Unarchived: Complete Architectural Roadmap & Implementation Guide

🎯 Executive Summary

Vision: Transform product development from reactive tool usage into a proactive AI-first design partnership that handles the complete journey from concept to manufactured product.

Core Principle: The agent is not a reactive chatbot, but a stateful design partner that understands project context, generates visual assets, and iteratively refines them based on user feedback, culminating in manufacturing-ready specifications.

Ultimate Goal: Users simply describe what they want, and the AI handles everything from design to delivery - true "Pay and Wait" manufacturing.

🏗️ Part I: Foundational Architecture

Core Data Models: The Foundation of Intelligence

The entire system is built upon two critical Django models that anchor all AI intelligence and state management.

1.1 Enhanced Project Model

File: `backend/projects/models.py`

```
from django.db import models
from django.conf import settings

class ProjectStage(models.Model):
    name = models.CharField(max_length=100, unique=True)
    order = models.PositiveIntegerField(
        default=0,
        help_text="Defines the sequence of stages for display."
    )

    class Meta:
        ordering = ['order']

    def __str__(self):
```

```
        return self.name

class Project(models.Model):
    """
    The highest-level container representing a single product development endeavor.
    Every conversation, DPG, and design decision happens within this context.
    """
    class ProjectStatus(models.TextChoices):
        ACTIVE = "ACTIVE", "Active"
        ON_HOLD = "ON_HOLD", "On Hold"
        COMPLETED = "COMPLETED", "Completed"
        ARCHIVED = "ARCHIVED", "Archived"

    # Core project information
    name = models.CharField(max_length=255)
    description = models.TextField(blank=True)
    parent = models.ForeignKey(
        'self',
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name='sub_projects'
    )
    status = models.CharField(
        max_length=20,
        choices=ProjectStatus.choices,
        default=ProjectStatus.ACTIVE
    )
    stage = models.ForeignKey(
        ProjectStage,
        on_delete=models.SET_NULL,
        null=True,
        blank=True,
        related_name="projects"
    )
    category = models.CharField(
        max_length=100,
        blank=True,
        help_text="e.g., Apparel, Electronics, Home Goods"
    )

    # Ownership and collaboration
    owner = models.ForeignKey(
        settings.AUTH_USER_MODEL,
```

```
        on_delete=models.PROTECT,
        related_name='owned_projects'
    )
    members = models.ManyToManyField(
        settings.AUTH_USER_MODEL,
        through="ProjectMember",
        related_name="projects"
    )

    # CRITICAL: The Agent's Focus Anchor
    active_dpg = models.OneToOneField(
        'dpgs.DigitalProductGenome',
        on_delete=models.SET_NULL,
        null=True,
        blank=True,
        related_name='active_in_project',
        help_text="The DPG currently being worked on by the agent in this project."
    )

    # Timestamps
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.name

class ProjectMember(models.Model):
    """Defines user roles and permissions within a project."""

    class MemberRole(models.TextChoices):
        OWNER = 'OWNER', 'Owner'
        EDITOR = 'EDITOR', 'Editor'
        VIEWER = 'VIEWER', 'Viewer'

    project = models.ForeignKey(
        Project,
        on_delete=models.CASCADE,
        related_name='memberships'
    )
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name='project_memberships'
    )
```

```
    role = models.CharField(
        max_length=20,
        choices=MemberRole.choices,
        default=MemberRole.VIEWER
    )
    joined_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        unique_together = ('user', 'project')
        ordering = ['-joined_at']
```

1.2 Enhanced Digital Product Genome (DPG)

File: `backend/dpgs/models.py`

```
from django.db import models
from django.conf import settings
from projects.models import Project

class DigitalProductGenome(models.Model):
    """
    The agent's primary output - a structured, living representation of a product
    that serves as both the agent's memory and the final manufacturing specification.
    """

    # CRITICAL: Agent State Machine
    LIFECYCLE_STAGES = [
        ('ignition', 'Ignition'),  # Files uploaded, processing
        ('context_synthesized', 'Context Synthesized'),  # Ready to propose design
        ('designing', 'Designing'),  # Iterative visual design loop
        ('specifying', 'Specifying'),  # Completing structured data
        ('reviewed', 'Reviewed'),  # Awaiting final approval
        ('approved', 'Approved'),  # Locked, ready for sourcing
    ]

    # Core relationships
    project = models.ForeignKey(
        Project,
        on_delete=models.CASCADE,
        related_name='dpgs'
    )
    owner = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
```

```
    related_name='dpgs'
)

# Product information
title = models.CharField(max_length=255)
description = models.TextField(blank=True)
version = models.CharField(max_length=20, default='1.0')
summary = models.TextField(
    blank=True,
    help_text="AI-generated summary of project context"
)

# Structured product data (flexible JSON schema)
data = models.JSONField(
    default=dict,
    help_text="Manufacturing specifications, materials, dimensions, etc."
)

# Agent state and memory
stage = models.CharField(
    max_length=50,
    choices=LIFECYCLE_STAGES,
    default='ignition'
)

# CRITICAL: Visual Asset History with Full Versioning
visual_assets = models.JSONField(
    default=dict,
    help_text="""
    Complete history of all generated visual assets:
    {
        "flat_views": {
            "front": [
                {"version": 1, "url": "s3://...", "prompt": "...", "timestamp": "..."},
                {"version": 2, "url": "s3://...", "prompt": "...", "timestamp": "..."}
            ],
            "back": [{"version": 1, "url": "...", "prompt": "..."}],
            "side": [{"version": 1, "url": "...", "prompt": "..."}]
        },
        "callouts": [
            {"name": "collar_detail", "url": "s3://...", "prompt": "...", "timestamp": "..."}
        ],
        "concept_sketches": [
            {"version": 1, "url": "s3://...", "prompt": "...", "timestamp": "..."}
```

```
        ]
    }
    """
)

# CRITICAL: Perfect Conversation Memory
conversation_history = models.JSONField(
    default=list,
    help_text="Complete log of all user-agent interactions for this specific DPG"
)

# Timestamps
created_at = models.DateTimeField(auto_now_add=True)
updated_at = models.DateTimeField(auto_now=True)

class Meta:
    ordering = ['-created_at']

def __str__(self):
    return f"{self.title} (v{self.version})"

def get_latest_visual_asset(self, asset_type: str, view: str = None):
    """Get the most recent version of a specific visual asset."""
    if view:
        assets = self.visual_assets.get(asset_type, {}).get(view, [])
    else:
        assets = self.visual_assets.get(asset_type, [])

    return assets[-1] if assets else None

def add_visual_asset(self, asset_type: str, asset_data: dict, view: str = None):
    """Add a new visual asset with proper versioning."""
    if asset_type not in self.visual_assets:
        self.visual_assets[asset_type] = {} if view else []

    if view:
        if view not in self.visual_assets[asset_type]:
            self.visual_assets[asset_type][view] = []
        self.visual_assets[asset_type][view].append(asset_data)
    else:
        self.visual_assets[asset_type].append(asset_data)

    self.save(update_fields=['visual_assets', 'updated_at'])
```

🤖 Part II: The Generative Intelligence Engine

2.1 Image Generation Service

File: `backend/generative/services.py`

```
import replicate
import boto3
import uuid
import requests
import logging
from decouple import config
from datetime import datetime

logger = logging.getLogger(__name__)

class ImageGenerationService:
    """
    Core service for all AI image generation and asset management.
    Handles external API calls and persistent storage of generated assets.
    """

    def __init__(self):
        self.replicate_client = self._init_replicate_client()
        self.s3_client = self._init_s3_client()
        self.bucket_name = config("AWS_STORAGE_BUCKET_NAME")

    def _init_replicate_client(self):
        try:
            return replicate.Client(api_token=config("REPLICATE_API_TOKEN"))
        except Exception as e:
            logger.error(f"Failed to initialize Replicate client: {e}")
            return None

    def _init_s3_client(self):
        try:
            return boto3.client(
                's3',
                aws_access_key_id=config("AWS_ACCESS_KEY_ID"),
                aws_secret_access_key=config("AWS_SECRET_ACCESS_KEY"),
                region_name=config("AWS_S3_REGION_NAME")
            )
        except Exception as e:
            logger.error(f"Failed to initialize S3 client: {e}")
```

```
            return None

    def _persist_asset_to_s3(self, temp_url: str, project_id: int, asset_type: str, version: int = 1) ->
str:
        """Download from temporary URL and upload to permanent S3 storage."""
        if not self.s3_client:
            logger.error("S3 client not available")
            return temp_url

        try:
            # Download from temporary URL
            response = requests.get(temp_url, stream=True, timeout=60)
            response.raise_for_status()

            # Generate permanent asset key
            asset_key =
f"generated_assets/project_{project_id}/{asset_type}_v{version}_{uuid.uuid4().hex}.png"

            # Upload to S3
            self.s3_client.upload_fileobj(
                response.raw,
                self.bucket_name,
                asset_key,
                ExtraArgs={'ContentType': 'image/png'}
            )

            # Return permanent URL
            permanent_url =
f"https://{self.bucket_name}.s3.{config('AWS_S3_REGION_NAME')}.amazonaws.com/{asset_ke
y}"
            logger.info(f"Persisted asset: {permanent_url}")
            return permanent_url

        except Exception as e:
            logger.error(f"Failed to persist asset: {e}")
            return temp_url

    def generate_technical_flats(self, project_id: int, prompt: str, reference_image_url: str = None)
-> list:
        """Generate professional technical flat sketches."""
        if not self.replicate_client:
            raise ConnectionError("Replicate client not initialized")
```

```
        model_endpoint =
"jagilley/controlnet-canny:aff48af9c68d167a28d20def99192a5146c103031023b0304337b019b8
84952b"

        base_prompt = f"professional technical flat sketch of a {prompt}, fashion design drawing,
clean vector line art, white background, no shadows, manufacturing spec style"

        generated_assets = []
        views = ['front', 'back', 'side']

        for view in views:
            try:
                full_prompt = f"{base_prompt}, {view} view"

                # Generate image
                output = self.replicate_client.run(
                    model_endpoint,
                    input={
                        "image": reference_image_url,
                        "prompt": full_prompt,
                        "num_inference_steps": 20,
                        "guidance_scale": 7.5
                    }
                )

                if output and len(output) > 0:
                    temp_url = output[0]
                    permanent_url = self._persist_asset_to_s3(
                        temp_url, project_id, f"flat_{view}", version=1
                    )

                    generated_assets.append({
                        "asset_type": f"flat_{view}_view",
                        "version": 1,
                        "url": permanent_url,
                        "prompt": full_prompt,
                        "timestamp": datetime.now().isoformat()
                    })
                else:
                    logger.warning(f"No output for {view} view")

            except Exception as e:
                logger.error(f"Failed to generate {view} view: {e}")
                generated_assets.append({
```

```
                "asset_type": f"flat_{view}_view",
                "version": 1,
                "url": None,
                "error": str(e),
                "timestamp": datetime.now().isoformat()
            })

        return generated_assets

    def inpaint_image(self, project_id: int, base_image_url: str, mask_prompt: str, edit_prompt:
str, version: int) -> dict:
        """Perform targeted edits on existing images using inpainting."""
        if not self.replicate_client:
            raise ConnectionError("Replicate client not initialized")

        model_endpoint =
"stability-ai/sdxl:c221b2b8ef527988fb59bf24a8b97c4561f1c671f73b786280b879ce0c961646"

        full_prompt = f"Professional technical flat sketch. In the area of '{mask_prompt}',
{edit_prompt}. Maintain same style and line weight throughout."

        try:
            output = self.replicate_client.run(
                model_endpoint,
                input={
                    "image": base_image_url,
                    "prompt": full_prompt,
                    "num_inference_steps": 25,
                    "guidance_scale": 8.0
                }
            )

            if output and len(output) > 0:
                temp_url = output[0]
                permanent_url = self._persist_asset_to_s3(
                    temp_url, project_id, "inpainted", version
                )

                return {
                    "asset_type": "inpainted_view",
                    "version": version,
                    "url": permanent_url,
                    "prompt": full_prompt,
                    "mask_prompt": mask_prompt,
```

```
                "edit_prompt": edit_prompt,
                "timestamp": datetime.now().isoformat()
            }
        else:
            logger.error("No output from inpainting model")
            return None

    except Exception as e:
        logger.error(f"Inpainting failed: {e}")
        return None

  def generate_concept_sketches(self, project_id: int, description: str, style_references: list =
None) -> list:
    """Generate initial concept sketches from description."""
    if not self.replicate_client:
        raise ConnectionError("Replicate client not initialized")

    model_endpoint =
"stability-ai/sdxl:39ed52f2a78e934b3ba6e2a89f5b1c712de7dfea535525255b1aa35c5565e08b"

    concepts = []
    variations = 3  # Generate multiple concept variations

    for i in range(variations):
        try:
            prompt = f"product concept sketch, {description}, clean line art, professional design
sketch, white background"

            output = self.replicate_client.run(
                model_endpoint,
                input={
                    "prompt": prompt,
                    "num_inference_steps": 30,
                    "guidance_scale": 7.0,
                    "width": 1024,
                    "height": 1024
                }
            )

            if output and len(output) > 0:
                temp_url = output[0]
                permanent_url = self._persist_asset_to_s3(
                    temp_url, project_id, f"concept_{i+1}", version=1
                )
```

```
            concepts.append({
                "asset_type": "concept_sketch",
                "version": 1,
                "variation": i + 1,
                "url": permanent_url,
                "prompt": prompt,
                "timestamp": datetime.now().isoformat()
            })

        except Exception as e:
            logger.error(f"Failed to generate concept {i+1}: {e}")

    return concepts
```

2.2 LangChain Tools Interface

File: `backend/generative/tools.py`

```
from langchain_core.tools import tool
from .services import ImageGenerationService
import logging

logger = logging.getLogger(__name__)

# Single service instance for efficiency
generation_service = ImageGenerationService()

@tool
def generate_technical_flats_tool(project_id: int, prompt: str, reference_image_url: str = None)
-> list:
    """
    Generate professional 2D technical flat sketches (front, back, side views)
    from a detailed product description and optional reference image.
    This is the primary tool for creating initial visual designs.
    """
    try:
        return generation_service.generate_technical_flats(
            project_id, prompt, reference_image_url
        )
    except Exception as e:
        logger.error(f"Technical flats generation failed: {e}")
        return [{"error": str(e)}]
```

```
@tool
def inpaint_image_tool(project_id: int, image_url: str, mask_prompt: str, edit_prompt: str,
version: int) -> dict:
    """
    Modify specific regions of an existing image for iterative design changes.

    Args:
        image_url: The base image to modify
        mask_prompt: What part to change (e.g., "the collar")
        edit_prompt: How to change it (e.g., "make it a pointed collar")
        version: Version number for the new asset
    """
    try:
        return generation_service.inpaint_image(
            project_id, image_url, mask_prompt, edit_prompt, version
        )
    except Exception as e:
        logger.error(f"Image inpainting failed: {e}")
        return {"error": str(e)}

@tool
def generate_concept_sketches_tool(project_id: int, description: str, style_references: list =
None) -> list:
    """
    Generate multiple concept sketch variations from a product description.
    Used for early-stage ideation and design exploration.
    """
    try:
        return generation_service.generate_concept_sketches(
            project_id, description, style_references
        )
    except Exception as e:
        logger.error(f"Concept sketch generation failed: {e}")
        return [{"error": str(e)}]
```

🧠 Part III: The Proactive Agent Intelligence

3.1 Core Agent Architecture

File: `backend/agentcore/agent.py`

```
import json
import re
from datetime import datetime
from decouple import config
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage
from projects.models import Project
from dpgs.models import DigitalProductGenome
from knowledge_base.models import KnowledgeChunk
from generative.tools import (
    generate_technical_flats_tool,
    inpaint_image_tool,
    generate_concept_sketches_tool
)
from django.db.models import Q
import logging

logger = logging.getLogger(__name__)

class ConversationalAgent:
    """
    A stateful, proactive, multi-modal agent that drives product development.

    Core Principles:
    1. Always operates within a Project context
    2. Maintains perfect memory via DPG conversation history
    3. Behavior driven by DPG lifecycle stage
    4. Proactively proposes next steps
    5. Generates and iterates on visual assets
    """

    def __init__(self, project_id: int, user_id: int):
        """Initialize agent with specific project and user context."""
        self.project_id = project_id
        self.user_id = user_id

        try:
            self.project = Project.objects.get(pk=project_id)
        except Project.DoesNotExist:
            logger.error(f"Agent initialized with non-existent project: {project_id}")
            raise ValueError("Project not found")

        self.llm = ChatOpenAI(
            model_name=config("OPENAI_MODEL", default="gpt-4-turbo"),
```

```
            temperature=0.3
        )

    def chat(self, user_message: str, files: list = None) -> dict:
        """
        Main entry point for all user interactions.
        Routes conversation based on DPG lifecycle stage.
        """
        # Get or create the active DPG for this project
        active_dpg = self._get_or_create_active_dpg(user_message)

        # Log user message to conversation history
        self._log_user_message(active_dpg, user_message)

        # Route based on current DPG stage
        if active_dpg.stage == 'ignition':
            return self._handle_ignition_stage(active_dpg, user_message, files)
        elif active_dpg.stage == 'context_synthesized':
            return self._handle_design_permission(active_dpg, user_message)
        elif active_dpg.stage == 'designing':
            return self._handle_design_iteration(active_dpg, user_message)
        elif active_dpg.stage == 'specifying':
            return self._handle_specification_stage(active_dpg, user_message)
        elif active_dpg.stage in ['reviewed', 'approved']:
            return self._handle_finalized_stage(active_dpg, user_message)

        # Fallback
        return {"response": "I'm not sure how to proceed. Could you clarify what you'd like to do
next?"}

    def _get_or_create_active_dpg(self, user_message: str) -> DigitalProductGenome:
        """Get existing active DPG or create new one."""
        if self.project.active_dpg:
            logger.info(f"Resuming DPG {self.project.active_dpg.id}")
            return self.project.active_dpg
        else:
            logger.info(f"Creating new DPG for project {self.project_id}")
            new_dpg = DigitalProductGenome.objects.create(
                project=self.project,
                owner_id=self.user_id,
                title=f"Product Design - {datetime.now().strftime('%Y-%m-%d')}",
                data={"initial_prompt": user_message},
                stage='ignition'
            )
```

```
            self.project.active_dpg = new_dpg
            self.project.save()
            return new_dpg

    def _log_user_message(self, dpg: DigitalProductGenome, message: str):
        """Log user message to DPG conversation history."""
        dpg.conversation_history.append({
            "role": "user",
            "content": message,
            "timestamp": datetime.now().isoformat()
        })
        dpg.save(update_fields=['conversation_history'])

    def _log_agent_response(self, dpg: DigitalProductGenome, response: str):
        """Log agent response to DPG conversation history."""
        dpg.conversation_history.append({
            "role": "assistant",
            "content": response,
            "timestamp": datetime.now().isoformat()
        })
        dpg.save(update_fields=['conversation_history'])

    def _handle_ignition_stage(self, dpg: DigitalProductGenome, user_message: str, files: list =
None) -> dict:
        """
        Initial stage: Process uploaded files and synthesize context.
        Transitions to context_synthesized stage.
        """
        logger.info(f"Processing ignition stage for DPG {dpg.id}")

        # Process any uploaded files (would typically be handled by Celery)
        if files:
            self._process_uploaded_files(files)

        # Synthesize project context from knowledge base
        context_summary = self._synthesize_project_context()

        # Update DPG with synthesized context
        dpg.summary = context_summary
        dpg.stage = 'context_synthesized'
        dpg.save()

        response_text = f"""I've analyzed your project context and here's what I understand you
want to create:
```

```
**{context_summary}**

Based on this understanding, I can generate initial design concepts and technical sketches to get us started. This will include:
- Multiple concept variations to explore different directions
- Professional technical flat sketches (front, back, side views)
- Detailed callout drawings of key features

**Shall I proceed with generating the initial design concepts?**"""

        self._log_agent_response(dpg, response_text)
        return {"response": response_text}

    def _handle_design_permission(self, dpg: DigitalProductGenome, user_message: str) -> dict:
        """
        Handle user response to design generation proposal.
        If approved, transition to designing stage and generate first concepts.
        """
        # Use LLM to interpret user's intent
        intent_prompt = f"""
        Analyze this user message and determine if they're giving permission to proceed with design generation.

        User message: "{user_message}"

        Respond with only 'YES' if they're agreeing/giving permission, or 'NO' if they're declining or asking for something else.
        """

        intent_response = self.llm.invoke([SystemMessage(content=intent_prompt)])
        user_approved = 'YES' in intent_response.content.upper()

        if user_approved:
            # User approved - transition to designing stage
            dpg.stage = 'designing'
            dpg.save()
            return self._generate_initial_concepts(dpg)
        else:
            response_text = "No problem! Let me know when you're ready for me to generate the design concepts, or if you have any questions about the project."
            self._log_agent_response(dpg, response_text)
            return {"response": response_text}
```

```
def _generate_initial_concepts(self, dpg: DigitalProductGenome) -> dict:
    """
    Generate the first set of visual assets for the product.
    This is the agent's first autonomous creative action.
    """
    logger.info(f"Generating initial concepts for DPG {dpg.id}")

    try:
        # Get reference image from knowledge base if available
        reference_chunk = KnowledgeChunk.objects.filter(
            metadata__project_id=self.project_id,
            metadata__file_type='image'
        ).first()

        reference_url = reference_chunk.metadata.get('s3_url') if reference_chunk else None

        # Generate concept sketches first
        concept_sketches = generate_concept_sketches_tool(
            project_id=self.project_id,
            description=dpg.summary
        )

        # Generate technical flats
        technical_flats = generate_technical_flats_tool(
            project_id=self.project_id,
            prompt=dpg.summary,
            reference_image_url=reference_url
        )

        # Store assets in DPG
        if concept_sketches:
            dpg.visual_assets['concept_sketches'] = concept_sketches

        if technical_flats:
            dpg.visual_assets['flat_views'] = {'front': [], 'back': [], 'side': []}
            for asset in technical_flats:
                if 'front' in asset.get('asset_type', ''):
                    dpg.visual_assets['flat_views']['front'].append(asset)
                elif 'back' in asset.get('asset_type', ''):
                    dpg.visual_assets['flat_views']['back'].append(asset)
                elif 'side' in asset.get('asset_type', ''):
                    dpg.visual_assets['flat_views']['side'].append(asset)

        dpg.save()
```

```
            response_text = """Perfect! I've generated your initial design concepts. Here's what I've created:

🎨 **Concept Sketches**: Multiple design directions to explore different approaches
📐 **Technical Flats**: Professional manufacturing-ready sketches (front, back, side views)

Take a look at these designs and let me know:
- Which concept direction resonates with you most?
- Any specific changes or modifications you'd like to see?
- Elements from different concepts you'd like to combine?

I can iterate on any of these designs based on your feedback!"""

            self._log_agent_response(dpg, response_text)

            all_assets = concept_sketches + technical_flats
            return {
                "response": response_text,
                "images": [asset for asset in all_assets if asset.get('url')]
            }

        except Exception as e:
            logger.error(f"Failed to generate initial concepts: {e}")
            error_response = "I encountered an issue generating the initial designs. Could you provide more details about what you'd like to create?"
            self._log_agent_response(dpg, error_response)
            return {"response": error_response}

    def _handle_design_iteration(self, dpg: DigitalProductGenome, user_message: str) -> dict:
        """
        Handle iterative design refinement based on user feedback.
        Uses LLM to understand intent and generative tools to modify designs.
        """
        logger.info(f"Handling design iteration for DPG {dpg.id}")

        # Analyze user intent
        intent_analysis = self._analyze_design_feedback(user_message)

        if intent_analysis['intent'] == 'MODIFY_DESIGN':
            return self._modify_existing_design(dpg, intent_analysis)
        elif intent_analysis['intent'] == 'APPROVE_DESIGN':
            return self._approve_design(dpg)
        elif intent_analysis['intent'] == 'GENERATE_VARIATIONS':
```

```
            return self._generate_design_variations(dpg, intent_analysis)
        elif intent_analysis['intent'] == 'ASK_QUESTION':
            return self._answer_design_question(dpg, user_message)
        else:
            # Default: ask for clarification
            response_text = "I want to make sure I understand correctly. Are you looking to:\n- Modify the current design in some way?\n- Approve the design and move to specifications?\n- Generate new variations?\n- Ask a question about the design?\n\nPlease let me know how you'd like to proceed!"
            self._log_agent_response(dpg, response_text)
            return {"response": response_text}

    def _analyze_design_feedback(self, user_message: str) -> dict:
        """Use LLM to analyze user feedback and determine intent."""
        analysis_prompt = f"""
        Analyze this user feedback about a product design and classify their intent.

        User message: "{user_message}"

        Possible intents:
        - MODIFY_DESIGN: User wants to change something specific about the current design
        - APPROVE_DESIGN: User likes the design and wants to move forward
        - GENERATE_VARIATIONS: User wants to see different variations or alternatives
        - ASK_QUESTION: User is asking a question about the design or process

        Respond with JSON:
        {{
            "intent": "INTENT_NAME",
            "confidence": 0.0-1.0,
            "details": "specific details about their request",
            "modification_type": "visual|structural|material|sizing" (if MODIFY_DESIGN),
            "target_element": "what part they want to change" (if MODIFY_DESIGN)
        }}
        """

        try:
            response = self.llm.invoke([SystemMessage(content=analysis_prompt)])
            return json.loads(response.content)
        except Exception as e:
            logger.error(f"Failed to analyze design feedback: {e}")
            return {"intent": "ASK_QUESTION", "confidence": 0.0, "details": "Could not parse intent"}

    def _modify_existing_design(self, dpg: DigitalProductGenome, intent_analysis: dict) -> dict:
        """Modify the current design based on user feedback."""
```

```
        try:
            # Get the latest front view design
            latest_front = dpg.get_latest_visual_asset('flat_views', 'front')
            if not latest_front or not latest_front.get('url'):
                return {"response": "I don't have a current design to modify. Let me generate a new
one first."}

            # Extract modification parameters
            modification_prompt = f"""
            Based on this user feedback: "{intent_analysis['details']}"

            Extract the specific modification instructions:
            - mask_prompt: What part of the design to change (be specific about the visual element)
            - edit_prompt: How to change it (the desired modification)

            Respond with JSON:
            {{
                "mask_prompt": "specific element to modify",
                "edit_prompt": "how to modify it"
            }}
            """

            mod_response = self.llm.invoke([SystemMessage(content=modification_prompt)])
            modification_data = json.loads(mod_response.content)

            # Generate new version using inpainting
            new_version = latest_front['version'] + 1
            modified_asset = inpaint_image_tool(
                project_id=self.project_id,
                image_url=latest_front['url'],
                mask_prompt=modification_data['mask_prompt'],
                edit_prompt=modification_data['edit_prompt'],
                version=new_version
            )

            if modified_asset and modified_asset.get('url'):
                # Add to DPG visual assets
                dpg.add_visual_asset('flat_views', modified_asset, 'front')

                response_text = f"Great feedback! I've updated the design with your requested
changes: {intent_analysis['details']}. How does this new version look?"
                self._log_agent_response(dpg, response_text)
                return {
                    "response": response_text,
```

```
                "images": [modified_asset]
            }
        else:
            error_response = "I had trouble making that specific change. Could you describe it
differently or be more specific about what you'd like modified?"
            self._log_agent_response(dpg, error_response)
            return {"response": error_response}

    except Exception as e:
        logger.error(f"Design modification failed: {e}")
        error_response = "I encountered an issue modifying the design. Could you rephrase
your request?"
        self._log_agent_response(dpg, error_response)
        return {"response": error_response}

def _approve_design(self, dpg: DigitalProductGenome) -> dict:
    """User approved the design - transition to specification stage."""
    dpg.stage = 'specifying'
    dpg.save()

    response_text = """Excellent! I'm glad you're happy with the design direction.

Now let's move on to finalizing the technical specifications. I'll need to gather some additional
details to create complete manufacturing documentation.

Let me analyze what specifications we still need based on the approved design..."""

    self._log_agent_response(dpg, response_text)

    # Immediately follow up with specification questions
    return self._handle_specification_stage(dpg, "Let's complete the specifications")

def _generate_design_variations(self, dpg: DigitalProductGenome, intent_analysis: dict) ->
dict:
    """Generate new design variations based on user request."""
    try:
        variation_prompt = f"{dpg.summary} - {intent_analysis['details']}"
        new_concepts = generate_concept_sketches_tool(
            project_id=self.project_id,
            description=variation_prompt
        )

        if new_concepts:
            # Add variations to existing concept sketches
```

```
                if 'concept_sketches' not in dpg.visual_assets:
                    dpg.visual_assets['concept_sketches'] = []
                dpg.visual_assets['concept_sketches'].extend(new_concepts)
                dpg.save()

                response_text = f"Here are some new design variations based on your request: {intent_analysis['details']}. Which direction interests you most?"
                self._log_agent_response(dpg, response_text)
                return {
                    "response": response_text,
                    "images": new_concepts
                }
            else:
                error_response = "I had trouble generating variations. Could you be more specific about what kind of alternatives you're looking for?"
                self._log_agent_response(dpg, error_response)
                return {"response": error_response}

        except Exception as e:
            logger.error(f"Variation generation failed: {e}")
            error_response = "I encountered an issue generating variations. Could you clarify what you're looking for?"
            self._log_agent_response(dpg, error_response)
            return {"response": error_response}

    def _answer_design_question(self, dpg: DigitalProductGenome, user_message: str) -> dict:
        """Answer user questions about the design using RAG."""
        # Get relevant context from knowledge base
        context = self._get_relevant_context(user_message)

        qa_prompt = f"""
        Answer this user question about their product design project:

        Question: "{user_message}"

        Context from project files:
        {context}

        Current design stage: {dpg.stage}
        Product summary: {dpg.summary}

        Provide a helpful, specific answer based on the available context.
        """
```

```
        try:
            response = self.llm.invoke([SystemMessage(content=qa_prompt)])
            answer = response.content
            self._log_agent_response(dpg, answer)
            return {"response": answer}
        except Exception as e:
            logger.error(f"Question answering failed: {e}")
            fallback_response = "I'm having trouble accessing the project context right now. Could
you rephrase your question?"
            self._log_agent_response(dpg, fallback_response)
            return {"response": fallback_response}

    def _handle_specification_stage(self, dpg: DigitalProductGenome, user_message: str) -> dict:
        """
        Handle the specification completion stage.
        Intelligently asks questions to fill in missing technical details.
        """
        logger.info(f"Handling specification stage for DPG {dpg.id}")

        # Analyze current DPG data to identify missing specifications
        missing_specs = self._identify_missing_specifications(dpg)

        if missing_specs:
            # Ask intelligent questions about the most critical missing spec
            critical_spec = missing_specs[0]  # Prioritized list
            question = self._generate_specification_question(critical_spec, dpg)

            response_text = f"""Perfect! Now let's finalize the technical specifications for
manufacturing.

{question}

This will help ensure we have all the details needed for accurate production quotes."""

            self._log_agent_response(dpg, response_text)
            return {"response": response_text}
        else:
            # All specifications complete - transition to review
            dpg.stage = 'reviewed'
            dpg.save()

            completion_response = """Excellent! We now have a complete Digital Product Genome
with:
```

✅ **Approved Design**: Final visual specifications
✅ **Technical Specifications**: All manufacturing details
✅ **Material Requirements**: Complete material specifications
✅ **Dimensional Data**: Precise measurements and tolerances

Next Steps:
1. **Review & Approve**: Final review of all specifications
2. **Supplier Matching**: I'll find qualified manufacturers
3. **RFQ Generation**: Create professional requests for quotes
4. **Quote Analysis**: Compare and recommend best options

Would you like to review the complete specifications before we proceed to sourcing?"""

```
            self._log_agent_response(dpg, completion_response)
            return {"response": completion_response}

    def _identify_missing_specifications(self, dpg: DigitalProductGenome) -> list:
        """Identify which specifications are still needed for the product."""
        current_data = dpg.data
        missing_specs = []

        # Core specifications that every product needs
        required_specs = [
            {"key": "materials", "name": "Materials & Composition", "priority": 1},
            {"key": "dimensions", "name": "Dimensions & Measurements", "priority": 1},
            {"key": "colors", "name": "Color Specifications", "priority": 2},
            {"key": "quantities", "name": "Production Quantities", "priority": 1},
            {"key": "quality_standards", "name": "Quality Standards", "priority": 2},
            {"key": "packaging", "name": "Packaging Requirements", "priority": 3},
            {"key": "labeling", "name": "Labeling & Branding", "priority": 3}
        ]

        for spec in required_specs:
            if not current_data.get(spec["key"]) or current_data.get(spec["key"]) == "":
                missing_specs.append(spec)

        # Sort by priority (1 = most critical)
        missing_specs.sort(key=lambda x: x["priority"])
        return missing_specs

    def _generate_specification_question(self, spec: dict, dpg: DigitalProductGenome) -> str:
        """Generate an intelligent question about a missing specification."""
        spec_key = spec["key"]
        product_summary = dpg.summary
```

```
        question_templates = {
            "materials": f"Based on your {product_summary}, I'd recommend considering these material options. What material preferences do you have? (e.g., waterproof, sustainable, specific fabric types, etc.)",
            "dimensions": f"I need the key dimensions for manufacturing. Could you provide the main measurements? If you have existing samples or references, those dimensions would be perfect.",
            "colors": f"What colors would you like for this product? Please specify if you need exact color matching (like Pantone colors) or if general color descriptions work.",
            "quantities": f"What quantity are you looking to produce? This helps determine the best manufacturing approach and pricing tiers.",
            "quality_standards": f"Are there specific quality standards or certifications required? (e.g., safety standards, durability requirements, regulatory compliance)",
            "packaging": f"What are your packaging requirements? (e.g., individual packaging, bulk packaging, custom branding on packaging)",
            "labeling": f"What labeling and branding elements need to be included? (e.g., care labels, size labels, brand logos, regulatory text)"
        }

        return question_templates.get(spec_key, f"Could you provide details about {spec['name']}?")

    def _handle_finalized_stage(self, dpg: DigitalProductGenome, user_message: str) -> dict:
        """Handle interactions when DPG is in reviewed or approved stage."""
        if dpg.stage == 'reviewed':
            # Check if user is approving or requesting changes
            approval_prompt = f"""
            The user's message: "{user_message}"

            Are they:
            - APPROVING the specifications to move forward
            - REQUESTING changes or modifications
            - ASKING questions about the specifications

            Respond with just the intent: APPROVING, REQUESTING_CHANGES, or ASKING_QUESTIONS
            """

            try:
                intent_response = self.llm.invoke([SystemMessage(content=approval_prompt)])
                intent = intent_response.content.strip()

                if "APPROVING" in intent:
```

```
                dpg.stage = 'approved'
                dpg.save()

                response_text = """Perfect! Your Digital Product Genome is now complete and approved.

🎉 **Ready for Manufacturing**

**Next Steps:**
1. **Supplier Matching**: I'll identify qualified manufacturers based on your specifications
2. **RFQ Generation**: Create professional requests for quotes
3. **Quote Comparison**: Analyze and compare supplier proposals
4. **Production Management**: Coordinate the manufacturing process

Would you like me to start finding suppliers and generating RFQs for your product?"""

                self._log_agent_response(dpg, response_text)
                return {"response": response_text}

            elif "REQUESTING_CHANGES" in intent:
                dpg.stage = 'specifying'  # Go back to specification stage
                dpg.save()

                response_text = "No problem! I can help you make those changes. What would you like to modify in the specifications?"
                self._log_agent_response(dpg, response_text)
                return {"response": response_text}
            else:  # ASKING_QUESTIONS
                return self._answer_specification_question(dpg, user_message)

        except Exception as e:
            logger.error(f"Failed to parse approval intent: {e}")

    elif dpg.stage == 'approved':
        # DPG is locked - offer to create new version or proceed to sourcing
        response_text = """This product specification is finalized and locked.

**Options:**
- **Start Sourcing**: Find suppliers and get quotes for this design
- **Create New Version**: Make a copy to modify the specifications
- **New Product**: Start fresh with a different product

What would you like to do?"""
```

```
            self._log_agent_response(dpg, response_text)
            return {"response": response_text}

        # Fallback
        response_text = "I'm not sure how to proceed at this stage. Could you clarify what you'd like to do next?"
        self._log_agent_response(dpg, response_text)
        return {"response": response_text}

    def _answer_specification_question(self, dpg: DigitalProductGenome, user_message: str) -> dict:
        """Answer questions about the current specifications."""
        context = f"""
        Current DPG Data: {json.dumps(dpg.data, indent=2)}
        Product Summary: {dpg.summary}
        Stage: {dpg.stage}
        """

        qa_prompt = f"""
        Answer this question about the product specifications:

        Question: "{user_message}"

        Product Context:
        {context}

        Provide a clear, helpful answer based on the current specifications.
        """

        try:
            response = self.llm.invoke([SystemMessage(content=qa_prompt)])
            answer = response.content
            self._log_agent_response(dpg, answer)
            return {"response": answer}
        except Exception as e:
            logger.error(f"Specification Q&A failed: {e}")
            fallback = "I'm having trouble accessing the specification details. Could you be more specific about what you'd like to know?"
            self._log_agent_response(dpg, fallback)
            return {"response": fallback}

    def _synthesize_project_context(self) -> str:
        """
        Synthesize all project knowledge into a coherent product brief.
```

```
    Uses vector database retrieval to get relevant context.
    """
    try:
        # Get all knowledge chunks for this project
        project_chunks = KnowledgeChunk.objects.filter(
            metadata__project_id=self.project_id
        ).order_by('-created_at')[:20]  # Limit for token management

        if not project_chunks.exists():
            return "No specific context files found. Ready to work with your product description."

        # Combine content from all chunks
        combined_context = "\n\n".join([
            f"[{chunk.metadata.get('filename', 'Unknown')}]: {chunk.content}"
            for chunk in project_chunks
        ])

        # Use LLM to synthesize into coherent brief
        synthesis_prompt = f"""
        Analyze these project documents and create a coherent product brief:

        {combined_context[:8000]}  # Limit context length

        Create a clear, concise product brief that includes:
        - What the product is
        - Key features and requirements
        - Any aesthetic or functional preferences
        - Target use case or market

        Keep it focused and actionable for design generation.
        """

        response = self.llm.invoke([SystemMessage(content=synthesis_prompt)])
        return response.content

    except Exception as e:
        logger.error(f"Context synthesis failed: {e}")
        return "Unable to synthesize project context. Please describe what you'd like to create."

def _get_relevant_context(self, query: str) -> str:
    """Get relevant context from knowledge base for answering questions."""
    try:
        # Simple relevance scoring - in production, use proper vector search
        relevant_chunks = KnowledgeChunk.objects.filter(
```

```
                Q(content__icontains=query.split()[0]) |
                Q(metadata__project_id=self.project_id)
            )[:5]

            return "\n".join([chunk.content for chunk in relevant_chunks])

        except Exception as e:
            logger.error(f"Context retrieval failed: {e}")
            return "No additional context available."

    def _process_uploaded_files(self, files: list):
        """Process uploaded files - typically handled by Celery workers."""
        # In production, this would trigger Celery tasks for:
        # - Image analysis and feature extraction
        # - Document parsing and content extraction
        # - Vector embedding generation
        # - Knowledge base population
        logger.info(f"Would process {len(files)} files for project {self.project_id}")
        pass
```

🌐 Part IV: API Layer & Integration

4.1 Enhanced Project ViewSet

File: `backend/projects/views.py`

```
from rest_framework import viewsets, status, permissions
from rest_framework.response import Response
from rest_framework.decorators import action
from django.db import transaction
from .models import Project, ProjectMember, ProjectStage
from .serializers import ProjectSerializer, ProjectMemberSerializer, ProjectStageSerializer
from .permissions import IsProjectMember, IsProjectOwner
from agentcore.agent import ConversationalAgent
import logging

logger = logging.getLogger(__name__)

class ProjectViewSet(viewsets.ModelViewSet):
    """
    Enhanced project management with integrated AI agent.
    All conversations now happen within project context.
```

```
"""
serializer_class = ProjectSerializer
permission_classes = [permissions.IsAuthenticated, IsProjectMember]

def get_queryset(self):
    return Project.objects.filter(members=self.request.user).distinct()

@transaction.atomic
def perform_create(self, serializer):
    project = serializer.save(owner=self.request.user)
    ProjectMember.objects.create(
        user=self.request.user,
        project=project,
        role=ProjectMember.MemberRole.OWNER
    )

@action(
    detail=True,
    methods=['post'],
    url_path='chat',
    permission_classes=[IsProjectMember]
)
def chat(self, request, pk=None):
    """
    CRITICAL: The new stateful, project-scoped chat endpoint.
    This replaces all previous stateless chat implementations.
    URL: POST /api/projects/{project_id}/chat/
    """
    project = self.get_object()
    user_message = request.data.get("message", "").strip()
    files = request.data.get("files", [])

    if not user_message:
        return Response(
            {"error": "Message field is required"},
            status=status.HTTP_400_BAD_REQUEST
        )

    try:
        # Initialize agent with project and user context
        agent = ConversationalAgent(
            project_id=project.pk,
            user_id=request.user.pk
        )
```

```
            # Process message through agent's state machine
            agent_response = agent.chat(user_message, files)

            logger.info(f"Agent response generated for project {project.pk}")
            return Response(agent_response, status=status.HTTP_200_OK)

        except Exception as e:
            logger.error(f"Agent failed for project {project.pk}: {e}")
            return Response(
                {"error": "Failed to process your message. Please try again."},
                status=status.HTTP_500_INTERNAL_SERVER_ERROR
            )

    @action(
        detail=True,
        methods=['get'],
        url_path='dpg',
        permission_classes=[IsProjectMember]
    )
    def get_active_dpg(self, request, pk=None):
        """Get the currently active DPG for this project."""
        project = self.get_object()

        if project.active_dpg:
            from dpgs.serializers import DigitalProductGenomeSerializer
            serializer = DigitalProductGenomeSerializer(project.active_dpg)
            return Response(serializer.data)
        else:
            return Response(
                {"message": "No active DPG for this project"},
                status=status.HTTP_204_NO_CONTENT
            )

    @action(
        detail=True,
        methods=['post'],
        url_path='dpg/approve',
        permission_classes=[IsProjectMember]
    )
    def approve_dpg(self, request, pk=None):
        """Approve the active DPG and lock it for sourcing."""
        project = self.get_object()
```

```
        if not project.active_dpg:
            return Response(
                {"error": "No active DPG to approve"},
                status=status.HTTP_400_BAD_REQUEST
            )

        if project.active_dpg.stage != 'reviewed':
            return Response(
                {"error": "DPG must be in 'reviewed' stage to approve"},
                status=status.HTTP_400_BAD_REQUEST
            )

        project.active_dpg.stage = 'approved'
        project.active_dpg.save()

        return Response({"message": "DPG approved and locked for sourcing"})

    @action(detail=True, methods=['get'])
    def members(self, request, pk=None):
        """Get all project members."""
        project = self.get_object()
        members = project.memberships.all()
        serializer = ProjectMemberSerializer(members, many=True)
        return Response(serializer.data)

    @action(detail=True, methods=['post'])
    def add_member(self, request, pk=None):
        """Add a new member to the project."""
        project = self.get_object()

        # Only owners can add members
        if not project.memberships.filter(
            user=request.user,
            role=ProjectMember.MemberRole.OWNER
        ).exists():
            return Response(
                {"error": "Only project owners can add members"},
                status=status.HTTP_403_FORBIDDEN
            )

        user_id = request.data.get('user_id')
        role = request.data.get('role', ProjectMember.MemberRole.VIEWER)

        try:
```

```
            from django.contrib.auth import get_user_model
            User = get_user_model()
            user = User.objects.get(pk=user_id)

            member, created = ProjectMember.objects.get_or_create(
                project=project,
                user=user,
                defaults={'role': role}
            )

            if created:
                serializer = ProjectMemberSerializer(member)
                return Response(serializer.data, status=status.HTTP_201_CREATED)
            else:
                return Response(
                    {"error": "User is already a member"},
                    status=status.HTTP_400_BAD_REQUEST
                )

        except User.DoesNotExist:
            return Response(
                {"error": "User not found"},
                status=status.HTTP_404_NOT_FOUND
            )

class ProjectStageViewSet(viewsets.ModelViewSet):
    """Manage project stages - admin only."""
    queryset = ProjectStage.objects.all()
    serializer_class = ProjectStageSerializer
    permission_classes = [permissions.IsAdminUser]
```

4.2 Enhanced DPG ViewSet

File: `backend/dpgs/views.py`

```
from rest_framework import viewsets, status, permissions
from rest_framework.response import Response
from rest_framework.decorators import action
from django.db import transaction
from .models import DigitalProductGenome
from .serializers import DigitalProductGenomeSerializer
from projects.permissions import IsProjectMember
import logging
```

```
logger = logging.getLogger(__name__)

class DigitalProductGenomeViewSet(viewsets.ModelViewSet):
    """
    Enhanced DPG management with full lifecycle support.
    """
    serializer_class = DigitalProductGenomeSerializer
    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):
        # Filter DPGs to only those in projects where user is a member
        return DigitalProductGenome.objects.filter(
            project__members=self.request.user
        ).distinct().order_by('-created_at')

    @action(detail=True, methods=['get'], url_path='visual-assets')
    def get_visual_assets(self, request, pk=None):
        """Get all visual assets for a DPG with version history."""
        dpg = self.get_object()
        return Response({
            "dpg_id": dpg.id,
            "visual_assets": dpg.visual_assets,
            "stage": dpg.stage
        })

    @action(detail=True, methods=['get'], url_path='conversation-history')
    def get_conversation_history(self, request, pk=None):
        """Get complete conversation history for this DPG."""
        dpg = self.get_object()
        return Response({
            "dpg_id": dpg.id,
            "conversation_history": dpg.conversation_history,
            "stage": dpg.stage,
            "last_updated": dpg.updated_at
        })

    @action(detail=True, methods=['post'], url_path='create-version')
    def create_version(self, request, pk=None):
        """Create a new version of this DPG for further iteration."""
        original_dpg = self.get_object()

        if original_dpg.stage != 'approved':
            return Response(
```

```
                {"error": "Can only create versions from approved DPGs"},
                status=status.HTTP_400_BAD_REQUEST
            )

        try:
            with transaction.atomic():
                # Create new version
                new_version_number = f"{float(original_dpg.version) + 0.1:.1f}"

                new_dpg = DigitalProductGenome.objects.create(
                    project=original_dpg.project,
                    owner=request.user,
                    title=f"{original_dpg.title} v{new_version_number}",
                    description=original_dpg.description,
                    version=new_version_number,
                    summary=original_dpg.summary,
                    data=original_dpg.data.copy(),
                    visual_assets=original_dpg.visual_assets.copy(),
                    stage='designing'  # Start in design stage for iteration
                )

                # Update project's active DPG
                original_dpg.project.active_dpg = new_dpg
                original_dpg.project.save()

                serializer = self.get_serializer(new_dpg)
                return Response(serializer.data, status=status.HTTP_201_CREATED)

        except Exception as e:
            logger.error(f"Failed to create DPG version: {e}")
            return Response(
                {"error": "Failed to create new version"},
                status=status.HTTP_500_INTERNAL_SERVER_ERROR
            )

    @action(detail=True, methods=['post'], url_path='generate-rfq')
    def generate_rfq(self, request, pk=None):
        """Generate RFQ from approved DPG specifications."""
        dpg = self.get_object()

        if dpg.stage != 'approved':
            return Response(
                {"error": "DPG must be approved before generating RFQ"},
                status=status.HTTP_400_BAD_REQUEST
```

```
            )

        try:
            # Import RFQ generation logic
            from rfq.services import RFQGenerationService
            rfq_service = RFQGenerationService()
            rfq_data = rfq_service.generate_from_dpg(dpg)

            return Response({
                "message": "RFQ generated successfully",
                "rfq_data": rfq_data,
                "dpg_id": dpg.id
            })

        except Exception as e:
            logger.error(f"RFQ generation failed for DPG {dpg.id}: {e}")
            return Response(
                {"error": "Failed to generate RFQ"},
                status=status.HTTP_500_INTERNAL_SERVER_ERROR
            )

    @action(detail=True, methods=['get'], url_path='manufacturing-analysis')
    def manufacturing_analysis(self, request, pk=None):
        """Get AI analysis of manufacturability and recommendations."""
        dpg = self.get_object()

        try:
            from agentcore.tools import manufacturing_analysis_tool
            analysis = manufacturing_analysis_tool(dpg.data)

            return Response({
                "dpg_id": dpg.id,
                "analysis": analysis,
                "recommendations": analysis.get('recommendations', []),
                "feasibility_score": analysis.get('feasibility_score', 0)
            })

        except Exception as e:
            logger.error(f"Manufacturing analysis failed for DPG {dpg.id}: {e}")
            return Response(
                {"error": "Analysis temporarily unavailable"},
                status=status.HTTP_500_INTERNAL_SERVER_ERROR
            )
```

🚀 Part V: Implementation Roadmap

Phase 1: Foundation (Weeks 1-4)

Week 1: Core Models & Database

- [ ] Implement enhanced Project and DigitalProductGenome models
- [ ] Create and run database migrations
- [ ] Set up proper foreign key relationships
- [ ] Add data validation and constraints
- [ ] Create database indexes for performance

Week 2: Generative Engine

- [ ] Implement ImageGenerationService with Replicate integration
- [ ] Set up S3 bucket and asset persistence logic
- [ ] Create LangChain tools interface
- [ ] Add comprehensive error handling and logging
- [ ] Test image generation pipeline end-to-end

Week 3: Agent Core Architecture

- [ ] Implement ConversationalAgent class structure
- [ ] Build state machine logic for DPG lifecycle stages
- [ ] Create conversation memory and context management
- [ ] Add LLM integration with proper prompt engineering
- [ ] Implement basic tool orchestration

Week 4: API Integration

- [ ] Refactor ProjectViewSet with new chat endpoint
- [ ] Enhanced DigitalProductGenomeViewSet with lifecycle actions
- [ ] Remove deprecated chat endpoints and views
- [ ] Update URL routing for new architecture
- [ ] Add comprehensive API documentation

Phase 2: Intelligence Enhancement (Weeks 5-8)

Week 5: Advanced Agent Capabilities

- [ ] Implement intelligent context synthesis from uploaded files
- [ ] Add sophisticated intent analysis and classification
- [ ] Build design modification logic with inpainting
- [ ] Create specification completion workflows
- [ ] Add manufacturing feasibility analysis

Week 6: Knowledge Base Integration

- [ ] Enhance vector database integration for RAG
- [ ] Implement project-specific context retrieval
- [ ] Add file processing and embedding generation
- [ ] Create intelligent question-answering system
- [ ] Build context-aware recommendation engine

Week 7: Visual Asset Management

- [ ] Implement complete asset versioning system
- [ ] Add asset comparison and diff visualization
- [ ] Create asset export and sharing capabilities
- [ ] Build visual asset search and filtering
- [ ] Add batch asset operations

Week 8: Quality Assurance

- [ ] Comprehensive testing of all agent workflows
- [ ] Load testing for concurrent users
- [ ] Error handling and recovery mechanisms
- [ ] Performance optimization and caching
- [ ] Security audit and vulnerability assessment

Phase 3: Advanced Features (Weeks 9-12)

Week 9: Supplier Integration Foundation

- [ ] Design supplier database schema
- [ ] Implement supplier verification workflows
- [ ] Create communication logging system
- [ ] Build supplier capability matching
- [ ] Add supplier risk assessment

Week 10: RFQ Generation & Management

- [ ] Advanced RFQ generation with templates
- [ ] Multi-format RFQ export (PDF, Word, etc.)
- [ ] Quote comparison and analysis tools
- [ ] Supplier response management
- [ ] Automated follow-up workflows

Week 11: Process Automation

- [ ] Implement Celery for background tasks
- [ ] Add email/SMS notification system

- [ ] Create workflow automation engine
- [ ] Build progress tracking dashboard
- [ ] Add milestone and deadline management

Week 12: Analytics & Optimization

- [ ] User behavior analytics and insights
- [ ] Agent performance metrics and monitoring
- [ ] A/B testing framework for agent improvements
- [ ] Cost and time savings measurement
- [ ] Continuous learning and model updates

Phase 4: Production Readiness (Weeks 13-16)

Week 13: Scalability & Performance

- [ ] Database optimization and query tuning
- [ ] Redis caching implementation
- [ ] CDN setup for static assets
- [ ] Load balancing configuration
- [ ] Auto-scaling infrastructure

Week 14: Security & Compliance

- [ ] Authentication and authorization hardening
- [ ] Data encryption at rest and in transit
- [ ] API rate limiting and abuse prevention
- [ ] GDPR compliance implementation
- [ ] Security penetration testing

Week 15: Monitoring & Observability

- [ ] Application performance monitoring (APM)
- [ ] Error tracking and alerting system
- [ ] Log aggregation and analysis
- [ ] Business metrics dashboard
- [ ] Health checks and status pages

Week 16: Launch Preparation

- [ ] User acceptance testing (UAT)
- [ ] Production deployment pipeline
- [ ] Backup and disaster recovery testing
- [ ] Documentation and training materials
- [ ] Go-live checklist and rollback plan

🎯 Part VI: Advanced Intelligence Features

6.1 Enhanced Agent Tools

File: `backend/agentcore/advanced_tools.py`

```
from langchain_core.tools import tool
from typing import Dict, List, Any
import json
import logging
from decouple import config
from langchain_openai import ChatOpenAI

logger = logging.getLogger(__name__)

@tool
def manufacturing_analysis_tool(dpg_data: dict) -> dict:
    """
    Analyze product specifications for manufacturability and provide recommendations.
    """
    llm = ChatOpenAI(model_name=config("OPENAI_MODEL", default="gpt-4-turbo"))

    analysis_prompt = f"""
    Analyze this product specification for manufacturability:

    Product Data: {json.dumps(dpg_data, indent=2)}

    Provide analysis in JSON format:
    {{
        "feasibility_score": 0-100,
        "complexity_level": "low|medium|high",
        "estimated_cost_range": "cost estimate",
        "production_timeline": "estimated timeline",
        "manufacturing_methods": ["method1", "method2"],
        "potential_challenges": ["challenge1", "challenge2"],
        "recommendations": [
            {{"type": "optimization", "description": "...", "impact": "cost_savings|quality|timeline"}},
            {{"type": "alternative", "description": "...", "benefit": "..."}}
        ],
        "required_certifications": ["cert1", "cert2"],
        "compliance_considerations": ["consideration1", "consideration2"]
    }}
    """
```

```
    try:
        response = llm.invoke(analysis_prompt)
        return json.loads(response.content)
    except Exception as e:
        logger.error(f"Manufacturing analysis failed: {e}")
        return {"error": "Analysis temporarily unavailable"}

@tool
def supplier_matching_tool(dpg_data: dict, requirements: dict = None) -> list:
    """
    Match product specifications with suitable suppliers.
    """
    # In production, this would query a real supplier database
    # For now, we'll use AI to generate realistic supplier recommendations
    llm = ChatOpenAI(model_name=config("OPENAI_MODEL", default="gpt-4-turbo"))

    matching_prompt = f"""
    Based on this product specification, recommend suitable supplier types:

    Product: {json.dumps(dpg_data, indent=2)}
    Requirements: {json.dumps(requirements or {}, indent=2)}

    Respond with JSON array of supplier recommendations:
    [
        {{
            "supplier_type": "type of manufacturer",
            "location_preferences": ["country1", "country2"],
            "capabilities_needed": ["capability1", "capability2"],
            "certifications_required": ["cert1", "cert2"],
            "estimated_moq": "minimum order quantity",
            "lead_time_estimate": "production timeline",
            "cost_tier": "low|medium|high",
            "quality_tier": "standard|premium|luxury"
        }}
    ]
    """

    try:
        response = llm.invoke(matching_prompt)
        return json.loads(response.content)
    except Exception as e:
        logger.error(f"Supplier matching failed: {e}")
        return [{"error": "Supplier matching temporarily unavailable"}]
```

```
@tool
def compliance_checker_tool(dpg_data: dict, target_markets: list = None) -> dict:
    """
    Check compliance requirements for target markets.
    """
    llm = ChatOpenAI(model_name=config("OPENAI_MODEL", default="gpt-4-turbo"))

    compliance_prompt = f"""
    Check compliance requirements for this product:

    Product: {json.dumps(dpg_data, indent=2)}
    Target Markets: {target_markets or ["US", "EU"]}

    Respond with JSON:
    {{
        "compliance_summary": "overview of requirements",
        "required_standards": [
            {{"standard": "ISO 9001", "description": "...", "mandatory": true}},
            {{"standard": "CE Marking", "description": "...", "mandatory": false}}
        ],
        "testing_requirements": [
            {{"test": "safety testing", "description": "...", "estimated_cost": "...", "timeline": "..."}}
        ],
        "labeling_requirements": ["requirement1", "requirement2"],
        "documentation_needed": ["doc1", "doc2"],
        "estimated_compliance_cost": "cost range",
        "compliance_timeline": "estimated timeline"
    }}
    """

    try:
        response = llm.invoke(compliance_prompt)
        return json.loads(response.content)
    except Exception as e:
        logger.error(f"Compliance checking failed: {e}")
        return {"error": "Compliance checking temporarily unavailable"}

@tool
def cost_estimation_tool(dpg_data: dict, quantities: list = None) -> dict:
    """
    Estimate production costs at different quantities.
    """
    llm = ChatOpenAI(model_name=config("OPENAI_MODEL", default="gpt-4-turbo"))
    quantities = quantities or [100, 500, 1000, 5000, 10000]
```

```
    cost_prompt = f"""
    Estimate production costs for this product at different quantities:

    Product: {json.dumps(dpg_data, indent=2)}
    Quantities: {quantities}

    Respond with JSON:
    {{
        "cost_breakdown": {{
            "materials": "percentage of total cost",
            "labor": "percentage of total cost",
            "overhead": "percentage of total cost",
            "tooling": "one-time cost"
        }},
        "quantity_pricing": [
            {{"qty": 100, "unit_cost": "$X.XX", "total_cost": "$XXX", "notes": "setup costs impact"}},
            {{"qty": 500, "unit_cost": "$X.XX", "total_cost": "$XXX", "notes": "..."}},
            {{"qty": 1000, "unit_cost": "$X.XX", "total_cost": "$XXX", "notes": "..."}}
        ],
        "cost_drivers": ["main factors affecting cost"],
        "optimization_opportunities": ["ways to reduce costs"],
        "break_even_analysis": "quantity for best value"
    }}
    """

    try:
        response = llm.invoke(cost_prompt)
        return json.loads(response.content)
    except Exception as e:
        logger.error(f"Cost estimation failed: {e}")
        return {"error": "Cost estimation temporarily unavailable"}

@tool
def quality_standards_recommender_tool(dpg_data: dict, product_category: str) -> dict:
    """
    Recommend appropriate quality standards and testing procedures.
    """
    llm = ChatOpenAI(model_name=config("OPENAI_MODEL", default="gpt-4-turbo"))

    quality_prompt = f"""
    Recommend quality standards for this product:

    Product: {json.dumps(dpg_data, indent=2)}
```

```
    Category: {product_category}

    Respond with JSON:
    {{
        "recommended_standards": [
            {{"standard": "ISO 9001", "relevance": "high|medium|low", "description": "..."}},
            {{"standard": "ASTM D123", "relevance": "high|medium|low", "description": "..."}}
        ],
        "quality_tests": [
            {{"test": "durability testing", "description": "...", "cost_estimate": "..."}}
        ],
        "inspection_points": ["point1", "point2"],
        "quality_metrics": [
            {{"metric": "defect rate", "target": "< 1%", "frequency": "per batch|sample",
"measurement": "how to measure"}}
        ],
        "supplier_quality_requirements": ["requirement1", "requirement2"]
    }}
    """

    try:
        response = llm.invoke(quality_prompt)
        return json.loads(response.content)
    except Exception as e:
        logger.error(f"Quality standards recommendation failed: {e}")
        return {"error": "Quality recommendations temporarily unavailable"}
```

6.2 RFQ Generation Service

File: `backend/rfq/services.py`

```
import json
from datetime import datetime, timedelta
from decouple import config
from langchain_openai import ChatOpenAI
from dpgs.models import DigitalProductGenome
import logging

logger = logging.getLogger(__name__)

class RFQGenerationService:
    """
    Professional RFQ generation service that creates comprehensive,
```

```
    industry-standard requests for quotes.
    """

    def __init__(self):
        self.llm = ChatOpenAI(
            model_name=config("OPENAI_MODEL", default="gpt-4-turbo"),
            temperature=0.1  # Low temperature for consistent, professional output
        )

    def generate_from_dpg(self, dpg: DigitalProductGenome) -> dict:
        """Generate comprehensive RFQ from approved DPG."""
        try:
            rfq_content = self._generate_rfq_content(dpg)
            rfq_data = {
                "rfq_id": f"RFQ-{dpg.project.id}-{dpg.id}-{datetime.now().strftime('%Y%m%d')}",
                "generated_at": datetime.now().isoformat(),
                "dpg_id": dpg.id,
                "project_id": dpg.project.id,
                "content": rfq_content,
                "response_deadline": (datetime.now() + timedelta(days=14)).isoformat(),
                "estimated_quantities": self._extract_quantities(dpg.data),
                "target_delivery": self._calculate_target_delivery(dpg.data)
            }
            return rfq_data
        except Exception as e:
            logger.error(f"RFQ generation failed for DPG {dpg.id}: {e}")
            raise

    def _generate_rfq_content(self, dpg: DigitalProductGenome) -> str:
        """Generate the main RFQ document content."""
        system_prompt = """You are a professional procurement specialist creating an RFQ
(Request for Quote) document.

Generate a comprehensive, professional RFQ that suppliers can easily understand and respond
to accurately.

Structure the RFQ with these sections:
1. EXECUTIVE SUMMARY
2. PRODUCT SPECIFICATIONS
3. TECHNICAL REQUIREMENTS
4. QUALITY STANDARDS & CERTIFICATIONS
5. QUANTITY & DELIVERY REQUIREMENTS
6. SUPPLIER QUALIFICATIONS
7. SUBMISSION REQUIREMENTS
```

```
8. EVALUATION CRITERIA
9. TERMS & CONDITIONS

Make it detailed, specific, and professional. Include all technical specifications clearly."""

        product_info = f"""
PROJECT INFORMATION:
Project: {dpg.project.name}
Product: {dpg.title}
Description: {dpg.description}
Summary: {dpg.summary}

TECHNICAL SPECIFICATIONS:
{json.dumps(dpg.data, indent=2)}

VISUAL ASSETS:
{json.dumps(dpg.visual_assets, indent=2)}

CONVERSATION CONTEXT:
{json.dumps(dpg.conversation_history[-10:], indent=2) if dpg.conversation_history else "No
conversation history"}
"""

        rfq_prompt = f"""
Create a comprehensive RFQ document for this product:

{product_info}

The RFQ should be professional, detailed, and include all necessary information for suppliers to
provide accurate quotes. Focus on manufacturability and clear specifications.
"""

        try:
            response = self.llm.invoke([
                {"role": "system", "content": system_prompt},
                {"role": "user", "content": rfq_prompt}
            ])
            return response.content
        except Exception as e:
            logger.error(f"RFQ content generation failed: {e}")
            raise

    def _extract_quantities(self, dpg_data: dict) -> list:
        """Extract quantity requirements from DPG data."""
```

```
        quantities = dpg_data.get('quantities', [])
        if not quantities:
            # Default quantity tiers if not specified
            quantities = [100, 500, 1000, 5000]
        return quantities

    def _calculate_target_delivery(self, dpg_data: dict) -> str:
        """Calculate target delivery date based on requirements."""
        target_date = dpg_data.get('target_delivery')
        if target_date:
            return target_date

        # Default to 8 weeks from now if not specified
        default_delivery = datetime.now() + timedelta(weeks=8)
        return default_delivery.strftime('%Y-%m-%d')

class QuoteAnalysisService:
    """
    Service for analyzing and comparing supplier quotes.
    """

    def __init__(self):
        self.llm = ChatOpenAI(
            model_name=config("OPENAI_MODEL", default="gpt-4-turbo"),
            temperature=0.2
        )

    def analyze_quotes(self, quotes: list, rfq_data: dict) -> dict:
        """Analyze multiple quotes and provide recommendations."""
        analysis_prompt = f"""
        Analyze these supplier quotes and provide a comprehensive comparison:

        RFQ Requirements: {json.dumps(rfq_data, indent=2)}
        Supplier Quotes: {json.dumps(quotes, indent=2)}

        Provide analysis in JSON format:
        {{
            "quote_summary": {{
                "total_quotes": number,
                "price_range": "lowest - highest",
                "average_lead_time": "weeks",
                "compliance_rate": "percentage meeting requirements"
            }},
            "detailed_comparison": [
```

```
        {{
            "supplier_name": "name",
            "total_score": 0-100,
            "price_score": 0-100,
            "quality_score": 0-100,
            "delivery_score": 0-100,
            "compliance_score": 0-100,
            "strengths": ["strength1", "strength2"],
            "concerns": ["concern1", "concern2"],
            "unit_price": "$X.XX",
            "total_cost": "$XXXX",
            "lead_time": "X weeks",
            "moq": "minimum order"
        }}
    ],
    "recommendations": [
        {{
            "rank": 1,
            "supplier": "supplier name",
            "rationale": "why this supplier is recommended",
            "risk_level": "low|medium|high",
            "confidence": 0-100
        }}
    ],
    "negotiation_opportunities": [
        {{
            "supplier": "supplier name",
            "opportunity": "what to negotiate",
            "potential_savings": "estimated savings"
        }}
    ],
    "red_flags": [
        {{
            "supplier": "supplier name",
            "issue": "concern description",
            "severity": "low|medium|high"
        }}
    ]
}}
"""

try:
    response = self.llm.invoke(analysis_prompt)
    return json.loads(response.content)
```

```
        except Exception as e:
            logger.error(f"Quote analysis failed: {e}")
            return {"error": "Quote analysis temporarily unavailable"}

    def generate_negotiation_strategy(self, analysis: dict, preferred_supplier: str) -> dict:
        """Generate negotiation strategy for preferred supplier."""
        strategy_prompt = f"""
        Based on this quote analysis, generate a negotiation strategy:

        Analysis: {json.dumps(analysis, indent=2)}
        Preferred Supplier: {preferred_supplier}

        Create a negotiation strategy in JSON format:
        {{
            "negotiation_objectives": ["objective1", "objective2"],
            "leverage_points": ["point1", "point2"],
            "negotiation_tactics": [
                {{
                    "tactic": "price negotiation",
                    "approach": "how to approach",
                    "expected_outcome": "what to expect"
                }}
            ],
            "fallback_options": ["option1", "option2"],
            "deal_breakers": ["breaker1", "breaker2"],
            "timeline": "negotiation timeline",
            "success_metrics": ["metric1", "metric2"]
        }}
        """

        try:
            response = self.llm.invoke(strategy_prompt)
            return json.loads(response.content)
        except Exception as e:
            logger.error(f"Negotiation strategy generation failed: {e}")
            return {"error": "Strategy generation temporarily unavailable"}
```

🌟 Part VII: The Complete Vision

7.1 Ultimate User Experience Flow

User Journey: "I want 10,000 custom phone cases"

1. PROJECT CREATION

User: "I want to create custom phone cases for my brand"

Agent: *Creates project, analyzes intent* → "Great! Let's design your custom phone cases. Could you upload your logo and any design references?"

2. CONTEXT SYNTHESIS

User: *Uploads logo, brand guidelines, reference images*

Agent: *Processes files, synthesizes context* → "I understand you want premium phone cases with your logo for iPhone 14/15. Based on your brand guidelines, I'm thinking sleek design with raised edges for protection. Shall I generate initial concepts?"

3. VISUAL DESIGN ITERATION

User: "Yes, but make them more minimalist"

Agent: *Generates concept sketches and technical flats* → "Here are 3 minimalist concepts with clean lines and subtle branding. Which direction resonates with your vision?"

User: "I like #2 but the logo should be smaller"

Agent: *Uses inpainting to modify logo size* → "Perfect! I've reduced the logo size. How does this new version look?"

4. SPECIFICATION COMPLETION

User: "That's exactly what I want!"

Agent: *Transitions to specification stage* → "Excellent! Now let's finalize the specs. For iPhone cases, I recommend flexible TPU with raised bezels. What quantity are you planning?"

User: "10,000 units, mixed between iPhone 14 and 15"

Agent: *Completes technical specifications* → "Perfect! I have all the specs. Ready for manufacturing quotes?"

5. SUPPLIER MATCHING & RFQ

User: "Yes, let's get quotes"

Agent: *Generates professional RFQ, matches suppliers* → "I've sent your RFQ to 8 qualified manufacturers in China and Vietnam. You should receive quotes within 48 hours."

6. QUOTE ANALYSIS & RECOMMENDATION

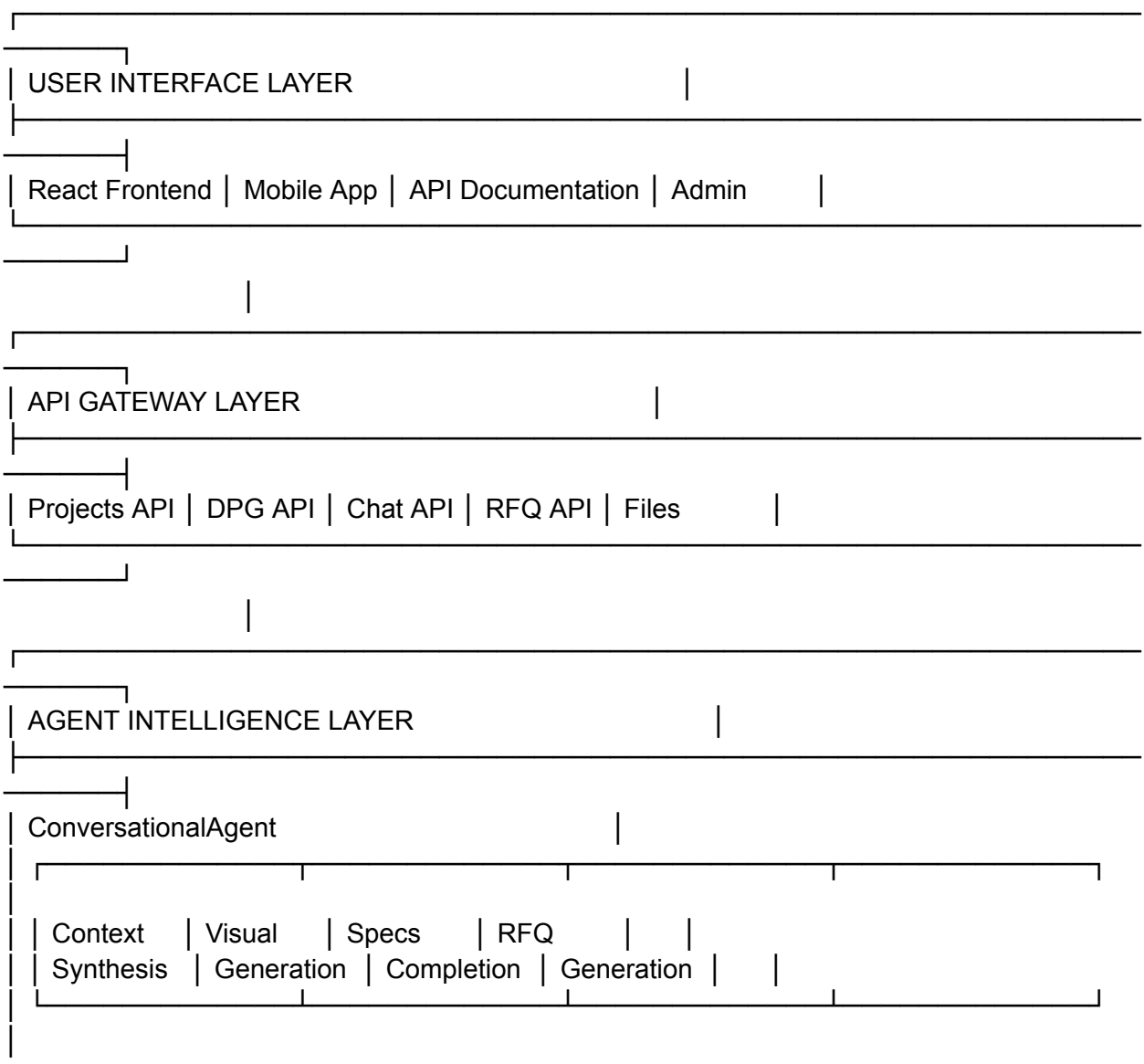
Agent: *Analyzes incoming quotes* → "I received 6 quotes. Based on price, quality, and reliability, I recommend Supplier A: $2.30/unit, 3-week delivery, ISO certified. Here's the full analysis..."

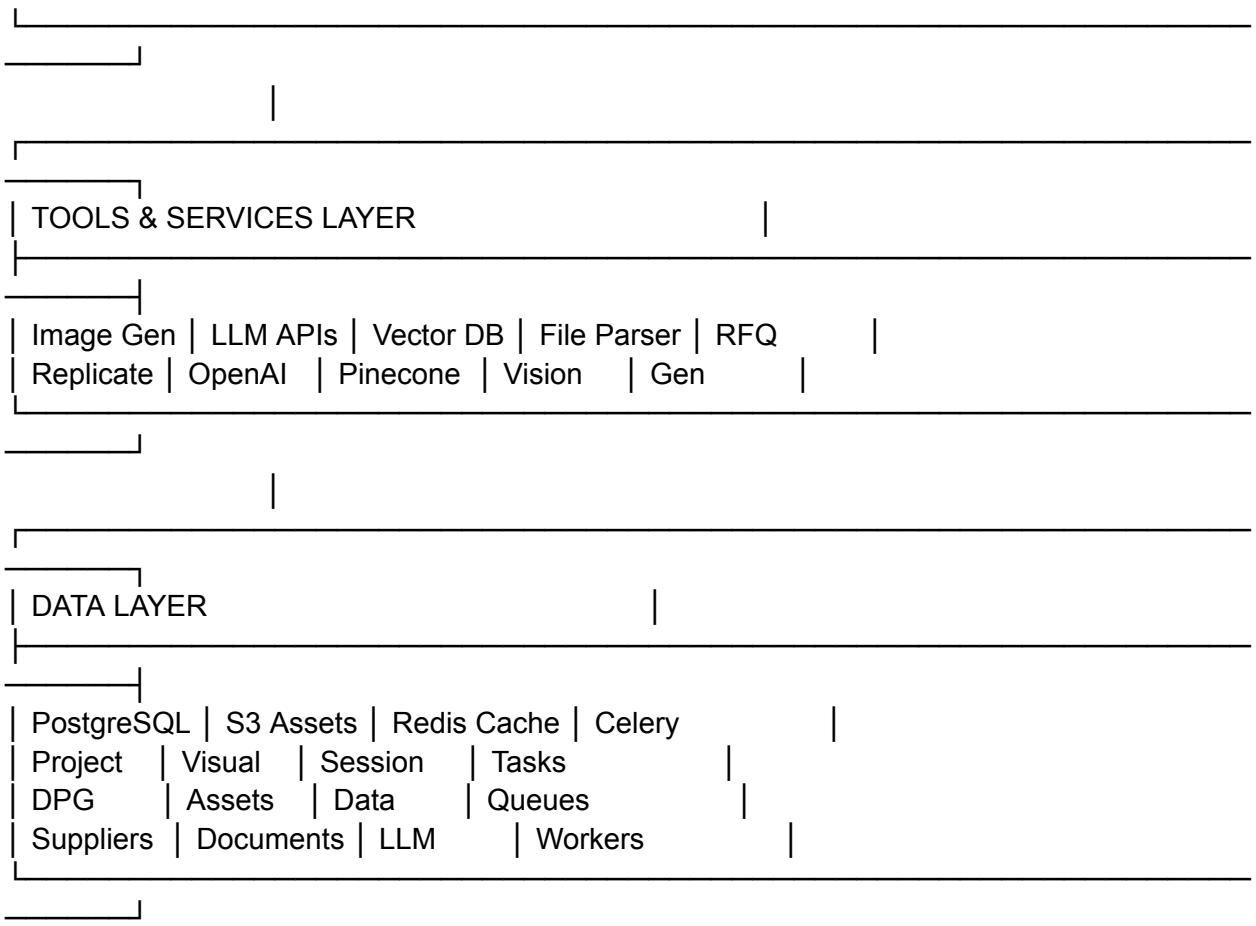
7. PRODUCTION MANAGEMENT

User: "Let's go with Supplier A"

Agent: *Handles contracting, payments, quality control* → "Order confirmed! I'll coordinate production, quality inspections, and shipping. Expected delivery: March 15th. I'll keep you updated on progress."

7.2 Technical Architecture Summary





7.3 Success Metrics & KPIs

User Experience Metrics

- **Time to First Visual:** < 5 minutes from project creation
- **Design Iteration Speed:** < 2 minutes per modification
- **Specification Completion:** < 15 minutes for complete specs
- **Quote Response Time:** < 24 hours for first quotes
- **End-to-End Time:** < 7 days from idea to production order

Quality Metrics

- **Design Accuracy:** 95% user satisfaction with generated visuals
- **Specification Completeness:** 99% manufacturability rate
- **Quote Accuracy:** 90% of quotes within 10% of actual cost
- **Supplier Match Quality:** 85% successful production completion
- **Defect Rate:** < 2% in delivered products

Business Impact

- **Cost Savings:** 20-40% vs traditional sourcing methods
- **Time Savings:** 70-80% faster than manual processes
- **User Retention:** 80% of users return for second project
- **Revenue Growth:** $10M ARR within 18 months
- **Market Penetration:** 1000+ active manufacturing projects

7.4 Competitive Advantages

1. **True AI Partnership:** Not just tools, but a proactive design partner
2. **Perfect Memory:** Every design decision and context is remembered
3. **Visual Intelligence:** Generate and iterate on actual product visuals
4. **End-to-End Orchestration:** From concept to delivered product
5. **Manufacturing Intelligence:** Real manufacturability analysis
6. **Supplier Network:** Curated, verified manufacturer relationships
7. **Quality Assurance:** Built-in quality control and inspection
8. **Cost Transparency:** Real-time cost analysis and optimization

🎯 Implementation Priority Matrix

Must Have (P0) - Launch Blockers

- [ ] Core agent conversation flow
- [ ] Visual asset generation and iteration
- [ ] Project-scoped conversations
- [ ] DPG lifecycle management
- [ ] Basic RFQ generation
- [ ] User authentication and permissions

Should Have (P1) - Early Value

- [ ] Advanced visual modifications (inpainting)
- [ ] Intelligent specification completion
- [ ] File upload and context synthesis
- [ ] Supplier matching recommendations
- [ ] Quote comparison and analysis
- [ ] Mobile-responsive interface

Could Have (P2) - Enhanced Experience

- [ ] Manufacturing feasibility analysis
- [ ] Compliance checking tools
- [ ] Cost estimation at scale
- [ ] Quality standards recommendations
- [ ] Advanced search and filtering

- [ ] Team collaboration features

Won't Have (P3) - Future Releases

- [ ] Real-time production monitoring
- [ ] IoT integration for factories
- [ ] Blockchain supply chain tracking
- [ ] AR/VR design visualization
- [ ] Voice interface
- [ ] Predictive market analytics

🚀 Conclusion: The Path to "Cursor for Hardware"

This comprehensive roadmap transforms your existing Unarchived platform into a revolutionary AI-first manufacturing operating system. By implementing this architecture, you will create:

🎯 **A True Design Partner:** An AI that doesn't just respond to requests, but proactively guides users through the entire product development journey.

🧠 **Perfect Memory:** Every conversation, design decision, and iteration is remembered and builds upon previous context.

🎨 **Visual Intelligence:** Generate, modify, and iterate on actual product visuals with professional manufacturing quality.

🔄 **End-to-End Orchestration:** From initial concept to delivered product, managing every step of the manufacturing process.

⚡ **Unprecedented Speed:** Reduce product development time from months to days, while maintaining professional quality.

The technical implementation is ambitious but achievable with the phased approach outlined. Each phase builds upon the previous, creating a robust, scalable platform that will fundamentally change how physical products are designed and manufactured.

The future of manufacturing is conversational, intelligent, and proactive. This roadmap is your blueprint for building that future.