**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["sentiment\_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.