand_performance[brand]["sentiment_scores"].append(mention_data["sentiment
_score"])

                    if question_category not in
brand_performance[brand]["question_categories"]:

brand_performance[brand]["question_categories"][question_category] = 0

brand_performance[brand]["question_categories"][question_category] +=
mention_data["mentions"]

        # Calculate averages
        for platform in platform_stats:
            for brand in platform_stats[platform]["avg_sentiment"]:
                scores = platform_stats[platform]["avg_sentiment"][brand]
                platform_stats[platform]["avg_sentiment"][brand] =
sum(scores) / len(scores) if scores else 0

        return {
            "audit_run": audit_run,
            "client_name": client_name,
            "competitors": competitors,
            "platform_stats": platform_stats,
            "brand_performance": brand_performance,
            "total_responses": len(responses),
            "date_range": {
                "start": audit_run.started_at,
                "end": audit_run.completed_at
            }
        }

    def _create_title_page(self, data: Dict[str, Any]) -> List[Any]:
        """Create report title page"""
        story = []

        # Title
        title = f"AEO Competitive Intelligence
Report<br/>{data['client_name']}"
        story.append(Paragraph(title, self.title_style))
        story.append(Spacer(1, 0.5*inch))

        # Report metadata
        metadata = [
            ["Report Date:", data['date_range']['end'].strftime('%B %d,
%Y')],
            ["Audit Period:", f"{data['date_range']['start'].strftime('%B
%d')} - {data['date_range']['end'].strftime('%B %d, %Y')}"],
            ["Total Questions Analyzed:", str(data['total_responses'])],
            ["Platforms Monitored:", ",
".join(data['platform_stats'].keys())],
            ["Competitors Analyzed:", ", ".join(data['competitors'])]
        ]

        table = Table(metadata, colWidths=[2*inch, 3*inch])
        table.setStyle(TableStyle([
            ('ALIGN', (0, 0), (-1, -1), 'LEFT'),
            ('FONTNAME', (0, 0), (0, -1), 'Helvetica-Bold'),
            ('FONTSIZE', (0, 0), (-1, -1), 12),
            ('BOTTOMPADDING', (0, 0), (-1, -1), 12),
        ]))

        story.append(table)

        return story

    def _create_executive_summary(self, data: Dict[str, Any]) -> List[Any]:
        """Create executive summary section"""
        story = []

        story.append(Paragraph("Executive Summary", self.title_style))

        # Calculate key metrics
        client_mentions =
data['brand_performance'].get(data['client_name'],
{}).get('total_mentions', 0)
        total_competitor_mentions = sum(
            data['brand_performance'].get(comp, {}).get('total_mentions',
0)
            for comp in data['competitors']
        )

        market_share = (client_mentions / (client_mentions +
total_competitor_mentions) * 100) if (client_mentions +
total_competitor_mentions) > 0 else 0

        # Key findings
        findings = [
            f"<b>AI Visibility Market Share:</b> {data['client_name']}
captures {market_share:.1f}% of brand mentions across AI platforms",
            f"<b>Total Brand Mentions:</b> {client_mentions} mentions for
{data['client_name']} vs {total_competitor_mentions} for all competitors
combined",
            f"<b>Platform Performance:</b> Strongest presence on
{self._get_best_platform(data)} with
{self._get_platform_mention_count(data)} mentions",
            f"<b>Competitive Position:</b>
{self._get_competitive_ranking(data)} out of {len(data['competitors']) + 1}
brands analyzed"
        ]

        for finding in findings:
            story.append(Paragraph(finding, self.styles['Normal']))
            story.append(Spacer(1, 12))

        return story

    def _create_competitive_analysis(self, data: Dict[str, Any]) ->
List[Any]:
        """Create competitive analysis section"""
        story = []

        story.append(Paragraph("Competitive Analysis", self.title_style))

        # Brand comparison table
        table_data = [["Brand", "Total Mentions", "Avg Sentiment",
"Platform Coverage"]]

        for brand in [data['client_name']] + data['competitors']:
            brand_data = data['brand_performance'].get(brand, {})
            mentions = brand_data.get('total_mentions', 0)
            sentiment_scores = brand_data.get('sentiment_scores', [])
            avg_sentiment = sum(sentiment_scores) / len(sentiment_scores)
if sentiment_scores else 0
            platform_count = len(brand_data.get('platforms', set()))

            sentiment_label = "Positive" if avg_sentiment > 0.1 else
"Negative" if avg_sentiment < -0.1 else "Neutral"

            table_data.append([
                brand,
                str(mentions),
                f"{sentiment_label} ({avg_sentiment:.2f})",
                f"{platform_count} platforms"
            ])

        table = Table(table_data, colWidths=[2*inch, 1*inch, 1.5*inch,
1.5*inch])
        table.setStyle(TableStyle([
            ('BACKGROUND', (0, 0), (-1, 0), colors.grey),
            ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),
            ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
            ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
            ('FONTSIZE', (0, 0), (-1, -1), 10),
            ('BOTTOMPADDING', (0, 0), (-1, -1), 12),
            ('BACKGROUND', (0, 1), (-1, -1), colors.beige),
            ('GRID', (0, 0), (-1, -1), 1, colors.black)
        ]))

        story.append(table)
        story.append(Spacer(1, 20))
        # Competitive insights
        story.append(Paragraph("Key Competitive Insights:",
self.heading_style))

        insights = self._generate_competitive_insights(data)
        for insight in insights:
            story.append(Paragraph(f"• {insight}", self.styles['Normal']))
            story.append(Spacer(1, 6))

        return story

    def _create_platform_analysis(self, data: Dict[str, Any]) -> List[Any]:
        """Create platform-specific analysis"""
        story = []

        story.append(Paragraph("Platform Performance Analysis",
self.title_style))

        for platform, stats in data['platform_stats'].items():
            story.append(Paragraph(f"{platform.title()} Analysis",
self.heading_style))

            # Platform summary
            client_mentions =
stats['brand_mentions'].get(data['client_name'], 0)
            total_questions = stats['total_questions']
            mention_rate = (client_mentions / total_questions * 100) if
total_questions > 0 else 0

            summary = f"""
            <b>Platform Overview:</b><br/>
            • Total questions analyzed: {total_questions}<br/>
            • {data['client_name']} mentions: {client_mentions}<br/>
            • Mention rate: {mention_rate:.1f}%<br/>
            • Average sentiment:
{stats['avg_sentiment'].get(data['client_name'], 0):.2f}
            """

            story.append(Paragraph(summary, self.styles['Normal']))
            story.append(Spacer(1, 20))

        return story

    def _create_content_gaps(self, data: Dict[str, Any]) -> List[Any]:
        """Create content gap analysis"""
        story = []

        story.append(Paragraph("Content Gap Analysis", self.title_style))

        # Analyze where competitors appear but client doesn't
        gaps = self._identify_content_gaps(data)

        story.append(Paragraph("Opportunity Areas:", self.heading_style))

        for gap in gaps[:10]:  # Top 10 opportunities
            story.append(Paragraph(f"• {gap}", self.styles['Normal']))
            story.append(Spacer(1, 6))

        return story

    def _create_recommendations(self, data: Dict[str, Any]) -> List[Any]:
        """Create actionable recommendations"""
        story = []

        story.append(Paragraph("Strategic Recommendations",
self.title_style))

        recommendations = self._generate_recommendations(data)

        for i, rec in enumerate(recommendations, 1):
            story.append(Paragraph(f"{i}. <b>{rec['title']}</b>",
self.heading_style))
            story.append(Paragraph(rec['description'],
self.styles['Normal']))
            story.append(Paragraph(f"<b>Expected Impact:</b>
{rec['impact']}", self.styles['Normal']))
            story.append(Spacer(1, 15))

        return story

    def _get_best_platform(self, data: Dict[str, Any]) -> str:
        """Identify platform with highest client mentions"""
        best_platform = ""
        max_mentions = 0

        for platform, stats in data['platform_stats'].items():
            mentions = stats['brand_mentions'].get(data['client_name'], 0)
            if mentions > max_mentions:
                max_mentions = mentions
                best_platform = platform

        return best_platform or "Unknown"

    def _get_platform_mention_count(self, data: Dict[str, Any]) -> int:
        """Get mention count for best platform"""
        best_platform = self._get_best_platform(data)
        return data['platform_stats'].get(best_platform,
{}).get('brand_mentions', {}).get(data['client_name'], 0)

    def _get_competitive_ranking(self, data: Dict[str, Any]) -> int:
        """Get client's ranking among all brands"""
        brand_mentions = [
            (brand, data['brand_performance'].get(brand,
{}).get('total_mentions', 0))
            for brand in [data['client_name']] + data['competitors']
        ]

        ranked_brands = sorted(brand_mentions, key=lambda x: x[1],
reverse=True)

        for i, (brand, mentions) in enumerate(ranked_brands, 1):
            if brand == data['client_name']:
                return i

        return len(ranked_brands)

    def _generate_competitive_insights(self, data: Dict[str, Any]) ->
List[str]:
        """Generate competitive insights"""
        insights = []

        # Market leader analysis
        brand_mentions = [
            (brand, data['brand_performance'].get(brand,
{}).get('total_mentions', 0))
```

```
            for brand in [data['client_name']] + data['competitors']
        ]

        leader = max(brand_mentions, key=lambda x: x[1])
        if leader[0] != data['client_name']:
            insights.append(f"{leader[0]} leads in AI visibility with
{leader[1]} total mentions")

        # Sentiment analysis
        client_sentiment =
data['brand_performance'].get(data['client_name'],
{}).get('sentiment_scores', [])
        if client_sentiment:
            avg_sentiment = sum(client_sentiment) / len(client_sentiment)
            if avg_sentiment > 0.1:
                insights.append(f"{data['client_name']} maintains positive
sentiment across AI platforms")
            elif avg_sentiment < -0.1:
                insights.append(f"{data['client_name']} shows negative
sentiment that needs attention")

        return insights

    def _identify_content_gaps(self, data: Dict[str, Any]) -> List[str]:
        """Identify content gap opportunities"""
        gaps = []

        # This would analyze where competitors get mentioned but client
doesn't
        # For now, return placeholder gaps
        gaps = [
            "Pricing comparison questions favor competitors",
            "Feature-specific queries show low client visibility",
            "Alternative-seeking questions miss client mentions",
            "Industry-specific use cases underrepresented"
        ]

        return gaps

    def _generate_recommendations(self, data: Dict[str, Any]) ->
List[Dict[str, str]]:
        """Generate strategic recommendations"""
        recommendations = [
            {
                "title": "Improve Content for AI Optimization",
                "description": "Create FAQ pages and structured content
that directly answers common industry questions to improve AI citation
rates.",
                "impact": "15-25% increase in mention rate"
            },
            {
                "title": "Competitive Content Strategy",
                "description": "Develop content that positions your brand
as an alternative to top-mentioned competitors in key question
categories.",
                "impact": "10-20% improvement in competitive scenarios"
            },
            {
                "title": "Platform-Specific Optimization",
                "description": f"Focus optimization efforts on
{self._get_best_platform(data)} where you show strongest performance.",
                "impact": "Enhanced visibility on primary platform"
            }
        ]

        return recommendations
```

## Phase 7: API & Frontend (Week 13-14)

### 7.1 FastAPI Endpoints
```python
# app/api/v1/audits.py
from fastapi import APIRouter, Depends, HTTPException, BackgroundTasks
from sqlalchemy.orm import Session
from typing import List, Optional
import uuid
from app.config.database import get_db
from app.models.audit import AuditConfig, AuditRun
from app.services.audit_processor import AuditProcessor
from app.tasks.audit_tasks import run_audit_task
from pydantic import BaseModel
from datetime import datetime

router = APIRouter(prefix="/audits", tags=["audits"])

class AuditConfigCreate(BaseModel):
    client_id: uuid.UUID
    name: str
    question_categories: List[str]
    platforms: List[str]
    frequency: str = "monthly"

class AuditConfigResponse(BaseModel):
    id: uuid.UUID
    name: str
    question_categories: List[str]
    platforms: List[str]
    frequency: str
    is_active: bool
    created_at: datetime

@router.post("/configs", response_model=AuditConfigResponse)
async def create_audit_config(
    config_data: AuditConfigCreate,
    db: Session = Depends(get_db)
):
    """Create new audit configuration"""

    audit_config = AuditConfig(
        id=uuid.uuid4(),
        client_id=config_data.client_id,
        name=config_data.name,
        question_categories=config_data.question_categories,
        platforms=config_data.platforms,
        frequency=config_data.frequency
    )

    db.add(audit_config)
    db.commit()
    db.refresh(audit_config)

    return audit_config

@router.post("/configs/{config_id}/run")
async def trigger_audit_run(
    config_id: uuid.UUID,
    background_tasks: BackgroundTasks,
    db: Session = Depends(get_db)
):
    """Trigger immediate audit run"""

    config = db.query(AuditConfig).filter(AuditConfig.id ==
config_id).first()
    if not config:
        raise HTTPException(status_code=404, detail="Audit config not
found")

    # Queue audit task
    background_tasks.add_task(run_audit_task, str(config_id))

    return {"message": "Audit queued for execution", "config_id":
config_id}

@router.get("/runs/{run_id}/status")
async def get_audit_status(run_id: uuid.UUID, db: Session =
Depends(get_db)):
    """Get audit run status"""

    audit_run = db.query(AuditRun).filter(AuditRun.id == run_id).first()
    if not audit_run:
        raise HTTPException(status_code=404, detail="Audit run not found")

    return {
        "id": audit_run.id,
        "status": audit_run.status,
        "progress":
f"{audit_run.processed_questions}/{audit_run.total_questions}",
        "started_at": audit_run.started_at,
        "completed_at": audit_run.completed_at,
        "error_log": audit_run.error_log
    }
```

7.2 Background Tasks with Celery

```
# app/tasks/audit_tasks.py
from celery import Celery
from app.config.settings import settings
from app.config.database import SessionLocal
from app.services.audit_processor import AuditProcessor
from app.services.ai_platforms.openai_client import OpenAIPlatform
from app.services.ai_platforms.anthropic_client import AnthropicPlatform
# Import other platform clients
import uuid

celery_app = Celery(
    "aeo_audit_tool",
    broker=settings.REDIS_URL,
    backend=settings.REDIS_URL
)

@celery_app.task(bind=True)
def run_audit_task(self, audit_config_id: str):
    """Background task to run audit"""

    db = SessionLocal()

    try:
        # Initialize processor
        processor = AuditProcessor(db)

        # Register platforms
        processor.register_platform("openai", OpenAIPlatform(
            settings.OPENAI_API_KEY,
            settings.RATE_LIMITS["openai"]
        ))
        processor.register_platform("anthropic", AnthropicPlatform(
            settings.ANTHROPIC_API_KEY,
            settings.RATE_LIMITS["anthropic"]
        ))
        # Register other platforms...

        # Run audit
        audit_run_id = await
processor.run_audit(uuid.UUID(audit_config_id))

        return {"status": "completed", "audit_run_id": str(audit_run_id)}

    except Exception as e:
        self.retry(countdown=60, max_retries=3)
        raise e
    finally:
        db.close()

@celery_app.task
def generate_report_task(audit_run_id: str):
    """Background task to generate report"""

    from app.services.report_generator import ReportGenerator

    db = SessionLocal()

    try:
        generator = ReportGenerator()
        report_path =
generator.generate_audit_report(uuid.UUID(audit_run_id), db)

        return {"status": "completed", "report_path": report_path}

    except Exception as e:
        raise e
    finally:
        db.close()
```

Phase 8: Deployment & Production Setup (Week 15-16)

8.1 Docker Configuration

```
# docker/Dockerfile
FROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    g++ \
    && rm -rf /var/lib/apt/lists/*

# Install Python dependencies
```

```
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Download spaCy model
RUN python -m spacy download en_core_web_sm

# Copy application code
COPY . .

# Create reports directory
RUN mkdir -p reports

EXPOSE 8000

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
# docker/docker-compose.yml
version: '3.8'

services:
  web:
    build:
      context: ../
      dockerfile: docker/Dockerfile
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=postgresql://postgres:password@db:5432/aeo_audit
      - REDIS_URL=redis://redis:6379
    depends_on:
      - db
      - redis
    volumes:
      - ../reports:/app/reports

  worker:
    build:
      context: ../
      dockerfile: docker/Dockerfile
    command: celery -A app.tasks.audit_tasks worker --loglevel=info
    environment:
      - DATABASE_URL=postgresql://postgres:password@db:5432/aeo_audit
      - REDIS_URL=redis://redis:6379
    depends_on:
      - db
      - redis
    volumes:
      - ../reports:/app/reports

  db:
    image: postgres:15
    environment:
      POSTGRES_DB: aeo_audit
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  redis:
    image: redis:alpine
    ports:
      - "6379:6379"
```

```
volumes:
  postgres_data:
```

Getting Started Instructions

1. Setup Development Environment

```
# Clone repository (or create new directory)
mkdir aeo-audit-tool && cd aeo-audit-tool

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

# Create requirements.txt with the dependencies listed above
# Install dependencies
pip install -r requirements.txt

# Download spaCy model
python -m spacy download en_core_web_sm
```

2. Environment Configuration

```
# Create .env file
cat > .env << EOF
DATABASE_URL=postgresql://postgres:password@localhost:5432/aeo_audit
REDIS_URL=redis://localhost:6379
OPENAI_API_KEY=your_openai_key_here
ANTHROPIC_API_KEY=your_anthropic_key_here
PERPLEXITY_API_KEY=your_perplexity_key_here
GOOGLE_AI_API_KEY=your_google_ai_key_here
SECRET_KEY=your_secret_key_here
DEBUG=True
LOG_LEVEL=INFO
EOF
```

3. Database Setup

```
# Start PostgreSQL and Redis (via Docker)
docker run -d --name postgres -e POSTGRES_DB=aeo_audit -e
POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password -p 5432:5432
postgres:15
docker run -d --name redis -p 6379:6379 redis:alpine

# Run database migrations
alembic upgrade head

# Create initial data
python scripts/seed_data.py
```

4. Start Development

```
# Terminal 1: Start web server
uvicorn app.main:app --reload --port 8000

# Terminal 2: Start Celery worker
celery -A app.tasks.audit_tasks worker --loglevel=info
```

```
# Terminal 3: Start Celery beat (for scheduling)
celery -A app.tasks.audit_tasks beat --loglevel=info
```

Testing Strategy

4.1 Unit Tests

```
# tests/test_brand_detector.py
import pytest
from app.services.brand_detector import BrandDetector

def test_brand_detection():
    detector = BrandDetector()
    text = "I recommend using Salesforce for CRM, it's better than 
HubSpot."
    brands = ["Salesforce", "HubSpot", "Pipedrive"]

    results = detector.detect_brands(text, brands)

    assert "Salesforce" in results
    assert "HubSpot" in results
    assert results["Salesforce"].mentions >= 1
    assert results["HubSpot"].mentions >= 1
    assert results["Salesforce"].sentiment_score > 0  # Positive mention

def test_brand_normalization():
    detector = BrandDetector()
    variations = detector.normalize_brand_name("Apple Inc")

    assert "apple inc" in variations
    assert "apple" in variations
    assert "apple corp" in variations

# tests/test_question_engine.py
import pytest
from app.services.question_engine import QuestionEngine, QuestionCategory

def test_question_generation():
    engine = QuestionEngine()
    questions = engine.generate_questions(
        client_brand="TestCRM",
        competitors=["Salesforce", "HubSpot"],
        industry="CRM",
        categories=[QuestionCategory.COMPARISON, 
QuestionCategory.RECOMMENDATION]
    )

    assert len(questions) > 0
    assert any("CRM" in q["question"] for q in questions)
    assert any("TestCRM" in q["question"] for q in questions)
    assert any("Salesforce" in q["question"] for q in questions)

def test_question_prioritization():
    engine = QuestionEngine()
    questions = [
        {"category": "comparison", "question": "What is the best CRM?", 
"type": "industry_general"},
        {"category": "pricing", "question": "How much does Salesforce 
cost?", "type": "competitor_specific"}
    ]
    prioritized = engine.prioritize_questions(questions, max_questions=10)
    assert len(prioritized) <= 10
    assert all("priority_score" in q for q in prioritized)
```

4.2 Integration Tests

```
# tests/test_audit_integration.py
import pytest
import asyncio
from unittest.mock import Mock, AsyncMock
from app.services.audit_processor import AuditProcessor
from app.services.ai_platforms.base import BasePlatform

class MockPlatform(BasePlatform):
    def __init__(self):
        super().__init__("mock_key", 100)

    async def query(self, question: str, **kwargs):
        return {
            "choices": [{
                "message": {
                    "content": f"Mock response for: {question}. Salesforce
is great, HubSpot is also good."
                }
            }]
        }

    def extract_text_response(self, raw_response):
        return raw_response["choices"][0]["message"]["content"]

@pytest.mark.asyncio
async def test_full_audit_process():
    # Mock database session
    mock_db = Mock()

    # Create processor with mock platform
    processor = AuditProcessor(mock_db)
    processor.register_platform("mock", MockPlatform())

    # This would require more complex mocking for full integration test
    # Focus on testing individual components thoroughly
```

4.3 Performance Tests

```
# tests/test_performance.py
import pytest
import time
import asyncio
from app.services.brand_detector import BrandDetector

def test_brand_detection_performance():
    detector = BrandDetector()

    # Large text with multiple brands
    large_text = "Salesforce is better than HubSpot. " * 1000
    brands = ["Salesforce", "HubSpot", "Pipedrive", "Zoho", "Microsoft"]

    start_time = time.time()
    results = detector.detect_brands(large_text, brands)
    end_time = time.time()
    # Should process within reasonable time
    assert end_time - start_time < 5.0  # 5 seconds max
    assert len(results) > 0

@pytest.mark.asyncio
async def test_rate_limiting():
    from app.services.ai_platforms.base import AIRateLimiter

    limiter = AIRateLimiter(requests_per_minute=60)  # 1 per second

    start_time = time.time()

    # Make 3 requests quickly
    await limiter.acquire()
    await limiter.acquire()
    await limiter.acquire()

    end_time = time.time()

    # Should take at least 2 seconds (rate limited)
    assert end_time - start_time >= 2.0
```

Production Deployment Checklist

Security

- [ ] API keys stored in secure environment variables
- [ ] Database credentials secured
- [ ] CORS configured properly
- [ ] Input validation on all endpoints
- [ ] Rate limiting implemented
- [ ] SSL/TLS certificates configured

Monitoring

- [ ] Application logging configured (structured logging)
- [ ] Error tracking (Sentry integration)
- [ ] Performance monitoring
- [ ] Database query monitoring
- [ ] API response time tracking
- [ ] Celery task monitoring

Scalability

- [ ] Database connection pooling
- [ ] Redis connection pooling
- [ ] Horizontal scaling capability
- [ ] Load balancer configuration
- [ ] CDN for report delivery
- [ ] Database read replicas (if needed)

Backup & Recovery

- [ ] Database backups scheduled
- [ ] Report file backups
- [ ] Configuration backups
- [ ] Disaster recovery plan
- [ ] Data retention policies

Key Success Metrics

Technical Metrics

- **API Response Time**: < 200ms for most endpoints
- **Audit Processing Time**: < 2 hours for 200 questions across 4 platforms
- **System Uptime**: > 99.5%
- **Error Rate**: < 1% of API calls
- **Report Generation**: < 5 minutes per report

Business Metrics

- **Agency Adoption**: 10+ agencies in first 6 months
- **Customer Retention**: > 80% monthly retention
- **Report Quality**: 4.5+ star rating from agencies
- **API Usage**: 1000+ audit runs per month
- **Revenue**: $50k+ ARR by month 12

Critical Lessons from First Build

1. **Start with robust error handling** - API failures will happen frequently
2. **Implement comprehensive rate limiting** - Each platform has different limits
3. **Design for brand entity complexity** - Simple keyword matching is insufficient
4. **Plan for data volume growth** - Database design matters from day one
5. **Build monitoring early** - You need visibility into what's working and what's not
6. **Focus on report quality** - Agencies judge the entire product by report quality
7. **Test with real data** - Mock responses don't reveal actual AI platform variations
8. **Design for white-labeling** - Agency customization requirements are complex
9. **Implement proper task queues** - Sequential processing doesn't scale
10. **Plan for API changes** - AI platforms frequently update their APIs

This build plan incorporates all the painful lessons learned from the first attempt and provides a production-ready architecture that can scale to hundreds of agencies while maintaining reliability and performance.