HomeVision: AI-Powered Architectural Visualization Platform

Comprehensive System Design Document

---

1. Executive Summary

HomeVision is a "Lovable-for-houses" platform that democratizes the early-stage architectural design process. The platform enables home buyers to effectively communicate their vision to architects through interactive 3D models generated from simple inputs like budget, plot dimensions, style preferences, and reference images.

Core Value Proposition: Transform the 12-month, iterative architectural design process into a minutes-long visual exploration, reducing miscommunication and accelerating project kickoff.

---

2. Product Requirements Analysis

2.1 Input Parameters

Parameter	Type	Purpose
Budget	Currency range	Cost band estimation, material selection
Plot dimensions	Length × Width (feet/meters)	Constraint for floor plan generation
Target square footage	Numeric	Room sizing, floor count calculations
Number of bedrooms	Integer	Room allocation in floor plan
Stylistic keywords	Text array	Aesthetic guidance (modern, farmhouse, etc.)
Geographic location	Lat/Long or ZIP	Climate reasoning, regional cost adjustments
Reference images	Image uploads	Material palette extraction (flooring, roofing, facade, kitchen)

2.2 Output Deliverables

1. **Four distinct floor plan options** with room layouts
2. **3D massing models** for each floor plan variant
3. **Cost band estimates** per option with justification
4. **Material library** coherent with uploaded references

5. **Interactive 3D viewer** with click-to-edit capability
6. **Shareable link** for architect handoff

2.3 Scale Assumptions

- **Target users**: ~5 concurrent users (MVP phase)
- **Focus**: Correctness and quality over scalability
- **Deployment**: Single-region, no complex distributed architecture needed

---

3. Recommended Technology Stack

3.1 Frontend

Component	Technology	Justification
Framework	Next.js 14+ (App Router)	Server components, API routes, excellent DX, Vercel deployment
3D Rendering	React Three Fiber + @react-three/drei	Declarative 3D in React, rich ecosystem, production-proven
Styling	Tailwind CSS	Rapid UI development, consistent design system
State Management	Zustand	Lightweight, perfect for 3D scene state
UI Components	shadcn/ui	Accessible, customizable, works well with Tailwind
3D Model Loading	@react-three/gltfjsx	GLB/glTF loading and JSX conversion

3.2 Backend

Component	Technology	Justification
API Framework	FastAPI (Python)	Async support, type safety, AI/ML ecosystem compatibility
Task Queue	Celery + Redis	Background job processing for generation tasks
Auth	Supabase Auth	Quick setup, social providers, JWT tokens
File Storage	AWS S3 / Supabase Storage	3D model storage, image uploads, generated assets

3.3 Database: Supabase (Recommended)

Why Supabase is a good fit for this project:

1. **PostgreSQL foundation**: Relational data model suits project/user/generation relationships
2. **Built-in Auth**: Eliminates auth implementation complexity
3. **Storage integration**: Seamless file uploads for images and 3D models
4. **Real-time subscriptions**: Useful for generation progress updates
5. **Row Level Security**: Secure multi-user data isolation
6. **Generous free tier**: Perfect for MVP with 5 users
7. **pgvector extension**: Future-proof for semantic search on materials/styles

Alternatives considered:

- **Firebase**: NoSQL doesn't fit the relational nature of floor plans, rooms, and materials
- **PlanetScale**: Overkill for this scale; MySQL vs PostgreSQL preference
- **Neon**: Good alternative, but Supabase's auth/storage bundle is more convenient

3.4 AI/ML Services

Service	Provider	Model/Endpoint	Use Case
Image Generation	Fal AI	Nano Banana Pro (Gemini 3 Pro Image)	Reference image processing, floor plan rendering
Image-to-3D	Fal AI	Trellis 2	Massing model generation from rendered views
Floor Plan Generation	Custom + Claude/GPT-4	LangGraph workflow	Deterministic floor plan generation
Material Analysis	Claude API	claude-sonnet-4-20250514	Extract palettes from reference images
Cost Estimation	Custom + 1build API	Regional cost data	Construction cost band calculation

3.5 Image-to-3D Model Comparison

Based on my research, here's the comparison for your use case:

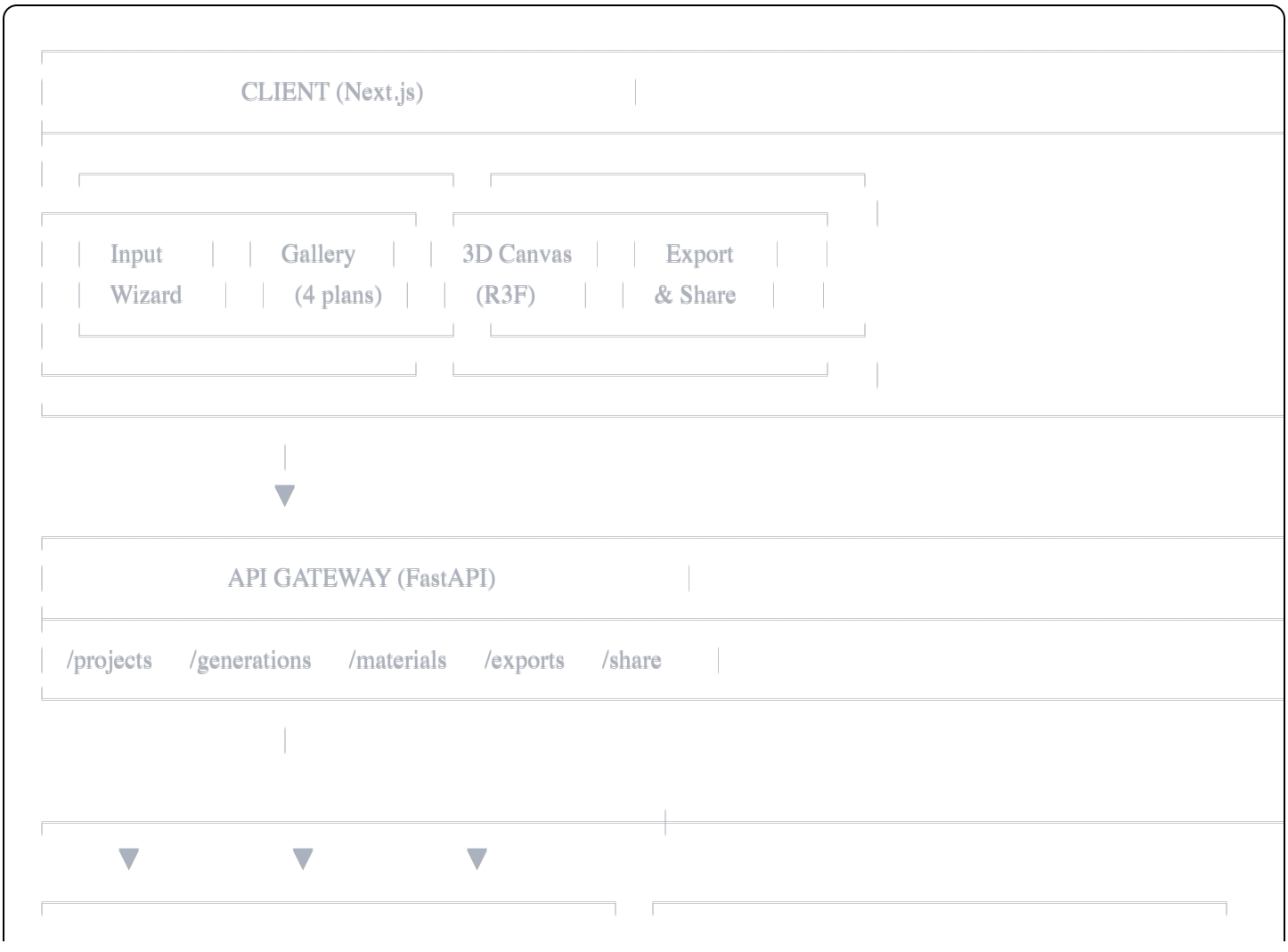
Model	Best For	Price	Quality	Architecture Suitability
Trellis 2	Hard-surface objects, buildings	$0.25-0.35/gen	High	✅ Excellent (increase ss_guidance_strength to 8-9)
Hyper3D Rodin	General purpose, text-to-3D	$0.40/gen	Very High	✅ Good, production-ready GLB output
TripoSR	Fast prototyping	Lower	Medium	⚠️ Less accurate for architecture

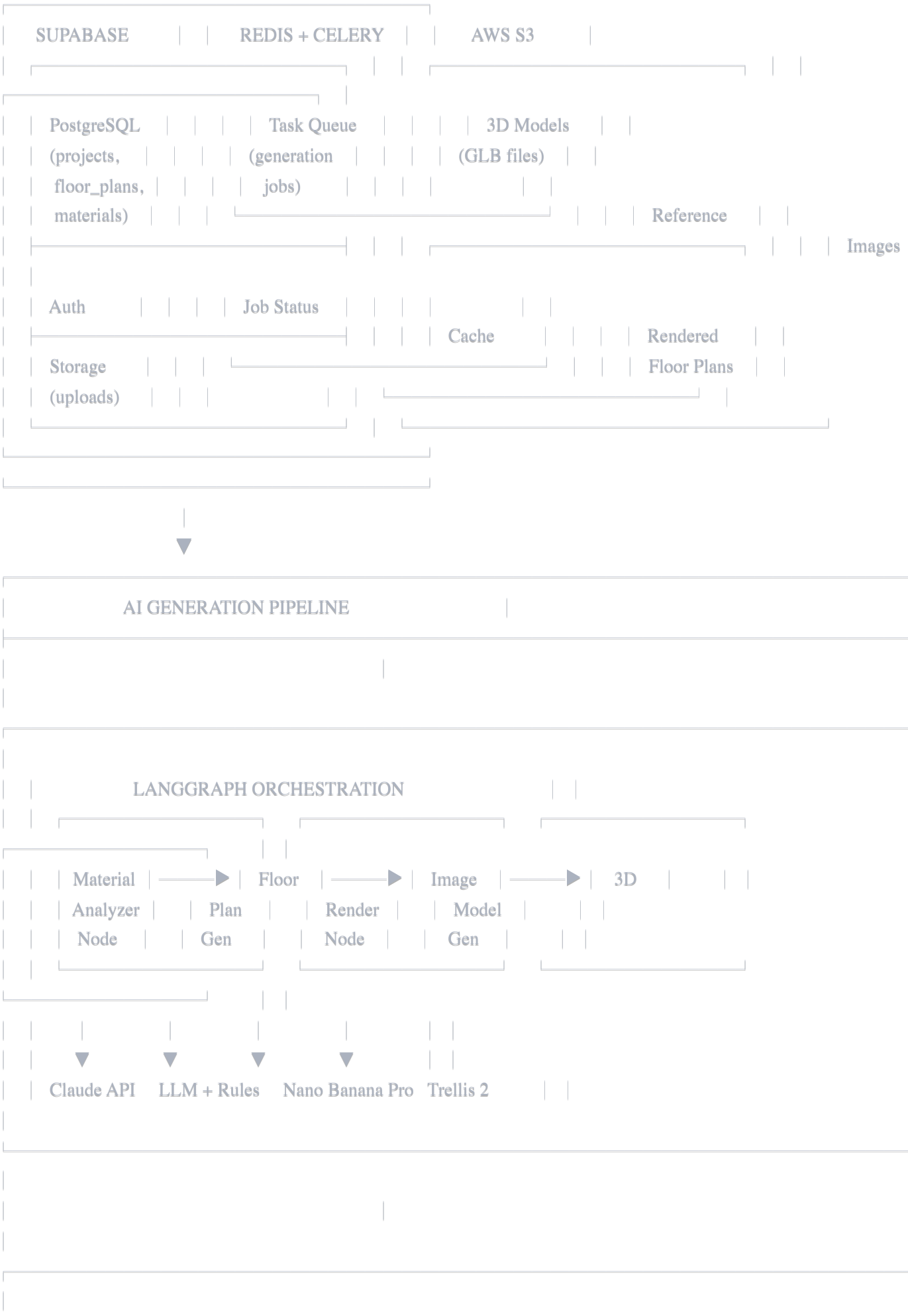
Recommendation: Use **Trellis 2** for the massing models with these parameters:

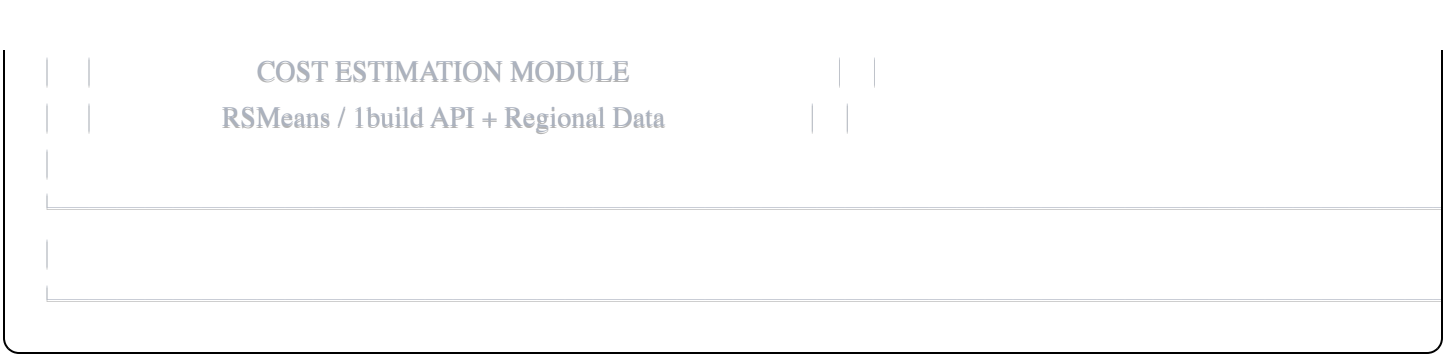
- `ss_guidance_strength`: 8.5 (stricter geometric adherence for buildings)
- `texture_size`: 1024 (balance quality/speed)
- `resolution`: 1024p

---

4. System Architecture




SUPABASE
REDIS + CELERY
AWS S3
PostgreSQL (projects, floor_plans, materials)
Task Queue (generation jobs)
3D Models (GLB files)
Reference Images
Auth
Job Status
Storage (uploads)
Cache
Rendered Floor Plans
AI GENERATION PIPELINE
LANGGRAPH ORCHESTRATION
Material Analyzer Node
Floor Plan Gen
Image Render Node
3D Model Gen
Claude API
LLM + Rules
Nano Banana Pro
Trellis 2



---

5. Agent Architecture: LangGraph Deterministic Workflow

5.1 Why Deterministic Workflow (Not Autonomous Agents)

For your use case, **deterministic workflows are the right choice** for these reasons:

1. **Predictable outputs**: Floor plans must meet specific constraints (dimensions, room counts)
2. **Auditability**: Each step can be inspected and debugged
3. **Cost control**: No runaway agent loops consuming API tokens
4. **Reproducibility**: Same inputs should produce consistent output quality
5. **Error handling**: Explicit retry/fallback logic at each node

5.2 LangGraph Workflow Definition

python

```python
from langgraph.graph import StateGraph, END
from typing import TypedDict, List, Optional

class ProjectState(TypedDict):
    # Inputs
    budget_min: float
    budget_max: float
    plot_width: float
    plot_length: float
    target_sqft: float
    bedrooms: int
    style_keywords: List[str]
    location: dict
    reference_images: List[str]

    # Intermediate state
    material_palette: Optional[dict]
    floor_plans: Optional[List[dict]]
    rendered_images: Optional[List[str]]

    # Outputs
    models_3d: Optional[List[str]]
    cost_estimates: Optional[List[dict]]

    # Control
    current_step: str
    errors: List[str]

# Define the workflow graph
workflow = StateGraph(ProjectState)

# Add nodes
workflow.add_node("analyze_materials", analyze_materials_node)
workflow.add_node("generate_floor_plans", generate_floor_plans_node)
workflow.add_node("render_floor_plans", render_floor_plans_node)
workflow.add_node("generate_3d_models", generate_3d_models_node)
workflow.add_node("estimate_costs", estimate_costs_node)
workflow.add_node("validate_outputs", validate_outputs_node)

# Define edges (deterministic flow)
workflow.set_entry_point("analyze_materials")
workflow.add_edge("analyze_materials", "generate_floor_plans")
workflow.add_edge("generate_floor_plans", "render_floor_plans")
```

```
workflow.add_edge("render_floor_plans", "generate_3d_models")
workflow.add_edge("generate_3d_models", "estimate_costs")
workflow.add_edge("estimate_costs", "validate_outputs")
workflow.add_edge("validate_outputs", END)

app = workflow.compile()
```

5.3 Node Implementations

Node 1: Material Analyzer

python

```
async def analyze_materials_node(state: ProjectState) -> ProjectState:
    """
    Extract color palettes, textures, and material types from reference images.
    Uses Claude's vision capabilities.
    """
    prompt = """Analyze these reference images and extract:
    1. Primary color palette (hex codes)
    2. Material types (wood species, tile patterns, metal finishes)
    3. Texture characteristics (matte/glossy, rough/smooth)
    4. Overall style classification

    Return as structured JSON."""

    response = await claude_client.analyze_images(
        images=state["reference_images"],
        prompt=prompt
    )

    return {**state, "material_palette": response}
```

Node 2: Floor Plan Generator

This is the most complex node. Use a hybrid approach:

python

```python
async def generate_floor_plans_node(state: ProjectState) -> ProjectState:
    """
    Generate 4 floor plan variants using LLM + constraint solver.

    Approach:
    1. LLM generates room adjacency graph and rough proportions
    2. Constraint solver ensures dimensional accuracy
    3. LLM refines circulation and daylight optimization
    """

    # Step 1: Generate room program
    room_program_prompt = f"""
    Design a residential floor plan with these constraints:
    - Plot: {state['plot_width']}' x {state['plot_length']}'
    - Target sqft: {state['target_sqft']}
    - Bedrooms: {state['bedrooms']}
    - Style: {', '.join(state['style_keywords'])}
    - Climate zone: {get_climate_zone(state['location'])}

    Output a room program with:
    - Room names and target sizes
    - Adjacency requirements (kitchen near dining, etc.)
    - Orientation preferences (bedrooms away from street, etc.)

    Generate 4 DISTINCT variants:
    1. Open-concept maximizing living space
    2. Traditional layout with defined rooms
    3. Split-bedroom for privacy
    4. Single-story vs two-story option
    """

    room_programs = await llm_generate(room_program_prompt)

    # Step 2: Convert to vectorized floor plans
    floor_plans = []
    for program in room_programs:
        plan = await vectorize_floor_plan(
            program=program,
            plot_dims=(state['plot_width'], state['plot_length']),
            target_sqft=state['target_sqft']
        )
        floor_plans.append(plan)
```

```
    return {**state, "floor_plans": floor_plans}
```

Node 3: Floor Plan Renderer

python

```
async def render_floor_plans_node(state: ProjectState) -> ProjectState:
    """
    Render vectorized floor plans to images using Nano Banana Pro.
    Generate:
    1. 2D colored floor plan view
    2. Isometric 3D massing view
    3. Multiple elevation views
    """
    rendered_images = []

    for plan in state["floor_plans"]:
        # Generate isometric rendering prompt
        prompt = f"""
        Architectural isometric rendering of a {plan['style']} house:
        - {plan['total_sqft']} square feet
        - {plan['stories']} stories
        - Materials: {state['material_palette']['exterior']}
        - Clean white background
        - Professional architectural visualization style
        - Show roof, walls, windows, and landscaping
        """

        # Call Nano Banana Pro via Fal AI
        result = await fal_client.run(
            "fal-ai/nano-banana-pro",
            input={
                "prompt": prompt,
                "aspect_ratio": "1:1",
                "output_format": "png"
            }
        )

        rendered_images.append(result["images"][0]["url"])

    return {**state, "rendered_images": rendered_images}
```

Node 4: 3D Model Generator

python

```python
async def generate_3d_models_node(state: ProjectState) -> ProjectState:
    """
    Convert rendered images to 3D models using Trellis 2.
    """
    models_3d = []

    for image_url in state["rendered_images"]:
        result = await fal_client.run(
            "fal-ai/trellis-2",
            input={
                "image_url": image_url,
                "ss_guidance_strength": 8.5,  # Higher for architecture
                "slat_guidance_strength": 3.5,
                "texture_size": 1024
            }
        )

        # Upload GLB to S3
        model_url = await upload_to_s3(
            result["model_glb"]["url"],
            f"models/{uuid4()}.glb"
        )
        models_3d.append(model_url)

    return {**state, "models_3d": models_3d}
```

Node 5: Cost Estimator

python

```python
async def estimate_costs_node(state: ProjectState) -> ProjectState:
    """
    Calculate cost bands for each floor plan variant.
    Uses regional construction cost data.
    """
    cost_estimates = []

    # Get regional cost multiplier
    regional_data = await get_regional_costs(state["location"])
    base_cost_per_sqft = regional_data["residential_avg"]  # ~$150-400/sqft

    for plan in state["floor_plans"]:
        # Calculate base cost
        base_cost = plan["total_sqft"] * base_cost_per_sqft

        # Apply modifiers
        modifiers = {
            "complexity": plan.get("complexity_factor", 1.0),
            "materials": state["material_palette"].get("cost_tier", 1.0),
            "stories": 1.15 if plan["stories"] > 1 else 1.0,
            "regional": regional_data["multiplier"]
        }

        total_modifier = 1.0
        for mod in modifiers.values():
            total_modifier *= mod

        estimated_cost = base_cost * total_modifier

        cost_estimates.append({
            "plan_id": plan["id"],
            "cost_low": int(estimated_cost * 0.85),
            "cost_high": int(estimated_cost * 1.15),
            "cost_breakdown": {
                "foundation": int(estimated_cost * 0.12),
                "framing": int(estimated_cost * 0.18),
                "exterior": int(estimated_cost * 0.15),
                "interior": int(estimated_cost * 0.25),
                "mechanical": int(estimated_cost * 0.15),
                "finishes": int(estimated_cost * 0.15)
            },
            "justification": generate_cost_justification(plan, modifiers)
        })
```

```
    return {**state, "cost_estimates": cost_estimates}
```

---

6. Database Schema

sql

```sql
-- Users (managed by Supabase Auth)
-- Supabase creates auth.users automatically

-- Projects
CREATE TABLE projects (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES auth.users(id) ON DELETE CASCADE,
    name TEXT NOT NULL,
    status TEXT DEFAULT 'draft', -- draft, generating, complete, archived

    -- Input parameters
    budget_min NUMERIC,
    budget_max NUMERIC,
    plot_width NUMERIC,
    plot_length NUMERIC,
    target_sqft NUMERIC,
    bedrooms INTEGER,
    style_keywords TEXT[],
    location JSONB, -- {lat, lng, zip, climate_zone}

    -- Metadata
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW(),
    share_token TEXT UNIQUE -- For shareable links
);

-- Reference Images
CREATE TABLE reference_images (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    project_id UUID REFERENCES projects(id) ON DELETE CASCADE,
    category TEXT NOT NULL, -- flooring, roofing, facade, kitchen
    storage_path TEXT NOT NULL,
    extracted_palette JSONB, -- AI-extracted colors and materials
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Floor Plans
CREATE TABLE floor_plans (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    project_id UUID REFERENCES projects(id) ON DELETE CASCADE,
    variant_number INTEGER NOT NULL, -- 1-4

    -- Plan data
```

```
    rooms JSONB NOT NULL, -- Array of room definitions
    total_sqft NUMERIC,
    stories INTEGER,

    -- Generated assets
    vector_data JSONB, -- SVG paths for floor plan
    rendered_image_url TEXT,
    model_3d_url TEXT,

    -- Cost estimation
    cost_estimate JSONB,

    -- Metadata
    selected BOOLEAN DEFAULT FALSE, -- User's chosen option
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Material Library
CREATE TABLE material_library (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    project_id UUID REFERENCES projects(id) ON DELETE CASCADE,

    material_type TEXT NOT NULL, -- flooring, wall, ceiling, exterior, roof
    name TEXT NOT NULL,
    color_hex TEXT,
    texture_url TEXT,
    properties JSONB, -- {finish, durability, cost_tier}

    source_image_id UUID REFERENCES reference_images(id)
);

-- Edit History (for click-to-edit feature)
CREATE TABLE model_edits (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    floor_plan_id UUID REFERENCES floor_plans(id) ON DELETE CASCADE,

    edit_type TEXT NOT NULL, -- retexture, structural
    target_element TEXT, -- What was clicked
    edit_data JSONB, -- Depends on edit_type

    before_state JSONB,
    after_state JSONB,

    created_at TIMESTAMPTZ DEFAULT NOW()
```

```
);

-- Indexes
CREATE INDEX idx_projects_user ON projects(user_id);
CREATE INDEX idx_floor_plans_project ON floor_plans(project_id);
CREATE INDEX idx_reference_images_project ON reference_images(project_id);

-- Enable Row Level Security
ALTER TABLE projects ENABLE ROW LEVEL SECURITY;
ALTER TABLE reference_images ENABLE ROW LEVEL SECURITY;
ALTER TABLE floor_plans ENABLE ROW LEVEL SECURITY;
ALTER TABLE material_library ENABLE ROW LEVEL SECURITY;
ALTER TABLE model_edits ENABLE ROW LEVEL SECURITY;

-- RLS Policies (user can only access their own data)
CREATE POLICY "Users can access own projects" ON projects
    FOR ALL USING (auth.uid() = user_id);

CREATE POLICY "Users can access own reference images" ON reference_images
    FOR ALL USING (project_id IN (SELECT id FROM projects WHERE user_id = auth.uid()));

-- Similar policies for other tables...
```

---

7. 3D Viewer Implementation (React Three Fiber)

7.1 Core Viewer Component

tsx

```tsx
// components/ModelViewer.tsx
import { Canvas } from '@react-three/fiber'
import { OrbitControls, Environment, useGLTF, Html } from '@react-three/drei'
import { Suspense, useState } from 'react'

interface ModelViewerProps {
  modelUrl: string
  onElementClick?: (elementId: string, position: [number, number, number]) => void
}

export function ModelViewer({ modelUrl, onElementClick }: ModelViewerProps) {
  const [selectedElement, setSelectedElement] = useState<string | null>(null)

  return (
    <Canvas
      camera={{ position: [5, 5, 5], fov: 50 }}
      style={{ width: '100%', height: '100%' }}
    >
      <Suspense fallback={<LoadingIndicator />}>
        <ambientLight intensity={0.5} />
        <directionalLight position={[10, 10, 5]} intensity={1} />

        <HouseModel
          url={modelUrl}
          onElementClick={(id, pos) => {
            setSelectedElement(id)
            onElementClick?.(id, pos)
          }}
          selectedElement={selectedElement}
        />

        <Environment preset="city" />
        <OrbitControls
          enablePan={true}
          enableZoom={true}
          enableRotate={true}
          minDistance={2}
          maxDistance={20}
        />
      </Suspense>
    </Canvas>
  )
}
```

```
function HouseModel({ url, onElementClick, selectedElement }) {
  const { scene } = useGLTF(url)

  // Make elements clickable
  scene.traverse((child) => {
    if (child.isMesh) {
      child.userData.clickable = true
    }
  })

  return (
    <primitive
      object={scene}
      onClick={(e) => {
        e.stopPropagation()
        const clickedMesh = e.object
        if (clickedMesh.userData.clickable) {
          onElementClick(clickedMesh.name, e.point.toArray())
        }
      }}
    />
  )
}

function LoadingIndicator() {
  return (
    <Html center>
      <div className="flex items-center gap-2">
        <div className="animate-spin h-6 w-6 border-2 border-blue-500 rounded-full border-t-transparent" />
        <span>Loading model...</span>
      </div>
    </Html>
  )
}
```

7.2 Click-to-Edit Panel

tsx

```tsx
// components/EditPanel.tsx
import { useState } from 'react'

interface EditPanelProps {
  elementId: string
  position: [number, number, number]
  onRetexture: (file: File) => Promise<void>
  onStructuralEdit: (instruction: string) => Promise<void>
  onClose: () => void
}

export function EditPanel({
  elementId,
  position,
  onRetexture,
  onStructuralEdit,
  onClose
}: EditPanelProps) {
  const [mode, setMode] = useState<'retexture' | 'structural'>('retexture')
  const [instruction, setInstruction] = useState('')
  const [loading, setLoading] = useState(false)

  const handleRetexture = async (file: File) => {
    setLoading(true)
    try {
      await onRetexture(file)
    } finally {
      setLoading(false)
    }
  }

  const handleStructuralEdit = async () => {
    setLoading(true)
    try {
      await onStructuralEdit(instruction)
    } finally {
      setLoading(false)
    }
  }

  return (
    <div className="absolute right-4 top-4 w-80 bg-white rounded-lg shadow-xl p-4">
      <div className="flex justify-between items-center mb-4">
```

```
  <h3 className="font-semibold">Edit: {elementId}</h3>
  <button onClick={onClose} className="text-gray-500 hover:text-gray-700">
    ×
  </button>
</div>

<div className="flex gap-2 mb-4">
  <button
    onClick={() => setMode('retexture')}
    className={`flex-1 py-2 rounded ${
      mode === 'retexture' ? 'bg-blue-500 text-white' : 'bg-gray-100'
    }`}
  >
    Re-texture
  </button>
  <button
    onClick={() => setMode('structural')}
    className={`flex-1 py-2 rounded ${
      mode === 'structural' ? 'bg-blue-500 text-white' : 'bg-gray-100'
    }`}
  >
    Structural Edit
  </button>
</div>

{mode === 'retexture' ? (
  <div className="space-y-4">
    <p className="text-sm text-gray-600">
      Upload a new texture image to apply to this surface
    </p>
    <input
      type="file"
      accept="image/*"
      onChange={(e) => e.target.files?.[0] && handleRetexture(e.target.files[0])}
      className="w-full"
    />
  </div>
) : (
  <div className="space-y-4">
    <p className="text-sm text-gray-600">
      Describe the change (e.g., "make this room 2 feet wider")
    </p>
    <textarea
      value={instruction}
```

```
          onChange={(e) => setInstruction(e.target.value)}
          placeholder="Enter your edit instruction..."
          className="w-full h-24 p-2 border rounded"
        />
        <button
          onClick={handleStructuralEdit}
          disabled={loading || !instruction}
          className="w-full py-2 bg-blue-500 text-white rounded disabled:opacity-50"
        >
          {loading ? 'Processing...' : 'Apply Edit'}
        </button>
      </div>
    )}
  </div>
 )
}
```

---

8. API Endpoints

8.1 FastAPI Router Structure

python

```python
# api/main.py
from fastapi import FastAPI, Depends, HTTPException, BackgroundTasks
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI(title="HomeVision API")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000", "https://yourdomain.com"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Routers
app.include_router(projects_router, prefix="/api/projects", tags=["projects"])
app.include_router(generations_router, prefix="/api/generations", tags=["generations"])
app.include_router(materials_router, prefix="/api/materials", tags=["materials"])
app.include_router(exports_router, prefix="/api/exports", tags=["exports"])
app.include_router(share_router, prefix="/api/share", tags=["share"])
```

8.2 Key Endpoints

python

```python
# api/routers/projects.py

@router.post("/", response_model=ProjectResponse)
async def create_project(
    project: ProjectCreate,
    user: User = Depends(get_current_user),
    db: AsyncSession = Depends(get_db)
):
    """Create a new project with input parameters."""
    ...

@router.post("/{project_id}/generate", response_model=GenerationJobResponse)
async def start_generation(
    project_id: UUID,
    background_tasks: BackgroundTasks,
    user: User = Depends(get_current_user)
):
    """
    Start the AI generation pipeline.
    Returns a job ID for polling status.
    """
    job_id = str(uuid4())

    # Queue the generation task
    background_tasks.add_task(
        run_generation_pipeline,
        project_id=project_id,
        job_id=job_id
    )

    return {"job_id": job_id, "status": "queued"}

@router.get("/{project_id}/status")
async def get_generation_status(
    project_id: UUID,
    user: User = Depends(get_current_user)
):
    """Poll generation progress."""
    status = await redis_client.get(f"job:{project_id}")
    return json.loads(status) if status else {"status": "unknown"}

# api/routers/share.py
```

```python
@router.post("/{project_id}/share")
async def create_share_link(
    project_id: UUID,
    user: User = Depends(get_current_user),
    db: AsyncSession = Depends(get_db)
):
    """Generate a shareable public link for architect handoff."""
    share_token = secrets.token_urlsafe(16)

    await db.execute(
        update(Project)
        .where(Project.id == project_id)
        .values(share_token=share_token)
    )
    await db.commit()

    return {
        "share_url": f"https://yourdomain.com/view/{share_token}",
        "expires_at": None  # Or set expiration
    }

@router.get("/view/{share_token}")
async def get_shared_project(
    share_token: str,
    db: AsyncSession = Depends(get_db)
):
    """
    Public endpoint - no auth required.
    Returns read-only project data for architect viewing.
    """
    project = await db.execute(
        select(Project).where(Project.share_token == share_token)
    )
    if not project:
        raise HTTPException(404, "Project not found")

    return ProjectPublicView.from_orm(project)
```

9. Implementation Phases

Phase 1: Foundation (Week 1-2)

- [ ] Set up Next.js project with Tailwind and shadcn/ui
- [ ] Configure Supabase (database, auth, storage)
- [ ] Build project creation wizard UI
- [ ] Implement image upload flow
- [ ] Set up FastAPI backend skeleton

Phase 2: AI Pipeline (Week 3-4)

- [ ] Implement LangGraph workflow structure
- [ ] Build material analyzer node (Claude vision)
- [ ] Build floor plan generator node
- [ ] Integrate Nano Banana Pro for rendering
- [ ] Integrate Trellis 2 for 3D generation
- [ ] Set up Celery + Redis for background jobs

Phase 3: 3D Viewer (Week 5-6)

- [ ] Build React Three Fiber viewer component
- [ ] Implement GLB model loading
- [ ] Add orbit controls and lighting
- [ ] Build click-to-select functionality
- [ ] Create edit panel UI

Phase 4: Cost & Polish (Week 7-8)

- [ ] Implement cost estimation module
- [ ] Build shareable link feature
- [ ] Add generation progress tracking
- [ ] Create gallery view for 4 options
- [ ] UI/UX polish and testing

Phase 5: Click-to-Edit (Week 9-10)

- [ ] Implement retexturing pipeline
- [ ] Build structural edit flow (regeneration)
- [ ] Add edit history tracking
- [ ] Test edge cases

---

10. Missing Components You Should Add

Based on my research, here are critical components you didn't mention:

10.1 Floor Plan Generation Engine

The hardest part of your system. Options:

1. **Custom LLM + SVG approach** (Recommended for MVP)
    - Use Claude/GPT-4 to generate room layouts as JSON
    - Convert to SVG for 2D visualization
    - Use as input for 3D massing generation
2. **Integrate with Maket.ai API** (If available)
    - Existing floor plan generation SaaS
    - May offer API access
3. **Fine-tune diffusion model** (Complex, future phase)
    - Train on RPLAN or similar dataset
    - ChatHouseDiffusion / HouseLLM approach

10.2 Climate Reasoning Module

For location-based constraints:

python

```
CLIMATE_CONSIDERATIONS = {
    "hot_humid": {
        "roof_pitch": "low",
        "window_orientation": "north_south",
        "considerations": ["cross_ventilation", "shade_structures"]
    },
    "cold": {
        "roof_pitch": "steep",
        "window_orientation": "south_facing",
        "considerations": ["thermal_mass", "vestibule_entry"]
    },
    # ... more climate zones
}
```

10.3 Authentication & Middleware

```python
# middleware/auth.py
from supabase import create_client
from fastapi import Depends, HTTPException
from fastapi.security import HTTPBearer

security = HTTPBearer()

async def get_current_user(token: str = Depends(security)):
    supabase = create_client(SUPABASE_URL, SUPABASE_KEY)

    try:
        user = supabase.auth.get_user(token.credentials)
        return user
    except Exception:
        raise HTTPException(401, "Invalid authentication")
```

10.4 WebSocket for Real-time Progress

```python
# For generation progress updates
from fastapi import WebSocket

@app.websocket("/ws/generation/{job_id}")
async def generation_progress(websocket: WebSocket, job_id: str):
    await websocket.accept()

    pubsub = redis_client.pubsub()
    await pubsub.subscribe(f"job_progress:{job_id}")

    async for message in pubsub.listen():
        if message["type"] == "message":
            await websocket.send_json(json.loads(message["data"]))
```

---

11. Cost Estimation Sources

For construction cost data:

1. **1build API** - Real-time pricing, 68M+ data points, $0.01-0.05/query
2. **RSMeans Data** - Industry standard, subscription-based
3. **Simplified approach for MVP**:

```python
REGIONAL_COST_PER_SQFT = {
    "CA": {"low": 200, "mid": 350, "high": 500},
    "TX": {"low": 150, "mid": 250, "high": 400},
    "NY": {"low": 250, "mid": 400, "high": 600},
    # ... regional data
}
```

---

12. Deployment Recommendation

For 5 users, keep it simple:

Component	Deployment
Frontend	Vercel (free tier)
Backend	Railway or Render ($5-20/mo)
Database	Supabase (free tier)
Redis	Upstash (free tier)
Storage	Supabase Storage or S3
Domain	Cloudflare ($0)

Estimated Monthly Cost: $5-25/month + AI API usage (~$50-100/project)

---

13. Summary of Recommendations

Decision Point	Recommendation	Reasoning
Database	Supabase	Auth + Storage + Postgres bundle, perfect for MVP
Image Generation	Nano Banana Pro	Best semantic understanding, handles architectural prompts well

Decision Point	Recommendation	Reasoning
Image-to-3D	Trellis 2	Best for hard-surface/architecture, good price-quality ratio
Agent Framework	LangGraph	Deterministic workflow, debuggable, matches your use case
3D Viewer	React Three Fiber	Declarative, React-native, rich ecosystem
Floor Plan Gen	Custom LLM + Rules	Most control, can iterate on prompt engineering
Shareable Output	Public URL with token	Easiest to implement, architect-friendly

Document generated for HomeVision architectural visualization platform Last updated: January 2025