Tutorial 21: Multimodal and Image Processing - Visual AI Agents

Difficulty: advanced
Reading Time: 2 hours
Tags: advanced, multimodal, images, vision, visual-ai
Description: Build agents that process images, documents, and multimodal content using Gemini's vision capabilities for visual analysis and understanding.

:::tip WORKING IMPLEMENTATION AVAILABLE

A complete, tested implementation of this tutorial is available!

- 📂 View Implementation (./../../tutorial_implementation/tutorial21)
- ✔️ **70 tests passing** (63% coverage)
- 🎨 **5 tools** including synthetic image generation ⭐
- 📚 User-friendly Makefile with comprehensive help system
- 🚀 4 automation scripts (download, analyze, generate, demo)
- 📖 Complete README synchronized with implementation

See the implementation directory for:
- **Complete vision catalog agent** with 5 specialized tools
- **Synthetic image generation** using Gemini 2.5 Flash Image ⭐ NEW
- **Multi-agent workflow** (vision analyzer + catalog generator)
- **Automation scripts** for batch processing and generation
- **Image loading utilities** and optimization
- **Comprehensive test suite** (70 tests)
- **Interactive demos** and sample images
- **User-friendly Makefile** with help system

Quick Start:

```
cd tutorial_implementation/tutorial21
make                    # Show all available commands
make setup              # Install dependencies
make download-images    # Get sample images
make generate           # Generate synthetic mockups ⭐
make dev                # Start interactive agent
```

:::

Tutorial 21: Multimodal & Image Generation

Goal: Master multimodal capabilities including image input/output, vision understanding, and image generation using Gemini models and Vertex AI Imagen.

Prerequisites:

- Tutorial 01 (Hello World Agent)
- Tutorial 02 (Function Tools)
- Tutorial 19 (Artifacts & File Management)
- Understanding of image formats and MIME types

What You'll Learn:

- Processing images with Gemini vision models
- Using `types.Part` for multimodal content
- **Synthetic image generation with Gemini 2.5 Flash Image** ⭐ NEW
- Handling `inline_data` vs `file_data`
- Building vision-based agents with 5 specialized tools
- Working with multiple image inputs
- Creating automation scripts for batch processing
- User-friendly Makefile with help system
- Best practices for multimodal applications

Time to Complete: 50-65 minutes

Why Multimodal Matters

Problem: Many real-world applications require understanding and generating images, not just text.

Solution: **Multimodal models** process text and images together, enabling vision-based applications and image generation.

Benefits:

- [MEM] **Vision Understanding**: Analyze images, extract information
- 🎨 **Image Generation**: Create images from text descriptions
- [FLOW] **Multimodal Reasoning**: Combine visual and textual context
- 📊 **Visual Analytics**: Charts, graphs, diagrams analysis
- 🏷️ **Object Detection**: Identify objects in images
- 📝 **OCR**: Extract text from images

Use Cases:

- Product catalog analysis
- Document understanding (invoices, receipts)
- Medical image analysis
- Visual inspection and quality control
- Content moderation
- Creative content generation

---

1. Multimodal Input Basics

Understanding types.Part

`types.Part` is the fundamental unit for multimodal content in ADK.

Source: `google.genai.types`

```
from google.genai import types

# Text part
text_part = types.Part.from_text("Describe this image")

# Image part (inline data)
image_part = types.Part(
    inline_data=types.Blob(
        data=image_bytes,           # Raw image bytes
        mime_type='image/png'       # MIME type
    )
)

# Image part (file reference)
image_part = types.Part(
    file_data=types.FileData(
        file_uri='gs://bucket/image.jpg',  # Cloud Storage URI
        mime_type='image/jpeg'
    )
)
```

Supported Image Formats

- **PNG**: `image/png`
- **JPEG**: `image/jpeg`
- **WEBP**: `image/webp`
- **HEIC**: `image/heic`
- **HEIF**: `image/heif`

Loading Images

```python
import base64
from google.genai import types

# Load from file
def load_image_from_file(path: str) -> types.Part:
    """Load image from local file."""

    with open(path, 'rb') as f:
        image_bytes = f.read()

    # Determine MIME type from extension
    if path.endswith('.png'):
        mime_type = 'image/png'
    elif path.endswith('.jpg') or path.endswith('.jpeg'):
        mime_type = 'image/jpeg'
    elif path.endswith('.webp'):
        mime_type = 'image/webp'
    else:
        raise ValueError(f"Unsupported image format: {path}")

    return types.Part(
        inline_data=types.Blob(
            data=image_bytes,
            mime_type=mime_type
        )
    )

# Load from URL
import requests

def load_image_from_url(url: str) -> types.Part:
    """Load image from URL."""

    response = requests.get(url)
    response.raise_for_status()

    image_bytes = response.content

    # Determine MIME type from Content-Type header
    mime_type = response.headers.get('Content-Type', 'image/jpeg')

    return types.Part(
        inline_data=types.Blob(
            data=image_bytes,
            mime_type=mime_type
        )
```

```
    )

# Load from Cloud Storage
def load_image_from_gcs(uri: str) -> types.Part:
    """Load image from Google Cloud Storage."""

    # For GCS, use file_data instead of inline_data
    mime_type = 'image/jpeg'  # Determine from file extension

    return types.Part(
        file_data=types.FileData(
            file_uri=uri,
            mime_type=mime_type
        )
    )
```

2. Vision Understanding

Basic Image Analysis

```
"""
Basic vision understanding with Gemini.
"""

import asyncio
import os
from google.adk.agents import Agent, Runner
from google.genai import types

# Environment setup
os.environ['GOOGLE_GENAI_USE_VERTEXAI'] = '1'
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project-id'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'

async def analyze_image():
    """Analyze an image with Gemini vision."""

    # Create vision agent
    agent = Agent(
        model='gemini-2.0-flash',  # Supports vision
        name='vision_analyst',
        instruction="""
You are a vision analysis expert. When given images, you:

1. Describe what you see in detail
2. Identify key objects and their relationships
3. Analyze composition and visual elements
4. Extract any visible text (OCR)
5. Provide insights and observations

Be specific and thorough in your analysis.
        """.strip()
    )

    # Load image
    image_part = load_image_from_file('product.jpg')

    # Create multimodal query
    query_parts = [
        types.Part.from_text("Analyze this product image in detail. What is it
        image_part
    ]

    # Run analysis
    runner = Runner()
    result = await runner.run_async(
```

```
        query_parts,
        agent=agent
    )

    print("VISION ANALYSIS:")
    print(result.content.parts[0].text)

if __name__ == '__main__':
    asyncio.run(analyze_image())
```

Multiple Image Analysis

```
async def compare_images():
    """Compare multiple images."""

    agent = Agent(
        model='gemini-2.0-flash',
        name='image_comparator',
        instruction="""
Compare the provided images and identify:

1. Similarities and differences
2. Common elements
3. Unique features of each
4. Overall assessment

Provide a structured comparison.
        """.strip()
    )

    # Load multiple images
    image1 = load_image_from_file('product_v1.jpg')
    image2 = load_image_from_file('product_v2.jpg')

    # Create query with multiple images
    query_parts = [
        types.Part.from_text("Compare these two product versions:"),
        types.Part.from_text("Version 1:"),
        image1,
        types.Part.from_text("Version 2:"),
        image2,
        types.Part.from_text("What are the key differences?")
    ]

    runner = Runner()
    result = await runner.run_async(query_parts, agent=agent)

    print("COMPARISON RESULT:")
    print(result.content.parts[0].text)
```

3. Real-World Example: Visual Product Catalog Analyzer

Let's build a complete product catalog analysis system with vision capabilities.

Complete Implementation

```
"""
Visual Product Catalog Analyzer
Analyzes product images, extracts information, and generates descriptions.
"""

import asyncio
import os
from typing import List, Dict
from google.adk.agents import Agent, Runner, Session
from google.adk.tools import FunctionTool
from google.adk.tools.tool_context import ToolContext
from google.genai import types

# Environment setup
os.environ['GOOGLE_GENAI_USE_VERTEXAI'] = '1'
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project-id'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'

class ProductCatalogAnalyzer:
    """Analyze product images and create catalog entries."""

    def __init__(self):
        """Initialize product catalog analyzer."""

        self.catalog: List[Dict] = []

        # Vision analysis agent
        self.vision_agent = Agent(
            model='gemini-2.0-flash',
            name='vision_analyzer',
            instruction="""
You are a product vision analyst. When analyzing product images:

1. Identify the product type and category
2. Describe key visual features (color, size, material, design)
3. Note any visible text (brand names, labels, specs)
4. Assess product condition and quality
5. Identify unique selling points

Provide structured, detailed analysis.
            """.strip(),
            generate_content_config=types.GenerateContentConfig(
                temperature=0.3,
                max_output_tokens=1024
            )
        )
```

```
        # Description generator agent
        async def generate_catalog_entry(
            product_name: str,
            analysis: str,
            tool_context: ToolContext
        ) -> str:
            """Generate marketing-ready catalog entry."""

            entry = f"""
# {product_name}

## Description

{analysis}

## Key Features

- High-quality construction
- Modern design
- Versatile use cases

## Specifications

- Material: [Extracted from analysis]
- Dimensions: [Extracted from analysis]
- Color: [Extracted from analysis]

*Analysis generated from product image*
            """.strip()

            # Save as artifact
            part = types.Part.from_text(entry)
            version = await tool_context.save_artifact(
                filename=f"{product_name}_catalog_entry.md",
                part=part
            )

            return f"Catalog entry created (version {version})"

        self.catalog_agent = Agent(
            model='gemini-2.0-flash',
            name='catalog_generator',
            instruction="""
You are a product catalog content creator. Generate professional,
marketing-ready product descriptions based on visual analysis.
```

```
Focus on:
- Compelling product descriptions
- Key features and benefits
- Technical specifications when available
- Customer-friendly language

Use the generate_catalog_entry tool to save entries.
            """.strip(),
            tools=[FunctionTool(generate_catalog_entry)],
            generate_content_config=types.GenerateContentConfig(
                temperature=0.6,
                max_output_tokens=1536
            )
        )

        self.runner = Runner()

    async def analyze_product(self, product_id: str, image_path: str):
        """
        Analyze product image and create catalog entry.

        Args:
            product_id: Unique product identifier
            image_path: Path to product image
        """

        print(f"\n{'='*70}")
        print(f"ANALYZING PRODUCT: {product_id}")
        print(f"IMAGE: {image_path}")
        print(f"{'='*70}\n")

        # Step 1: Visual analysis
        print("📸 Step 1: Visual Analysis...")

        image_part = load_image_from_file(image_path)

        analysis_query = [
            types.Part.from_text(f"Analyze this product image for {product_id}
            image_part
        ]

        analysis_result = await self.runner.run_async(
            analysis_query,
            agent=self.vision_agent
        )

        analysis_text = analysis_result.content.parts[0].text
```

```
        print(f"\n🔍 VISUAL ANALYSIS:\n{analysis_text}\n")

        # Step 2: Generate catalog entry
        print("📝 Step 2: Generating Catalog Entry...")

        catalog_query = f"""
Based on this visual analysis, create a professional catalog entry for {produc

{analysis_text}

Use the generate_catalog_entry tool to save the entry.
        """.strip()

        catalog_result = await self.runner.run_async(
            catalog_query,
            agent=self.catalog_agent
        )

        print(f"\n✔️ RESULT:\n{catalog_result.content.parts[0].text}\n")

        # Store in catalog
        self.catalog.append({
            'product_id': product_id,
            'image_path': image_path,
            'analysis': analysis_text,
            'timestamp': 'timestamp_here'
        })

        print(f"{'='*70}\n")

    async def batch_analyze(self, products: List[tuple[str, str]]):
        """
        Analyze multiple products.

        Args:
            products: List of (product_id, image_path) tuples
        """

        for product_id, image_path in products:
            await self.analyze_product(product_id, image_path)
            await asyncio.sleep(2)

    def get_catalog_summary(self) -> str:
        """Get summary of analyzed products."""

        summary = f"\nPRODUCT CATALOG SUMMARY\n{'='*70}\n"
```

```
        summary += f"Total Products Analyzed: {len(self.catalog)}\n\n"

        for i, product in enumerate(self.catalog, 1):
            summary += f"{i}. {product['product_id']}\n"
            summary += f"   Image: {product['image_path']}\n"
            summary += f"   Analysis: {product['analysis'][:100]}...\n\n"

        summary += f"{'='*70}\n"

        return summary

def load_image_from_file(path: str) -> types.Part:
    """Load image from file."""

    with open(path, 'rb') as f:
        image_bytes = f.read()

    if path.endswith('.png'):
        mime_type = 'image/png'
    elif path.endswith('.jpg') or path.endswith('.jpeg'):
        mime_type = 'image/jpeg'
    elif path.endswith('.webp'):
        mime_type = 'image/webp'
    else:
        mime_type = 'image/jpeg'

    return types.Part(
        inline_data=types.Blob(
            data=image_bytes,
            mime_type=mime_type
        )
    )

async def main():
    """Main entry point."""

    analyzer = ProductCatalogAnalyzer()

    # Analyze multiple products
    # Note: Replace with actual image paths
    products = [
        ('PROD-001', 'images/laptop.jpg'),
        ('PROD-002', 'images/headphones.jpg'),
        ('PROD-003', 'images/smartwatch.jpg')
    ]

    # For demo, create placeholder images
```

```
    import io
    from PIL import Image

    os.makedirs('images', exist_ok=True)

    for product_id, image_path in products:
        # Create placeholder image
        img = Image.new('RGB', (400, 400), color=(73, 109, 137))
        img.save(image_path)

    # Batch analyze
    await analyzer.batch_analyze(products)

    # Show summary
    print(analyzer.get_catalog_summary())

if __name__ == '__main__':
    asyncio.run(main())
```

Expected Output

```
================================================================================
ANALYZING PRODUCT: PROD-001
IMAGE: images/laptop.jpg
================================================================================

📸 Step 1: Visual Analysis...

🔍 VISUAL ANALYSIS:
This is a laptop computer with a modern, sleek design. Key observations:

**Product Type**: Laptop/Notebook computer

**Visual Features**:
- Color: Dark gray or space gray metallic finish
- Design: Thin profile with minimal bezels
- Screen: Approximately 13-15 inch display
- Build Quality: Premium aluminum unibody construction
- Keyboard: Full-size backlit keyboard visible
- Trackpad: Large, integrated trackpad

**Branding**: [Brand logo visible on lid]

**Condition**: Appears new, pristine condition

**Unique Features**:
- Ultra-portable design
- Modern port configuration (USB-C)
- High-resolution display
- Professional aesthetic

**Target Market**: Business professionals, students, creative professionals

📝 Step 2: Generating Catalog Entry...

✔️ RESULT:
Catalog entry created (version 1)

I've created a comprehensive catalog entry for PROD-001 that highlights its
premium build quality, modern design, and professional features. The entry
emphasizes its portability and versatility for various user needs.

================================================================================


================================================================================
ANALYZING PRODUCT: PROD-002
IMAGE: images/headphones.jpg
```

```
================================================================================

📸 Step 1: Visual Analysis...

🔍 VISUAL ANALYSIS:
These are over-ear wireless headphones with premium features.

**Product Type**: Over-ear wireless headphones

**Visual Features**:
- Color: Matte black finish
- Design: Closed-back, circumaural design
- Ear Cups: Large, cushioned ear pads
- Headband: Adjustable with soft padding
- Build: Combination of metal and high-quality plastic
- Controls: Physical buttons visible on ear cup

**Technical Indicators**:
- Wireless capability (no visible cable)
- Likely active noise cancellation (based on design)
- Folding mechanism for portability

**Condition**: New, retail-ready

**Key Features**:
- Premium comfort design
- Professional audio quality
- Portable with carrying case
- Modern aesthetic

**Target Market**: Audiophiles, commuters, content creators

📝 Step 2: Generating Catalog Entry...

✔️ RESULT:
Catalog entry created (version 1)

Professional catalog entry generated for PROD-002, emphasizing audio quality,
comfort, and wireless convenience. Targeted at users seeking premium audio
experience.

================================================================================


================================================================================
ANALYZING PRODUCT: PROD-003
IMAGE: images/smartwatch.jpg
================================================================================
```

```
📸 Step 1: Visual Analysis...

🔍 VISUAL ANALYSIS:
This is a smartwatch with fitness and health tracking capabilities.

**Product Type**: Smartwatch / Fitness Tracker

**Visual Features**:
- Display: Circular AMOLED touchscreen
- Case: Stainless steel or aluminum
- Color: Black with matching band
- Band: Silicone sport band, appears comfortable
- Interface: Digital crown visible
- Design: Modern, minimalist aesthetic

**Technical Features Visible**:
- Heart rate sensor on back
- Water-resistant design
- Multiple button/crown controls
- Likely GPS enabled (based on form factor)

**Condition**: New condition

**Key Features**:
- Health and fitness tracking
- Always-on display (likely)
- Interchangeable bands
- Smart notifications
- Modern design suitable for any occasion

**Target Market**: Fitness enthusiasts, health-conscious users, tech adopters

📝 Step 2: Generating Catalog Entry...

✔️ RESULT:
Catalog entry created (version 1)

Catalog entry created for PROD-003, highlighting health tracking features,
modern design, and versatility for both fitness and everyday wear.

================================================================================

PRODUCT CATALOG SUMMARY
================================================================================
Total Products Analyzed: 3
```

```
1. PROD-001
   Image: images/laptop.jpg
   Analysis: This is a laptop computer with a modern, sleek design. Key observ

**Product Type**: Laptop...

2. PROD-002
   Image: images/headphones.jpg
   Analysis: These are over-ear wireless headphones with premium features.

**Product Type**: Over-ear wireless ...

3. PROD-003
   Image: images/smartwatch.jpg
   Analysis: This is a smartwatch with fitness and health tracking capabilitie

**Product Type**: Smartwatch / ...

================================================================================
```

4. Synthetic Image Generation with Gemini 2.5 Flash Image ⭐ NEW

Overview

Gemini 2.5 Flash Image is a text-to-image model that generates photorealistic product images from text descriptions. Perfect for:

- 🎨 **Rapid Prototyping**: Test catalog designs before photography
- 💡 **Concept Visualization**: Show clients product concepts
- 🔄 **Variations**: Generate multiple style/color variations quickly
- 📐 **Layout Testing**: Create mockups for different aspect ratios
- 💰 **Cost Savings**: No studio equipment or photographers needed

Basic Synthetic Generation

```
"""
Generate synthetic product images using Gemini 2.5 Flash Image.
"""

import os
from google import genai
from google.genai import types as genai_types
from PIL import Image
from io import BytesIO

async def generate_product_mockup(
    product_description: str,
    style: str = "photorealistic product photography",
    aspect_ratio: str = "1:1"
) -> str:
    """
    Generate synthetic product image.

    Args:
        product_description: Detailed product description
        style: Photography style (photorealistic, studio, lifestyle)
        aspect_ratio: Image aspect ratio (1:1, 16:9, 4:3, 3:2, etc.)

    Returns:
        Path to generated image
    """

    # Create detailed professional prompt
    detailed_prompt = f"""
A {style} of {product_description}.

The image should be:
- High-resolution and professional quality
- Well-lit with studio lighting
- Sharp focus on the product
- Clean composition
- Suitable for e-commerce or marketing materials
    """.strip()

    # Initialize client
    client = genai.Client(api_key=os.environ.get('GOOGLE_API_KEY'))

    # Generate image
    response = client.models.generate_content(
        model='gemini-2.5-flash-image',
        contents=[detailed_prompt],
```

```
        config=genai_types.GenerateContentConfig(
            response_modalities=['Image'],
            image_config=genai_types.ImageConfig(
                aspect_ratio=aspect_ratio
            )
        )
    )

    # Extract and save image
    for part in response.candidates[0].content.parts:
        if part.inline_data:
            image = Image.open(BytesIO(part.inline_data.data))
            image_path = f"generated_{product_description[:20]}.jpg"
            image.save(image_path, 'JPEG', quality=95)

            return image_path

    raise ValueError("No image generated")

# Example usage
async def demo_synthetic_generation():
    """Demo synthetic image generation."""

    # Generate desk lamp mockup
    lamp_path = await generate_product_mockup(
        product_description="minimalist desk lamp with brushed aluminum finish
        style="photorealistic product photography",
        aspect_ratio="1:1"
    )

    print(f"Generated lamp mockup: {lamp_path}")

    # Generate leather wallet mockup
    wallet_path = await generate_product_mockup(
        product_description="premium leather wallet with gold stitching and ca
        style="photorealistic product photography on marble surface",
        aspect_ratio="4:3"
    )

    print(f"Generated wallet mockup: {wallet_path}")

    # Generate gaming mouse mockup
    mouse_path = await generate_product_mockup(
        product_description="wireless gaming mouse with RGB lighting and ergon
        style="photorealistic product photography with dramatic lighting",
        aspect_ratio="16:9"
    )
```

```
    print(f"Generated mouse mockup: {mouse_path}")

if __name__ == '__main__':
    import asyncio
    asyncio.run(demo_synthetic_generation())
```

Supported Aspect Ratios

Gemini 2.5 Flash Image supports various aspect ratios:

- **1:1** (1024x1024) - Perfect for social media, product catalogs
- **16:9** (1344x768) - Wide product shots, lifestyle photography
- **4:3** (1184x864) - Standard product photos
- **3:2** (1248x832) - Professional photography format
- **9:16** (768x1344) - Vertical/mobile-first layouts

Style Options

Customize the photography style in your prompt:

- **Photorealistic product photography** (default)
- **Studio lighting with white background**
- **Lifestyle/contextual photography**
- **Artistic/creative product shots**
- **Minimalist composition**
- **Dramatic lighting**

Integration with Vision Analysis

Combine synthetic generation with vision analysis:

```
async def generate_and_analyze_product():
    """Generate synthetic image and analyze it."""

    # Step 1: Generate synthetic mockup
    image_path = await generate_product_mockup(
        product_description="modern wireless earbuds with charging case",
        aspect_ratio="1:1"
    )

    # Step 2: Load generated image
    image_part = load_image_from_file(image_path)

    # Step 3: Analyze with vision model
    vision_agent = Agent(
        model='gemini-2.0-flash-exp',
        name='vision_analyzer'
    )

    runner = Runner()
    analysis = await runner.run_async(
        [
            types.Part.from_text("Analyze this product mockup and create a cat
            image_part
        ],
        agent=vision_agent
    )

    print(f"Generated image: {image_path}")
    print(f"Analysis: {analysis.content.parts[0].text}")
```

Use Cases

E-commerce Prototyping:

```
# Generate product variations
for color in ['black', 'white', 'silver']:
    await generate_product_mockup(
        f"smartphone in {color} color, modern design",
        aspect_ratio="1:1"
    )
```

Marketing Materials:

```
# Create lifestyle shots
await generate_product_mockup(
    "coffee mug on wooden desk with morning sunlight",
    style="lifestyle photography with warm tones",
    aspect_ratio="16:9"
)
```

Concept Testing:

```
# Test different designs
for design in ['minimalist', 'luxury', 'sporty']:
    await generate_product_mockup(
        f"water bottle, {design} design aesthetic",
        aspect_ratio="3:2"
    )
```

5. Image Generation with Vertex AI Imagen (Alternative)

Basic Image Generation

```
"""
Generate images using Vertex AI Imagen (alternative to Gemini 2.5 Flash Image)
"""

from google.cloud import aiplatform
from vertexai.preview.vision_models import ImageGenerationModel

def generate_image(prompt: str, output_path: str):
    """
    Generate image from text prompt.

    Args:
        prompt: Text description of desired image
        output_path: Where to save generated image
    """

    # Initialize Vertex AI
    aiplatform.init(
        project='your-project-id',
        location='us-central1'
    )

    # Load Imagen model
    model = ImageGenerationModel.from_pretrained('imagen-3.0-generate-001')

    # Generate images
    response = model.generate_images(
        prompt=prompt,
        number_of_images=1,
        aspect_ratio='1:1',  # Options: 1:1, 9:16, 16:9, 4:3, 3:4
        safety_filter_level='block_some',  # Options: block_most, block_some,
        person_generation='allow_all'  # Options: allow_all, allow_adult, bloc
    )

    # Save first generated image
    image = response.images[0]
    image.save(output_path)

    print(f"Image saved to: {output_path}")
```

Image Generation Agent

```
async def create_image_generation_agent():
    """Agent that generates images based on requests."""

    def generate_product_image(description: str, style: str = 'photorealistic'
        """Generate product image from description."""

        # Construct prompt
        prompt = f"{description}, {style} style, professional product photogra
        prompt += "high quality, detailed, studio lighting, white background"

        # Generate image
        output_path = f"generated_{hash(description) % 10000}.png"
        generate_image(prompt, output_path)

        return f"Image generated: {output_path}"

    agent = Agent(
        model='gemini-2.0-flash',
        name='image_generator',
        instruction="""
You help generate product images based on descriptions.

When asked to create an image:
1. Clarify the requirements
2. Use generate_product_image tool with detailed description
3. Specify style (photorealistic, illustration, etc.)

Always provide helpful descriptions for best results.
        """.strip(),
        tools=[FunctionTool(generate_product_image)]
    )

    return agent
```

6. Best Practices

✔️ DO: Optimize Image Sizes

```
from PIL import Image
import io

def optimize_image(image_bytes: bytes, max_size_kb: int = 500) -> bytes:
    """Optimize image size for API calls."""

    image = Image.open(io.BytesIO(image_bytes))

    # Resize if too large
    max_dimension = 1024
    if max(image.size) > max_dimension:
        image.thumbnail((max_dimension, max_dimension), Image.LANCZOS)

    # Save with compression
    output = io.BytesIO()
    image.save(output, format='JPEG', quality=85, optimize=True)

    return output.getvalue()

# Use optimized images
original_bytes = open('large_image.jpg', 'rb').read()
optimized_bytes = optimize_image(original_bytes)

image_part = types.Part(
    inline_data=types.Blob(
        data=optimized_bytes,
        mime_type='image/jpeg'
    )
)
```

✔️ DO: Handle Multiple Image Formats

```
def get_mime_type(file_path: str) -> str:
    """Determine MIME type from file extension."""

    extension = file_path.lower().split('.')[-1]

    mime_types = {
        'png': 'image/png',
        'jpg': 'image/jpeg',
        'jpeg': 'image/jpeg',
        'webp': 'image/webp',
        'heic': 'image/heic',
        'heif': 'image/heif'
    }

    return mime_types.get(extension, 'image/jpeg')
```

✔️ DO: Provide Clear Image Context

```
# ✔️ Good - Clear context
query = [
    types.Part.from_text("This is an image of our new product launch:"),
    image_part,
    types.Part.from_text("Analyze the product design and identify key features
]

# ❌ Bad - Ambiguous
query = [image_part, types.Part.from_text("What is this?")]
```

Summary

You've mastered multimodal capabilities and synthetic image generation:

Key Takeaways:

- ✔️ `types.Part` for multimodal content (text + images)
- ✔️ `inline_data` for embedded images, `file_data` for references

- ✔️ Gemini 2.0 Flash supports vision understanding
- ✔️ **Gemini 2.5 Flash Image for synthetic generation** ⭐ NEW
- ✔️ Vertex AI Imagen for alternative image generation
- ✔️ Multiple image analysis for comparisons
- ✔️ Vision-based product catalog applications (5 tools)
- ✔️ Automation scripts for batch processing
- ✔️ User-friendly Makefile with help system
- ✔️ Image optimization for API efficiency

Implementation Highlights:

- 🎨 **5 Specialized Tools**: list, generate, upload, analyze, compare
- 📸 **4 Automation Scripts**: download, analyze, generate, demo
- 🧪 **70 Tests** with 63% coverage
- 📚 **User-Friendly Makefile**: Comprehensive help system
- ⭐ **Synthetic Generation**: Gemini 2.5 Flash Image integration

Production Checklist:

- [ ] Image optimization implemented (size, format)
- [ ] Error handling for invalid images
- [ ] MIME type validation
- [ ] Vision model tested on representative images
- [ ] **Synthetic generation tested with various prompts** ⭐
- [ ] Generated images reviewed for quality
- [ ] Cost monitoring for image operations
- [ ] Image storage strategy defined
- [ ] Compliance with image generation policies
- [ ] **Makefile help system documented**
- [ ] **Automation scripts for batch operations**

Next Steps:

- **Tutorial 22**: Master Model Selection & Optimization
- **Tutorial 23**: Learn Production Deployment
- **Tutorial 24**: Explore Advanced Observability

Resources:

- Gemini Vision Documentation (https://cloud.google.com/vertex-ai/docs/generative-ai/multimodal/overview)
- Imagen Documentation (https://cloud.google.com/vertex-ai/docs/generative-ai/image/overview)
- Multimodal Best Practices (https://cloud.google.com/vertex-ai/docs/generative-ai/multimodal/best-practices)
- Working Implementation (./../../tutorial_implementation/tutorial21) - Complete, tested code

---

🎉 **Tutorial 21 Complete!** You now know how to work with images in ADK. Continue to Tutorial 22 to learn about model selection and optimization.

:::info Implementation Available

Check out the complete implementation (./../../tutorial_implementation/tutorial21) with:

- **Vision catalog agent** with 5 specialized tools
- **70 passing tests** (63% coverage)
- **Synthetic image generation** using Gemini 2.5 Flash Image ⭐
- **4 automation scripts** (download, analyze, generate, demo)
- **User-friendly Makefile** with comprehensive help system
- **Image processing utilities** and optimization
- **Multi-agent workflow** (vision + catalog)
- **Interactive demos** and sample images
- **Comprehensive documentation**

Quick Start:

```
cd tutorial_implementation/tutorial21
make                # Show all commands
make setup          # Install dependencies
make generate       # Generate synthetic mockups ⭐
make dev            # Start interactive agent
```

:::

Source: Google ADK Training Hub