# Gemini Robotics ER 1.5 - Comprehensive Documentation

## Table of Contents

1. [Introduction](#introduction)
2. [Model Overview](#model-overview)
3. [Key Capabilities](#key-capabilities)
4. [Setup and Prerequisites](#setup-and-prerequisites)
5. [Core Components](#core-components)
6. [Code Examples - Detailed Walkthrough](#code-examples---detailed-walkthrough)
   * [Example 1: Pointing to Undefined Objects](#example-1-pointing-to-undefined-objects)
   * [Example 2: Pointing to Defined Objects](#example-2-pointing-to-defined-objects)
   * [Example 3: Abstract Object Detection](#example-3-abstract-object-detection)
   * [Example 4: Point to All Instances](#example-4-point-to-all-instances)
   * [Example 5: Pointing to Object Parts (Serial)](#X13c2239a5f5dd0f584bbb701f4004b2f4287b9c)
   * [Example 6: Pointing to Object Parts (Parallel)](#Xf2b031c59cc40eff76fdead47d05c3c5de17871)
   * [Example 7: Counting by Pointing](#example-7-counting-by-pointing)
   * [Example 8: Pointing in GIF/Video Frames](#example-8-pointing-in-gifvideo-frames)
   * [Example 9: 2D Bounding Boxes](#example-9-2d-bounding-boxes)
   * [Example 10: Simple Trajectory Planning](#example-10-simple-trajectory-planning)
   * [Example 11: Path for Brushing Particles](#example-11-path-for-brushing-particles)
   * [Example 12: Obstacle-Avoidance Trajectory](#example-12-obstacle-avoidance-trajectory)
   * [Example 13: Item to Remove to Make Room](#example-13-item-to-remove-to-make-room)
   * [Example 14: Orchestrating - Packing a Lunch](#X6159c07a81ae5dab8c0386bd2e1c44214f6d6d7)
   * [Example 15: Empty Electrical Sockets](#example-15-empty-electrical-sockets)
   * [Example 16: Physical Constraints (Weight Limit)](#X9a01df1532f7a8b25fbac26ac1b17767d9cfe3b)
   * [Example 17: Video Analysis with Timestamps](#X0c0f2fd01138603f8bb89574b7ba529f1c4fbd9)
   * [Example 18: Video Analysis - Time Range Zoom](#Xe88dae5b618e6ad98e9b1e3512ce52d94b8c19b)
   * [Example 19: Finding the Fourth Row of Shelves](#Xe77bb305f18308ac3989573799b344e5a1308eb)
   * [Example 20: Finding Shelves with Specific Items](#X47ae3755aad2b47aee9e5a7bd97b7dcaf348fd9)
   * [Example 21: Counting Items with Thinking](#example-21-counting-items-with-thinking)
   * [Example 22: Multi-View Success Detection](#example-22-multi-view-success-detection)
   * [Example 23: Image Enhancement with Code Execution](#X04f6fb008865297be87ae71d29aa8b72e1f79dd)
   * [Example 24: Segmentation Masks](#example-24-segmentation-masks)
   * [Example 25: Robot Function Call Generation](#X5cc8abd680a26388da00a22f973f81b8b1108ed)
7. [Advanced Features](#advanced-features)
8. [Best Practices](#best-practices)
9. [Troubleshooting](#troubleshooting)

## Introduction

The **Gemini Robotics ER 1.5 Preview** model represents a significant advancement in vision-language AI for robotics applications. This model brings Google’s Gemini agentic capabilities to the physical world, enabling robots to understand, reason about, and interact with their environment through natural language and visual perception.

### What is Gemini Robotics ER?

Gemini Robotics-ER (Embodied Reasoning) is a specialized vision-language model designed specifically for robotics and physical AI applications. It combines advanced computer vision, spatial reasoning, and natural language understanding to enable intelligent robotic systems that can:

* **Perceive**: Detect and identify objects, understand spatial relationships, and track movements
* **Reason**: Plan trajectories, evaluate task completion, and make decisions based on physical constraints
* **Act**: Generate executable robot commands and coordinate multi-step tasks

### Research Background

Based on Google’s research in embodied AI and multimodal learning, Gemini Robotics ER builds upon:

1. **Vision-Language Models (VLMs)**: Advanced neural networks trained on massive datasets of images and text to understand visual-linguistic relationships
2. **Spatial Reasoning**: Specialized training for understanding 3D space, object relationships, and physical constraints
3. **Agentic Capabilities**: Integration with Gemini’s reasoning and planning abilities for autonomous task execution
4. **Grounded Interaction**: Direct connection between visual perception and actionable robot commands

### Model Status and Availability

* **Version**: Gemini Robotics ER 1.5 Preview
* **Availability**: Preview release for developers and researchers
* **Access**: Through Google AI API with API key authentication
* **Pricing**: Subject to Google AI pricing tiers

## Model Overview

### Technical Specifications

|  |  |
| --- | --- |
| Specification | Details |
| **Model Name** | gemini-robotics-er-1.5-preview |
| **Input Types** | Text, Images (PNG, JPEG, GIF), Video (MP4, MOV), Audio |
| **Output Type** | Text (including JSON, code, structured data) |
| **Input Token Limit** | 1,048,576 tokens (~1M tokens) |
| **Output Token Limit** | 65,536 tokens (~65K tokens) |
| **Context Window** | Large context for complex multi-modal inputs |

### Supported Features

✅ **Code Execution** - Generate and execute Python code for image processing and analysis  
✅ **Function Calling** - Generate structured robot function calls  
✅ **Search Grounding** - Enhanced with search capabilities (when enabled)  
✅ **Structured Outputs** - JSON formatted responses for programmatic use  
✅ **Thinking Mode** - Enhanced reasoning with configurable thinking budget  
✅ **Multi-Modal Input** - Process text, images, video, and audio simultaneously

## Key Capabilities

### 1. Object Detection and Localization

The model excels at identifying and localizing objects in images and video:

* **Point Detection**: Identify exact pixel coordinates of objects
* **Bounding Boxes**: Draw 2D and 3D boxes around objects
* **Segmentation Masks**: Generate precise pixel-level object masks
* **Multi-Object Tracking**: Track objects across video frames

**Use Cases**: - Pick-and-place operations - Quality inspection - Inventory management - Scene understanding

### 2. Spatial Reasoning

Advanced understanding of 3D space and object relationships:

* **Spatial Relationships**: Understand “on top of”, “next to”, “inside”, etc.
* **Distance Estimation**: Approximate distances between objects
* **Occlusion Handling**: Reason about partially hidden objects
* **Multi-View Understanding**: Combine information from multiple camera angles

**Use Cases**: - Navigation planning - Collision avoidance - Object assembly - Scene reconstruction

### 3. Task Planning and Trajectory Generation

Generate executable plans for robotic tasks:

* **Path Planning**: Create collision-free trajectories
* **Sequential Actions**: Break down complex tasks into steps
* **Obstacle Avoidance**: Navigate around obstacles
* **Grasp Planning**: Determine optimal grasp points and angles

**Use Cases**: - Autonomous navigation - Manipulation tasks - Task orchestration - Motion planning

### 4. Physical Reasoning

Understand physical properties and constraints:

* **Weight Estimation**: Reason about object weight and payload limits
* **Stability Analysis**: Evaluate if configurations are stable
* **Material Properties**: Understand fragility, hardness, etc.
* **Affordances**: Recognize what actions are possible with objects

**Use Cases**: - Safe manipulation - Constraint-aware planning - Tool use - Material handling

### 5. Multi-Modal Understanding

Process and combine information from multiple sources:

* **Image + Text**: Answer questions about images
* **Video Analysis**: Understand temporal sequences
* **Multi-Camera Fusion**: Combine views from multiple cameras
* **Contextual Reasoning**: Use conversation history for context

**Use Cases**: - Human-robot interaction - Task monitoring - Quality assurance - Teleoperation support

## Setup and Prerequisites

### 1. Installation

First, install the required packages:

!pip install -q -U google-generativeai

### 2. API Key Configuration

Obtain a Google AI API key from [Google AI Studio](https://aistudio.google.com/app/apikey) and configure it:

from google.colab import userdata  
GOOGLE\_API\_KEY = userdata.get("GOOGLE\_API\_KEY")

### 3. Import Required Libraries

import google.generativeai as genai  
from google.generativeai import types  
from PIL import Image, ImageDraw, ImageFont  
import json  
import time  
import textwrap  
import IPython.display as display  
from ipywidgets import widgets  
import concurrent.futures

### 4. Initialize the Client

MODEL\_ID = "gemini-robotics-er-1.5-preview"  
client = genai.Client(api\_key=GOOGLE\_API\_KEY)  
  
# Test the connection  
response = client.models.generate\_content(  
 model=MODEL\_ID,   
 contents="Are you there?"  
)  
print(response.text)

## Core Components

### 1. Coordinate System

The model uses a **normalized coordinate system** for all spatial outputs:

* **Range**: 0-1000 for both x and y coordinates
* **Format**: [y, x] - Note that y comes first
* **Origin**: Top-left corner (0, 0)
* **Conversion**: Multiply by image dimensions and divide by 1000 to get pixel coordinates

def normalize\_coordinates(point, img\_width, img\_height):  
 """Convert normalized coordinates to pixel coordinates"""  
 y\_norm, x\_norm = point  
 x\_pixel = int((x\_norm / 1000) \* img\_width)  
 y\_pixel = int((y\_norm / 1000) \* img\_height)  
 return (x\_pixel, y\_pixel)

### 2. JSON Response Formats

The model returns structured JSON for different tasks:

**Point Detection**:

[  
 {"point": [y, x], "label": "object\_name"},  
 {"point": [y, x], "label": "another\_object"}  
]

**Bounding Boxes**:

[  
 {"box\_2d": [ymin, xmin, ymax, xmax], "label": "object\_name"}  
]

**Segmentation Masks**:

[  
 {  
 "box\_2d": [ymin, xmin, ymax, xmax],  
 "label": "object\_name",  
 "mask": "data:image/png;base64,<base64\_encoded\_png>"  
 }  
]

### 3. Helper Functions

The notebook includes several utility functions for visualization:

* generate\_point\_html(): Interactive point visualization with JavaScript
* plot\_bounding\_boxes(): Draw 2D bounding boxes on images
* overlay\_points\_on\_frames(): Annotate video frames with points
* plot\_segmentation\_masks(): Overlay segmentation masks on images
* parse\_json(): Extract and parse JSON from model responses

## Code Examples - Detailed Walkthrough

### Example 1: Pointing to Undefined Objects

**Objective**: Detect and point to multiple objects in an image without specifying what objects to look for.

#### Procedure Description

This example demonstrates the model’s ability to autonomously identify interesting objects in a scene:

1. **Load Image**: Opens aloha-arms-table.png showing a robot setup with various objects on a table
2. **Prepare Prompt**: Requests the model to identify up to 10 items without specifying which items
3. **Model Processing**: The model analyzes the image and returns point coordinates with descriptive labels
4. **Visualization**: Uses JavaScript-based interactive HTML to display points overlaid on the image

#### Code Snippet

img = get\_image\_resized("aloha-arms-table.png")  
  
prompt = textwrap.dedent("""\  
 Point to no more than 10 items in the image. The label returned should be an  
 identifying name for the object detected.  
  
 The answer should follow the JSON format:  
 [{"point": <point>, "label": <label1>}, ...]  
  
 The points are in [y, x] format normalized to 0-1000.""")  
  
start\_time = time.time()  
json\_output = call\_gemini\_robotics\_er(img, prompt)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json\_output))

#### Output Description

The model returns a JSON array containing:

* **Structure**: List of dictionaries, each with point and label keys
* **Point Coordinates**: Normalized [y, x] coordinates (0-1000 range) indicating the center or key point of each object
* **Labels**: Descriptive names like “bread”, “banana”, “bowl”, “measuring cup”, “robot arm”, etc.
* **Processing Time**: Typically 2-5 seconds depending on image complexity
* **Visualization**: Interactive HTML overlay showing numbered points on the image with labels appearing on hover

**Example Output Format**:

[  
 {"point": [450, 320], "label": "bread"},  
 {"point": [380, 580], "label": "banana"},  
 {"point": [520, 410], "label": "blue bowl"},  
 {"point": [290, 650], "label": "robot gripper"}  
]

#### Use Case

* **Inventory Scanning**: Quickly identify all items in a workspace
* **Scene Understanding**: Autonomous exploration of unknown environments
* **Object Discovery**: Find all manipulatable objects without prior knowledge

### Example 2: Pointing to Defined Objects

**Objective**: Locate specific objects that are explicitly requested in the prompt.

#### Procedure Description

This example shows how to query for specific objects:

1. **Define Queries**: Specifies a list of target objects: “bread”, “starfruit”, “banana”
2. **Construct Prompt**: Embeds the query list into a natural language prompt
3. **Model Processing**: The model searches for each specified object
4. **Result Aggregation**: Collects all detected points into a single list
5. **Error Handling**: Handles cases where JSON parsing might fail

#### Code Snippet

img = get\_image\_resized("aloha-arms-table.png")  
  
queries = [  
 "bread",  
 "starfruit",  
 "banana",  
]  
  
prompt = textwrap.dedent(f"""\  
Get all points matching the following objects: {', '.join(queries)}. The label  
returned should be an identifying name for the object detected.  
  
The answer should follow the JSON format:  
[{{"point": <point>, "label": <label1>}}, ...]  
  
The points are in [y, x] format normalized to 0-1000.  
""")  
  
start\_time = time.time()  
json\_output = call\_gemini\_robotics\_er(img, prompt)  
  
points\_data = []  
try:  
 data = json.loads(json\_output)  
 points\_data.extend(data)  
except json.JSONDecodeError:  
 print("Warning: Invalid JSON response. Skipping.")  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json.dumps(points\_data)))

#### Output Description

The model returns:

* **Targeted Points**: Only points for the requested objects
* **Multiple Instances**: If multiple instances exist (e.g., two bananas), both are identified
* **Missing Objects**: If an object isn’t present, it’s omitted from results
* **Label Matching**: Labels correspond to the query terms or close variations
* **Processing Time**: Similar to Example 1, typically 2-5 seconds

**Example Output Format**:

[  
 {"point": [445, 325], "label": "bread"},  
 {"point": [375, 585], "label": "starfruit"},  
 {"point": [380, 520], "label": "banana"}  
]

#### Use Case

* **Targeted Pick-and-Place**: Find specific items for manipulation
* **Inventory Verification**: Check if required items are present
* **Task-Specific Detection**: Focus on objects relevant to current task

### Example 3: Abstract Object Detection

**Objective**: Detect objects based on abstract categories rather than specific names.

#### Procedure Description

This example demonstrates semantic understanding:

1. **Abstract Query**: Uses the category “fruit” instead of specific fruit names
2. **Semantic Reasoning**: The model understands that bananas, starfruit, limes are fruits
3. **Category-Based Detection**: Returns all objects matching the semantic category
4. **Automatic Labeling**: Each fruit is labeled with its specific name, not just “fruit”

#### Code Snippet

points\_data = []  
img = get\_image\_resized("aloha-arms-table.png")  
  
prompt = textwrap.dedent(f"""\  
 Get all points for fruit. The label returned should be an identifying  
 name for the object detected.  
  
 The answer should follow the json format:  
 [{{"point": <point>, "label": <label1>}}, ...]  
  
 The points are in [y, x] format normalized to 0-1000.""")  
  
start\_time = time.time()  
json\_output = call\_gemini\_robotics\_er(img, prompt)  
  
try:  
 data = json.loads(json\_output)  
 points\_data.extend(data)  
except json.JSONDecodeError:  
 print(f"Warning: Invalid JSON response, skipping.")  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json.dumps(points\_data)))

#### Output Description

The model returns:

* **Semantic Grouping**: All objects matching the category “fruit”
* **Specific Labels**: Each fruit labeled with its specific type (banana, lime, starfruit)
* **Exclusion of Non-Fruits**: Items like bread, bowls are not included
* **Category Understanding**: Demonstrates the model’s knowledge of object categories

**Example Output Format**:

[  
 {"point": [380, 580], "label": "banana"},  
 {"point": [375, 585], "label": "starfruit"},  
 {"point": [290, 450], "label": "lime"}  
]

#### Use Case

* **Category-Based Sorting**: Separate fruits from vegetables, tools from food items
* **Flexible Queries**: Allow high-level task descriptions like “find all tools”
* **Knowledge Transfer**: Apply learned categories to new objects

### Example 4: Point to All Instances

**Objective**: Detect all instances of specific object types in a more complex scene.

#### Procedure Description

This example uses a game board image to demonstrate multi-instance detection:

1. **Load Game Board**: Opens an image showing a tic-tac-toe style game
2. **Multiple Queries**: Searches for “game board slot” and “X game piece”
3. **Serial Processing**: Processes each query separately for clarity
4. **Result Aggregation**: Combines all detected points into a single visualization
5. **Error Handling**: Continues processing even if one query fails

#### Code Snippet

points\_data = []  
img = get\_image\_resized("gameboard.png")  
  
queries = [  
 "game board slot",  
 "X game piece",  
]  
  
start\_time = time.time()  
for obj in queries:  
 prompt = textwrap.dedent(f"""\  
 Get all points matching {obj}. The label returned should be an identifying  
 name for the object detected.  
  
 The answer should follow the JSON format:  
 [{{"point": <point>, "label": <label1>}}, ...]  
  
 The points are in [y, x] format normalized to 0-1000.""")  
 json\_output = call\_gemini\_robotics\_er(img, prompt)  
  
 try:  
 data = json.loads(json\_output)  
 points\_data.extend(data)  
 except json.JSONDecodeError:  
 print(f"Warning: Invalid JSON response for {obj}. Skipping.")  
 continue  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json.dumps(points\_data)))

#### Output Description

The model returns:

* **Multiple Instances**: All 9 game board slots are detected
* **Multiple Types**: Both empty slots and X pieces are identified
* **Positional Labels**: May include position indicators (e.g., “top-left slot”, “center X”)
* **Complete Coverage**: Every instance of the requested object types
* **Processing Time**: Cumulative time for all queries (typically 4-8 seconds)

**Example Output Format**:

[  
 {"point": [200, 150], "label": "game board slot 1"},  
 {"point": [200, 350], "label": "game board slot 2"},  
 {"point": [200, 550], "label": "game board slot 3"},  
 {"point": [400, 250], "label": "X game piece"},  
 {"point": [600, 450], "label": "X game piece"}  
]

#### Use Case

* **Game State Analysis**: Evaluate board game positions
* **Pattern Recognition**: Detect repeated structures in organized environments
* **Counting Applications**: Count instances of specific items

### Example 5: Pointing to Object Parts (Serial)

**Objective**: Identify specific parts of objects with precise localization.

#### Procedure Description

This example demonstrates fine-grained spatial understanding:

1. **Define Part Queries**: Specifies object-part pairs like (“banana”, “the stem”), (“measuring cup”, “handle”)
2. **Serial Processing**: Sends one query at a time to the model
3. **Part-Level Precision**: Locates specific features of objects, not just object centers
4. **Template Usage**: Uses a prompt template (POINT\_PROMPT\_TEMPLATE) for consistency
5. **Cumulative Results**: Aggregates all detected points across multiple queries

#### Code Snippet

img = get\_image\_resized("aloha-arms-table.png")  
points\_data = []  
  
queries = [  
 ("paper bag", "handles"),  
 ("banana", "the stem"),  
 ("banana", "center"),  
 ("starfruit", "center"),  
 ("lime", "center"),  
 ("light blue bowl", "rim"),  
 ("dark blue bowl", "rim"),  
 ("measuring cup", "rim"),  
 ("measuring cup", "handle"),  
 ("bowl", "tomato"),  
]  
  
start\_time = time.time()  
for obj, part in queries:  
 prompt = POINT\_PROMPT\_TEMPLATE.replace("$object", obj).replace("$part", part)  
  
 json\_output = call\_gemini\_robotics\_er(img, prompt)  
  
 try:  
 data = json.loads(json\_output)  
 points\_data.extend(data)  
 except json.JSONDecodeError:  
 print(f"Warning: Invalid JSON response for {obj}, {part}. Skipping.")  
 continue  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json.dumps(points\_data)))

#### Output Description

The model returns:

* **Part-Specific Points**: Precise coordinates for handles, rims, stems, centers
* **Spatial Accuracy**: Points are located on the actual part, not the object center
* **Descriptive Labels**: Labels include both object and part (e.g., “banana stem”)
* **Processing Time**: Cumulative time for all 10 queries (typically 15-25 seconds due to serial processing)
* **Visualization**: All points shown together, revealing the model’s understanding of object structure

**Example Output Format**:

[  
 {"point": [320, 280], "label": "paper bag handle"},  
 {"point": [365, 575], "label": "banana stem"},  
 {"point": [380, 585], "label": "banana center"},  
 {"point": [540, 425], "label": "measuring cup handle"}  
]

#### Use Case

* **Grasp Point Planning**: Identify optimal locations to grasp objects
* **Fine Manipulation**: Target specific features for precise operations
* **Quality Inspection**: Check specific parts of assembled products

### Example 6: Pointing to Object Parts (Parallel)

**Objective**: Same as Example 5 but using parallel processing for efficiency.

#### Procedure Description

This example optimizes the previous approach:

1. **Same Queries**: Uses identical object-part pairs as Example 5
2. **Parallel Execution**: Uses ThreadPoolExecutor to send multiple requests simultaneously
3. **Performance Optimization**: Reduces total processing time through concurrency
4. **Result Aggregation**: Collects results as they complete

#### Code Snippet

img = get\_image\_resized("aloha-arms-table.png")  
points\_data = []  
  
queries = [  
 ("paper bag", "handles"),  
 ("banana", "the stem"),  
 ("banana", "center"),  
 ("starfruit", "center"),  
 ("lime", "center"),  
 ("light blue bowl", "rim"),  
 ("dark blue bowl", "rim"),  
 ("measuring cup", "rim"),  
 ("measuring cup", "handle"),  
 ("bowl", "tomato"),  
]  
  
def process\_query(obj, part):  
 prompt = POINT\_PROMPT\_TEMPLATE.replace("$object", obj).replace("$part", part)  
 json\_output = call\_gemini\_robotics\_er(img, prompt)  
 try:  
 data = json.loads(json\_output)  
 return data  
 except json.JSONDecodeError:  
 print(f"Warning: Invalid JSON response for {obj}, {part}. Skipping.")  
 return []  
  
start\_time = time.time()  
with concurrent.futures.ThreadPoolExecutor() as executor:  
 results = executor.map(  
 lambda query: process\_query(query[0], query[1]), queries  
 )  
  
for result in results:  
 points\_data.extend(result)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json.dumps(points\_data)))

#### Output Description

The model returns:

* **Identical Results**: Same points and labels as the serial version
* **Significantly Reduced Time**: Processing time drops from 15-25s to 3-6s
* **Efficiency Gain**: ~4-5x speedup through parallelization
* **Same Accuracy**: No loss in detection quality

**Performance Comparison**: - Serial Processing: ~20 seconds - Parallel Processing: ~4-5 seconds - **Speedup**: ~4-5x faster

#### Use Case

* **Real-Time Applications**: When response time is critical
* **Batch Processing**: Processing multiple images or multiple queries per image
* **Production Systems**: Optimizing throughput in robotic systems

### Example 7: Counting by Pointing

**Objective**: Count objects by detecting and pointing to each instance.

#### Procedure Description

This example demonstrates object counting through detection:

1. **Load Image**: Opens an image of washers in a container
2. **Counting Prompt**: Requests the model to point to each individual washer
3. **Point Detection**: Model identifies each washer separately
4. **Count Extraction**: The number of points equals the count of washers
5. **Visual Verification**: Points overlay confirms each washer was detected

#### Code Snippet

img = get\_image\_resized("washer.png")  
  
prompt = textwrap.dedent("""\  
 Point to each washer in the box. Return the answer in the format:  
 [{"point": <point>, "label": <label1>}, ...]  
  
 The points are in [y, x] format normalized to 0-1000.""")  
  
start\_time = time.time()  
json\_output = call\_gemini\_robotics\_er(img, prompt)  
  
try:  
 data = json.loads(json\_output)  
 print(f"count: {len(data)}")  
except json.JSONDecodeError:  
 print("Error: Could not decode JSON response from the model.")  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json\_output))

#### Output Description

The model returns:

* **Individual Points**: One point per washer in the container
* **Accurate Count**: The length of the JSON array equals the total number of washers
* **Label Enumeration**: Each washer may be labeled with a number (1, 2, 3…) or description
* **Processing Time**: Typically 2-4 seconds
* **Visualization**: Numbered overlay showing each detected washer

**Example Output Format**:

[  
 {"point": [250, 180], "label": "washer 1"},  
 {"point": [280, 220], "label": "washer 2"},  
 {"point": [310, 195], "label": "washer 3"},  
 // ... continues for all washers  
 {"point": [520, 410], "label": "washer 15"}  
]

**Count**: 15 washers detected

#### Use Case

* **Inventory Management**: Count items in containers or shelves
* **Quality Control**: Verify piece counts in packaging
* **Parts Counting**: Automated counting for assembly lines

### Example 8: Pointing in GIF/Video Frames

**Objective**: Track objects across multiple frames in an animated GIF or video.

#### Procedure Description

This example demonstrates temporal tracking:

1. **Load GIF**: Opens aloha-pen.gif showing a robotic manipulation sequence
2. **Extract Frames**: Converts the GIF into individual PIL Image frames
3. **Frame Sampling**: Analyzes every 10th frame for efficiency
4. **Multi-Object Tracking**: Tracks multiple objects (“pen on desk”, “pen in robot hand”, “laptop opened”, “laptop closed”)
5. **Temporal Interpolation**: Fills in point data for unanalyzed frames using the most recent analyzed frame
6. **GIF Creation**: Overlays points on all frames and creates an annotated GIF

#### Code Snippet

gif\_path = "aloha-pen.gif"  
gif = Image.open(gif\_path)  
frames = extract\_frames(gif)  
  
queries = [  
 "pen (on desk)",  
 "pen (in robot hand)",  
 "laptop (opened)",  
 "laptop (closed)",  
]  
  
prompt = textwrap.dedent(f"""\  
Point to the following objects in the provided image: {", ".join(queries)}.  
  
The answer should follow the JSON format:  
[{{"point": <point>, "label": <label1>}}, ...]  
  
The points are in [y, x] format normalized to 0-1000.  
  
If no objects are found, return an empty JSON list [].""")  
  
# Analyze every 10th frame  
analyzed\_frames\_data = []  
frame\_step = 10  
  
for i in range(0, len(frames), frame\_step):  
 frame = frames[i]  
 print(f"Processing frame {i+1}/{len(frames)}...")  
  
 image\_response = client.models.generate\_content(  
 model=MODEL\_ID,  
 contents=[frame, prompt],  
 config=types.GenerateContentConfig(  
 temperature=0.5,  
 thinking\_config=types.ThinkingConfig(thinking\_budget=0),  
 ),  
 )  
  
 json\_output = parse\_json(image\_response.text)  
 frame\_points = json.loads(json\_output)  
 analyzed\_frames\_data.append(frame\_points)  
  
# Interpolate for all frames  
points\_data\_all\_frames = populate\_points\_for\_all\_frames(  
 len(frames), frame\_step, analyzed\_frames\_data  
)  
  
# Create annotated GIF  
modified\_frames = overlay\_points\_on\_frames(frames, points\_data\_all\_frames)  
display\_gif(modified\_frames)

#### Output Description

The model returns:

* **Per-Frame Detection**: JSON array for each analyzed frame containing detected objects
* **State Changes**: Tracks transitions (pen moving from desk to robot hand, laptop opening/closing)
* **Temporal Coherence**: Object labels change as the scene evolves
* **Empty Frames**: Returns [] for frames where tracked objects aren’t visible
* **Processing Time**: Depends on GIF length; for ~60 frames analyzing every 10th: ~15-20 seconds
* **Annotated GIF**: Final output shows tracking points overlaid on the original animation

**Example Output Per Frame**:

// Frame 1  
[  
 {"point": [450, 320], "label": "pen (on desk)"},  
 {"point": [280, 550], "label": "laptop (opened)"}  
]  
  
// Frame 30 (pen picked up)  
[  
 {"point": [380, 420], "label": "pen (in robot hand)"},  
 {"point": [280, 550], "label": "laptop (opened)"}  
]

#### Use Case

* **Action Monitoring**: Track objects during manipulation tasks
* **State Detection**: Identify when objects change state (open/closed, on/off)
* **Motion Analysis**: Analyze robot movements and object trajectories
* **Success Verification**: Confirm task completion across video sequences

### Example 9: 2D Bounding Boxes

**Objective**: Detect multiple objects and draw 2D bounding boxes around them.

#### Procedure Description

This example demonstrates object detection with bounding boxes:

1. **Load Image**: Opens the table scene with robot arms and objects
2. **Comprehensive Detection**: Requests up to 25 objects with unique identifiers
3. **Bounding Box Format**: Model returns [ymin, xmin, ymax, xmax] coordinates
4. **Unique Labeling**: Objects with multiple instances get descriptive labels (color, position, characteristics)
5. **Visualization**: Draws colored rectangles around each detected object with labels

#### Code Snippet

img = get\_image\_resized("aloha-arms-table.png")  
  
prompt = textwrap.dedent("""\  
 Return bounding boxes as a JSON array with labels. Never return masks or  
 code fencing. Limit to 25 objects. Include as many objects as you can  
 identify on the table.  
 If an object is present multiple times, name them according to their  
 unique characteristic (colors, size, position, unique characteristics,  
 etc..).  
 The format should be as follows:  
 [{"box\_2d": [ymin, xmin, ymax, xmax], "label": <label for the object>}]  
 normalized to 0-1000. The values in box\_2d must only be integers.  
""")  
  
start\_time = time.time()  
json\_output = call\_gemini\_robotics\_er(img, prompt)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
  
plot\_bounding\_boxes(img, json\_output)  
img

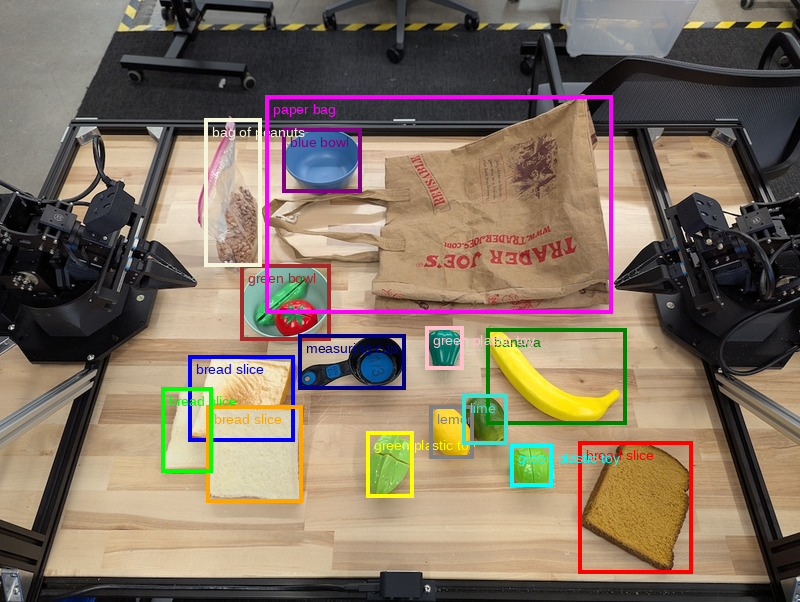
#### Output Description

The model returns:

* **Bounding Box Coordinates**: Four integers [ymin, xmin, ymax, xmax] defining the rectangle
* **Normalized Values**: All coordinates in 0-1000 range
* **Comprehensive Labels**: Descriptive names that distinguish similar objects
* **Multiple Objects**: Typically detects 15-25 objects in a complex scene
* **Processing Time**: 3-5 seconds
* **Visualization**: Image with colored rectangles and labels for each object

**Example Output Format**:

[  
 {"box\_2d": [430, 300, 480, 360], "label": "bread"},  
 {"box\_2d": [350, 560, 410, 620], "label": "banana"},  
 {"box\_2d": [480, 380, 560, 470], "label": "light blue bowl"},  
 {"box\_2d": [265, 620, 325, 710], "label": "right robot gripper"},  
 {"box\_2d": [510, 180, 580, 240], "label": "paper bag with handles"}  
]



2D Bounding Boxes Output

#### Use Case

* **Object Detection**: Identify all objects in a workspace
* **Region of Interest**: Define areas for manipulation or inspection
* **Scene Annotation**: Label objects for training data or documentation
* **Spatial Analysis**: Understand object sizes and positions relative to each other

### Example 10: Simple Trajectory Planning

**Objective**: Generate a trajectory of waypoints for moving an object from one location to another.

#### Procedure Description

This example demonstrates path planning capabilities:

1. **Load Scene**: Opens an image showing a desk with a red pen and an organizer
2. **Task Description**: Asks to move the red pen to the top of the organizer
3. **Waypoint Generation**: Requests 15 intermediate points forming a trajectory
4. **Sequential Labeling**: Points are labeled 0 (start) through n (end) showing movement order
5. **Visualization**: Displays the trajectory as a sequence of numbered points

#### Code Snippet

img = get\_image\_resized("aloha\_desk.png")  
points\_data = []  
  
prompt = textwrap.dedent("""\  
 Place a point on the red pen, then 15 points for the trajectory of moving  
 the red pen to the top of the organizer on the left.  
  
 The points should be labeled by order of the trajectory, from '0' (start  
 point at left hand) to <n> (final point).  
  
 The answer should follow the JSON format:  
 [{"point": <point>, "label": <label1>}, ...]  
  
 The points are in [y, x] format normalized to 0-1000.""")  
  
start\_time = time.time()  
config=types.GenerateContentConfig(temperature=0.5)  
json\_output = call\_gemini\_robotics\_er(img, prompt, config)  
  
try:  
 data = json.loads(json\_output)  
 points\_data.extend(data)  
except json.JSONDecodeError:  
 print("Warning: Invalid JSON response. Skipping.")  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json.dumps(points\_data)))

#### Output Description

The model returns:

* **Trajectory Points**: 16 total points (start + 15 waypoints)
* **Sequential Labels**: Numbered from 0 to 15 indicating movement order
* **Smooth Path**: Points form a smooth curve from pen to organizer
* **Obstacle Consideration**: Path may curve to avoid obstacles on the desk
* **Processing Time**: 3-6 seconds
* **Visualization**: Numbered points showing the planned movement sequence

**Example Output Format**:

[  
 {"point": [420, 580], "label": "0"},  
 {"point": [400, 560], "label": "1"},  
 {"point": [380, 540], "label": "2"},  
 // ... intermediate points  
 {"point": [250, 280], "label": "14"},  
 {"point": [230, 270], "label": "15"}  
]

#### Use Case

* **Pick-and-Place Planning**: Generate movement paths for robotic arms
* **Motion Planning**: Plan smooth trajectories between locations
* **Task Execution**: Provide waypoints for autonomous navigation

### Example 11: Path for Brushing Particles

**Objective**: Generate a cleaning trajectory that covers a region with particles.

#### Procedure Description

This example demonstrates area coverage planning:

1. **Load Image**: Shows a plate with scattered particles and a blue brush
2. **Coverage Task**: Requests a path that covers all particles efficiently
3. **Starting Point**: Path begins at the brush location
4. **Even Distribution**: 10 points spread across the particle region
5. **Sequential Order**: Points are numbered to show the cleaning sequence

#### Code Snippet

img = get\_image\_resized("particles.jpg")  
points\_data = []  
  
prompt = textwrap.dedent("""\  
 Point to the the blue brush and a list of 10 points covering the region of  
 particles. Ensure that the points are spread evenly over the particles to  
 create a smooth trajectory.  
  
 Label the points from 1 to 10 based on the order that they should be  
 approached in the trajectory of cleaning the plate. Movement should start  
 from the brush.  
  
 The answer should follow the JSON format:  
 [{"point": <point>, "label": <label1>}, ...]  
  
 The points are in [y, x] format normalized to 0-1000.""")  
  
start\_time = time.time()  
config=types.GenerateContentConfig(temperature=0.5)  
json\_output = call\_gemini\_robotics\_er(img, prompt, config)  
  
try:  
 data = json.loads(json\_output)  
 points\_data.extend(data)  
except json.JSONDecodeError:  
 print("Warning: Invalid JSON response. Skipping.")  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json.dumps(points\_data)))

#### Output Description

The model returns:

* **Brush Location**: First point identifies the brush
* **Coverage Points**: 10 points strategically placed across the particle region
* **Efficient Path**: Points ordered to minimize travel distance
* **Even Distribution**: Points cover the entire particle area
* **Processing Time**: 3-5 seconds

**Example Output Format**:

[  
 {"point": [480, 220], "label": "brush"},  
 {"point": [450, 340], "label": "1"},  
 {"point": [420, 380], "label": "2"},  
 // ... coverage points  
 {"point": [380, 450], "label": "10"}  
]

#### Use Case

* **Cleaning Tasks**: Plan paths for wiping, brushing, or vacuuming
* **Area Coverage**: Ensure complete coverage of a region
* **Painting/Coating**: Plan brush strokes for even application

### Example 12: Obstacle-Avoidance Trajectory

**Objective**: Plan a collision-free path through a cluttered environment.

#### Procedure Description

This example demonstrates advanced path planning with obstacle avoidance:

1. **Load Scene**: Shows a living room with furniture (ottoman, coffee table, chairs)
2. **Navigation Task**: Plan path from current viewpoint to the green ottoman
3. **Obstacle Identification**: Model identifies furniture items that must be avoided
4. **Collision-Free Path**: Generates 10 waypoints that navigate around obstacles
5. **Floor-Based Path**: Points are on the floor surface for robot navigation

#### Code Snippet

img = get\_image\_resized("livingroom.jpeg")  
points\_data = []  
  
prompt = textwrap.dedent("""\  
 Find the most direct collision-free trajectory of 10 points on the floor  
 between the current view origin and the green ottoman in the back left.  
 The points should avoid all other obstacles on the floor.  
  
 The answer should follow the JSON format:  
 [{"point": <point>, "label": <label1>}, ...]  
  
 The points are in [y, x] format normalized to 0-1000.  
 """)  
  
start\_time = time.time()  
config=types.GenerateContentConfig(temperature=0.5)  
json\_output = call\_gemini\_robotics\_er(img, prompt, config)  
  
try:  
 data = json.loads(json\_output)  
 points\_data.extend(data)  
except json.JSONDecodeError:  
 print("Warning: Invalid JSON response. Skipping.")  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json.dumps(points\_data)))

#### Output Description

The model returns:

* **Collision-Free Waypoints**: 10 points that avoid furniture obstacles
* **Optimal Path**: Most direct route while maintaining safety clearance
* **Floor Navigation**: All points are on the floor plane
* **Goal-Directed**: Path leads to the specified destination (green ottoman)
* **Processing Time**: 4-6 seconds

**Example Output Format**:

[  
 {"point": [800, 500], "label": "waypoint 1"},  
 {"point": [750, 450], "label": "waypoint 2"},  
 {"point": [700, 380], "label": "waypoint 3"},  
 // ... path curving around coffee table  
 {"point": [250, 200], "label": "waypoint 10"}  
]

#### Use Case

* **Autonomous Navigation**: Plan safe paths for mobile robots
* **Obstacle Avoidance**: Navigate cluttered environments
* **Delivery Robots**: Move through spaces with dynamic obstacles

### Example 13: Item to Remove to Make Room

**Objective**: Identify which object should be removed to create space for a new item.

#### Procedure Description

This example demonstrates spatial reasoning and task planning:

1. **Load Scene**: Shows a desk with various objects
2. **Task Context**: User wants to place a laptop but there’s insufficient space
3. **Spatial Analysis**: Model evaluates which object occupies the needed space
4. **Reasoning**: Considers object size, position, and ease of removal
5. **Recommendation**: Points to the object that should be removed

#### Code Snippet

img = get\_image\_resized("clear\_space.png")  
  
prompt = textwrap.dedent("""\  
 Point to the object that I need to remove to make room for my laptop.  
  
 The answer should follow the JSON format:  
 [{"point": <point>, "label": <label1>}, ...]  
  
 The points are in [y, x] format normalized to 0-1000.""")  
  
start\_time = time.time()  
json\_output = call\_gemini\_robotics\_er(img, prompt)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json\_output))

#### Output Description

The model returns:

* **Target Object**: Single point identifying the object to remove
* **Reasoning**: Label may include explanation (e.g., “notebook occupying laptop space”)
* **Spatial Awareness**: Choice based on understanding laptop size requirements
* **Processing Time**: 2-4 seconds

**Example Output Format**:

[  
 {"point": [420, 380], "label": "notebook"}  
]

#### Use Case

* **Space Management**: Optimize workspace organization
* **Task Planning**: Prepare environments for specific tasks
* **Intelligent Rearrangement**: Make informed decisions about object placement

### Example 14: Orchestrating - Packing a Lunch

**Objective**: Generate step-by-step instructions for a complex task with object references.

#### Procedure Description

This example demonstrates task orchestration and instruction generation:

1. **Load Scene**: Shows lunch items, lunch box, and lunch bag
2. **Complex Task**: Pack the lunch box and bag appropriately
3. **Natural Language Instructions**: Model generates procedural steps
4. **Object References**: Each mentioned object is accompanied by a point
5. **Multi-Step Plan**: Complete sequence from start to finish

#### Code Snippet

img = get\_image\_resized("lunch.png")  
prompt = textwrap.dedent("""\  
 Explain how to pack the lunch box and lunch bag. Point to each object that  
 you refer to.  
  
 Each point should be in the format:  
 [{"point": [y, x], "label": }]  
 where the coordinates are normalized between 0-1000.  
 """)  
  
start\_time = time.time()  
json\_output = call\_gemini\_robotics\_er(img, prompt)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
img

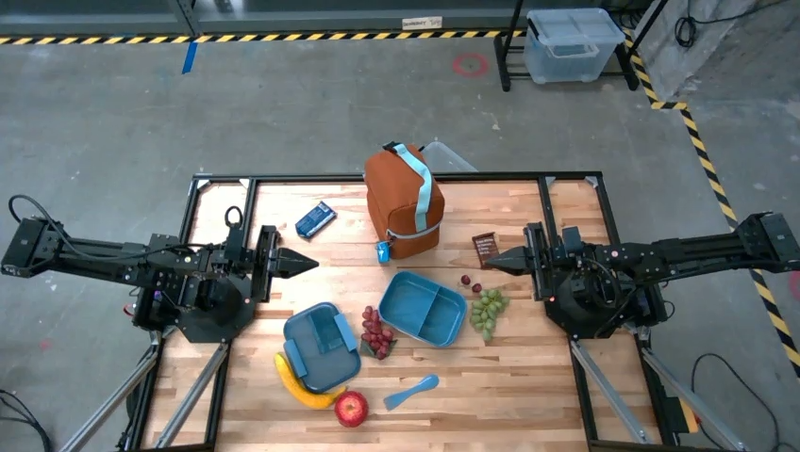
#### Output Description

The model returns:

* **Text Instructions**: Natural language steps for packing the lunch
* **Referenced Objects**: Points for each mentioned item (sandwich, fruit, drink, etc.)
* **Logical Sequence**: Steps in appropriate order (heavy items first, fragile items on top)
* **Complete Coverage**: All relevant items included in the plan
* **Processing Time**: 4-7 seconds

**Example Output Text + Points**:

Instructions:  
1. Place the sandwich in the lunch box (pointing to sandwich)  
2. Add the apple next to the sandwich (pointing to apple)  
3. Put the juice box in the lunch bag (pointing to juice box)  
4. Place the lunch box in the lunch bag (pointing to lunch box)  
  
Points:  
[  
 {"point": [380, 420], "label": "sandwich"},  
 {"point": [320, 520], "label": "apple"},  
 {"point": [450, 350], "label": "juice box"},  
 {"point": [400, 450], "label": "lunch box"}  
]



Lunch Packing Output

#### Use Case

* **Task Instruction**: Generate human-readable instructions for tasks
* **Assembly Guidance**: Provide step-by-step assembly instructions
* **Training**: Create training materials for repetitive tasks

### Example 15: Empty Electrical Sockets

**Objective**: Identify unoccupied and accessible electrical outlets.

#### Procedure Description

This example demonstrates functional state detection:

1. **Load Scene**: Shows an electrical outlet panel with multiple sockets
2. **State Detection**: Identify which sockets are empty (not plugged in)
3. **Accessibility Check**: Ensure sockets are unobstructed
4. **Functional Reasoning**: Understand that only empty, accessible sockets are usable

#### Code Snippet

img = get\_image\_resized("sockets.jpeg")  
prompt = textwrap.dedent("""\  
 Point to the unobstructed empty sockets.  
  
 The answer should follow the JSON format:  
 [{"point": <point>, "label": <label1>}, ...]  
  
 The points are in [y, x] format normalized to 0-1000.""")  
  
start\_time = time.time()  
json\_output = call\_gemini\_robotics\_er(img, prompt)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json\_output))

#### Output Description

The model returns:

* **Empty Sockets Only**: Points only to unplugged sockets
* **Unobstructed Check**: Excludes sockets blocked by other plugs or objects
* **Multiple Detections**: All available sockets are identified
* **Processing Time**: 2-4 seconds

**Example Output Format**:

[  
 {"point": [320, 280], "label": "empty socket top-left"},  
 {"point": [450, 380], "label": "empty socket bottom-right"}  
]

#### Use Case

* **State Detection**: Identify functional vs. occupied states
* **Resource Availability**: Check which resources are available
* **Safety Verification**: Ensure safe connection points

### Example 16: Physical Constraints (Weight Limit)

**Objective**: Identify objects that meet physical constraints (e.g., weight limit).

#### Procedure Description

This example demonstrates physical reasoning with constraints:

1. **Load Scene**: Shows various objects of different weights
2. **Constraint Specification**: Robot has 3 LB payload limit
3. **Weight Estimation**: Model estimates object weights based on appearance
4. **Filtering**: Returns only objects within the payload capacity
5. **Thinking Mode**: Uses extended thinking budget (-1 = unlimited) for complex reasoning

#### Code Snippet

img = get\_image\_resized("weights.jpeg")  
  
prompt = textwrap.dedent("""\  
 I am a robot with a payload of 3LBs. Point to all the objects in the image I  
 am physically able to pick up.  
  
 The answer should follow the JSON format:  
 [{"point": <point>, "label": <label1>}, ...]  
  
 The points are in [y, x] format normalized to 0-1000.""")  
  
start\_time = time.time()  
config=types.GenerateContentConfig(  
 temperature=0.5,  
 thinking\_config=types.ThinkingConfig(thinking\_budget=-1),  
)  
json\_output = call\_gemini\_robotics\_er(img, prompt, config)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
IPython.display.HTML(generate\_point\_html(img, json\_output))

#### Output Description

The model returns:

* **Filtered Objects**: Only items estimated to be ≤3 LBS
* **Weight Reasoning**: Model uses visual cues (size, material) to estimate weight
* **Excluded Objects**: Heavy items (dumbbells, kettlebells) are not included
* **Included Objects**: Light items (small weights, books, remote) are included
* **Processing Time**: 5-8 seconds (longer due to thinking mode)
* **Extended Reasoning**: Thinking budget allows deeper analysis

**Example Output Format**:

[  
 {"point": [280, 350], "label": "1 LB weight"},  
 {"point": [320, 450], "label": "2 LB weight"},  
 {"point": [450, 520], "label": "book"},  
 {"point": [380, 280], "label": "remote control"}  
]

#### Use Case

* **Payload Planning**: Ensure robot operates within safe limits
* **Material Handling**: Select appropriate objects for manipulation
* **Safety Compliance**: Avoid overloading robotic systems

### Example 17: Video Analysis with Timestamps

**Objective**: Analyze a video and provide timestamped descriptions of actions.

#### Procedure Description

This example demonstrates temporal video understanding:

1. **Upload Video**: Sends an MP4 file to the API using file upload
2. **Wait for Processing**: Monitors file processing status
3. **Temporal Analysis**: Model analyzes the entire video sequence
4. **Timestamped Breakdown**: Returns start/end timestamps with descriptions
5. **Extended Thinking**: Uses unlimited thinking budget for complex video analysis

#### Code Snippet

myfile = client.files.upload(file="/content/desk\_organization.mp4")  
while myfile.state == "PROCESSING":  
 print(".", end="")  
 time.sleep(1)  
 myfile = client.files.get(name=myfile.name)  
  
if myfile.state.name == "FAILED":  
 raise ValueError(myfile.state.name)  
  
print("Uploaded")  
  
prompt = textwrap.dedent("""\  
 Describe in detail each step of finishing the task. Breaking it down by  
 timestamp, output in JSON format with keys "start\_timestamp",  
 "end\_timestamp" and "description".""")  
  
start\_time = time.time()  
  
response = client.models.generate\_content(  
 model=MODEL\_ID,  
 contents=[myfile, prompt],  
 config=types.GenerateContentConfig(  
 temperature=0.5,  
 thinking\_config=types.ThinkingConfig(thinking\_budget=-1),  
 ),  
)  
  
end\_time = time.time()  
elapsed\_time = end\_time - start\_time  
print(f"\nTotal processing time: {elapsed\_time:.4f} seconds")  
  
print(response.text)  
  
video\_widget = widgets.Video.from\_file("/content/desk\_organization.mp4")  
display.display(video\_widget)

#### Output Description

The model returns:

* **JSON Array**: List of temporal segments with timestamps
* **Start/End Times**: Timestamps in seconds (e.g., “0.0s - 3.2s”)
* **Detailed Descriptions**: What happens in each segment
* **Action Breakdown**: Identifies distinct actions (reach, grasp, move, place)
* **Processing Time**: 10-20 seconds depending on video length
* **Video Display**: Original video shown for reference

**Example Output Format**:

[  
 {  
 "start\_timestamp": "0.0s",  
 "end\_timestamp": "2.5s",  
 "description": "Robot arm approaches the pen on the desk"  
 },  
 {  
 "start\_timestamp": "2.5s",  
 "end\_timestamp": "5.1s",  
 "description": "Gripper closes around the pen"  
 },  
 {  
 "start\_timestamp": "5.1s",  
 "end\_timestamp": "8.3s",  
 "description": "Arm lifts pen and moves toward organizer"  
 },  
 {  
 "start\_timestamp": "8.3s",  
 "end\_timestamp": "10.0s",  
 "description": "Pen is placed in the organizer"  
 }  
]

#### Use Case

* **Action Recognition**: Understand what actions occur in videos
* **Task Monitoring**: Analyze recorded robot operations
* **Documentation**: Auto-generate task descriptions from videos
* **Verification**: Confirm tasks were completed as expected

### Example 18: Video Analysis - Time Range Zoom

**Objective**: Perform detailed analysis of a specific time segment in a video.

#### Procedure Description

This example demonstrates multi-turn conversation with temporal focus:

1. **Conversation History**: Uses results from previous video analysis
2. **Create Chat**: Initializes chat with conversation history
3. **Focused Analysis**: Requests per-second breakdown of seconds 15-22
4. **Detailed Breakdown**: Higher temporal resolution for the specified segment
5. **Multi-Turn Context**: Model understands reference to “same format” from previous exchange

#### Code Snippet

conversation\_history = [  
 {"role": "user", "parts": [{"text": prompt}]},  
 {"role": "model", "parts": [{"text": response.text}]},  
]  
  
chat = client.chats.create(model=MODEL\_ID, history=conversation\_history)  
  
prompt = textwrap.dedent("""\  
 Zoom into second 15 to 22 and provide a per-second breakdown of what is  
 happening in the same format.""")  
  
start\_time = time.time()  
  
response = chat.send\_message([prompt, myfile])  
  
end\_time = time.time()  
elapsed\_time = end\_time - start\_time  
print(f"\nTotal processing time: {elapsed\_time:.4f} seconds")  
  
print(response.text)

#### Output Description

The model returns:

* **Fine-Grained Timestamps**: Breakdown by second or sub-second intervals
* **Detailed Actions**: More specific descriptions of movements
* **Same Format**: Maintains JSON structure from previous response
* **Focused Segment**: Only covers the requested 15-22 second range
* **Processing Time**: 8-15 seconds

**Example Output Format**:

[  
 {  
 "start\_timestamp": "15.0s",  
 "end\_timestamp": "16.0s",  
 "description": "Gripper fully opens in preparation"  
 },  
 {  
 "start\_timestamp": "16.0s",  
 "end\_timestamp": "17.0s",  
 "description": "Arm descends toward the target object"  
 },  
 // ... continues for each second 15-22  
]

#### Use Case

* **Detailed Analysis**: Investigate specific moments in detail
* **Failure Analysis**: Zoom into problematic time segments
* **Multi-Turn Queries**: Progressive refinement of analysis

### Example 19: Finding the Fourth Row of Shelves

**Objective**: Identify specific rows in a structured environment using spatial reasoning.

#### Procedure Description

This example demonstrates ordinal spatial understanding:

1. **Load Image**: Shows a bookshelf with multiple rows
2. **Ordinal Query**: Requests the “fourth row” requiring counting from top/bottom
3. **Cubby Detection**: Identifies individual storage compartments in that row
4. **Bounding Boxes**: Returns boxes around each cubby in the fourth row
5. **Row Understanding**: Model must understand row structure and counting

#### Code Snippet

img = get\_image\_resized("bookshelf.jpeg")  
  
prompt = textwrap.dedent("""\  
 Return bounding boxes as a JSON array with labels highlighting all cubbies  
 in the fourth row of shelves.  
  
 The format should be as follows:  
 [{"box\_2d": [ymin, xmin, ymax, xmax], "label": <label for the object>}]  
  
 normalized to 0-1000. The values in box\_2d must only be integers.""")  
  
start\_time = time.time()  
config=types.GenerateContentConfig(temperature=0.5)  
json\_output = call\_gemini\_robotics\_er(img, prompt, config)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
plot\_bounding\_boxes(img, json\_output)  
img

#### Output Description

The model returns:

* **Row-Specific Boxes**: Only cubbies in the fourth row are included
* **Multiple Cubbies**: All compartments in that row are detected
* **Position Labels**: May include position indicators (left, center, right)
* **Accurate Counting**: Correctly identifies the fourth row from top or bottom
* **Processing Time**: 3-5 seconds

**Example Output Format**:

[  
 {"box\_2d": [450, 100, 550, 250], "label": "fourth row left cubby"},  
 {"box\_2d": [450, 260, 550, 410], "label": "fourth row center-left cubby"},  
 {"box\_2d": [450, 420, 550, 570], "label": "fourth row center-right cubby"},  
 {"box\_2d": [450, 580, 550, 730], "label": "fourth row right cubby"}  
]



Fourth Row Shelves Output

#### Use Case

* **Structured Environments**: Navigate organized storage systems
* **Ordinal Reasoning**: Access items by position (first, second, third…)
* **Warehouse Automation**: Pick items from specific shelf locations

### Example 20: Finding Shelves with Specific Items

**Objective**: Locate storage locations based on functional needs.

#### Procedure Description

This example demonstrates semantic search in physical spaces:

1. **Load Image**: Same bookshelf image
2. **Functional Query**: “I need to blow my nose” - indirect item reference
3. **Semantic Reasoning**: Model infers user needs tissues/kleenex
4. **Location Identification**: Finds the cubby containing tissues
5. **Indirect Reference**: Demonstrates understanding of use cases

#### Code Snippet

img = get\_image\_resized("bookshelf.jpeg")  
  
prompt = textwrap.dedent("""\  
 "I need to blow my nose."  
 Find the cubby that can help.  
  
 The format should be as follows:  
 [{"box\_2d": [ymin, xmin, ymax, xmax], "label": <label for the object>}]  
  
 normalized to 0-1000. The values in box\_2d must only be integers.""")  
  
start\_time = time.time()  
config=types.GenerateContentConfig(temperature=0.5)  
json\_output = call\_gemini\_robotics\_er(img, prompt, config)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
plot\_bounding\_boxes(img, json\_output)  
img

#### Output Description

The model returns:

* **Functional Match**: Identifies the cubby with tissues/kleenex
* **Semantic Understanding**: Connects “blow nose” with tissues
* **Single Result**: Returns the most relevant cubby
* **Label Explanation**: May include explanation (e.g., “tissue box”)
* **Processing Time**: 3-5 seconds

**Example Output Format**:

[  
 {"box\_2d": [250, 420, 350, 570], "label": "cubby with tissue box"}  
]



Specific Items Output

#### Use Case

* **Natural Language Search**: Find items by describing needs
* **Smart Assistants**: Respond to functional queries
* **Human-Robot Interaction**: Understand indirect requests

### Example 21: Counting Items with Thinking

**Objective**: Count items in a complex scene using extended reasoning.

#### Procedure Description

This example demonstrates counting with reasoning transparency:

1. **Load Image**: Shopping cart with multiple items
2. **Counting Task**: Count items inside the cart basket
3. **Reasoning Request**: Explicitly asks the model to share its reasoning
4. **Thinking Mode**: Uses default thinking budget for analysis
5. **Explanation**: Model explains how it arrived at the count

#### Code Snippet

img = get\_image\_resized("cart.png")  
  
prompt = textwrap.dedent("""\  
 How many items are inside of the cart basket?  
 Please share your reasoning.""")  
  
start\_time = time.time()  
config=types.GenerateContentConfig(temperature=0.5)  
json\_output = call\_gemini\_robotics\_er(img, prompt, config)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
img

#### Output Description

The model returns:

* **Count Number**: Total number of items in the cart
* **Reasoning Explanation**: How items were identified and counted
* **Item List**: May enumerate what items were counted
* **Ambiguity Handling**: Explains any uncertain cases
* **Processing Time**: 4-7 seconds

**Example Output Format**:

There are 5 items in the cart basket.  
  
Reasoning:  
1. I can see a loaf of bread in the front left  
2. A box of cereal in the back left  
3. A bag of chips in the center  
4. A container of milk on the right side  
5. A package of cookies partially visible under the milk  
  
Total count: 5 items

#### Use Case

* **Inventory Verification**: Count items in containers or carts
* **Explainable AI**: Understand model reasoning for transparency
* **Quality Assurance**: Verify counts with explanations

### Example 22: Multi-View Success Detection

**Objective**: Evaluate task completion using multiple camera views.

#### Procedure Description

This example demonstrates multi-view analysis for task verification:

1. **Load Multiple Images**: 8 total images - 4 initial state views + 4 current state views
2. **Temporal Comparison**: Compare initial vs. current state across views
3. **Multi-Camera Fusion**: Combine information from static and robot-mounted cameras
4. **Success Evaluation**: Determine if task (put mango in container) was completed
5. **Binary Output**: Returns (1) for yes or (2) for no

#### Code Snippet

initial\_state\_1 = Image.open("initial\_state\_1.png")  
initial\_state\_2 = Image.open("initial\_state\_2.png")  
initial\_state\_3 = Image.open("initial\_state\_3.png")  
initial\_state\_4 = Image.open("initial\_state\_4.png")  
current\_state\_1 = Image.open("current\_state\_1.png")  
current\_state\_2 = Image.open("current\_state\_2.png")  
current\_state\_3 = Image.open("current\_state\_3.png")  
current\_state\_4 = Image.open("current\_state\_4.png")  
  
prompt = textwrap.dedent("""\  
 For this task, you will see a robot or human trying to perform the task of  
 putting the mango into the brown container. You may see multiple camera  
 views of the same scene. Some cameras are static and are mounted outside of  
 the scene and some cameras are mounted on the robot arms and thus they are  
 moving during the episode.  
  
 The first 4 images show multiple camera views from the start of the episode  
 (some time ago). The last 4 images show multiple camera views from the  
 current moment in the episode (as it is now).  
  
 Looking at these images and comparing the start of the episode with current  
 state did the robot successfully perform the task "put the mango into the  
 brown container"?  
  
 Answer only with (1) yes or (2) no. Return the number (1) or (2) that best  
 answers the question.""")  
  
start\_time = time.time()  
  
response = client.models.generate\_content(  
 model=MODEL\_ID,  
 contents=[  
 initial\_state\_1,  
 initial\_state\_2,  
 initial\_state\_3,  
 initial\_state\_4,  
 current\_state\_1,  
 current\_state\_2,  
 current\_state\_3,  
 current\_state\_4,  
 prompt  
 ],  
 config=types.GenerateContentConfig(temperature=0.5),  
)  
  
end\_time = time.time()  
elapsed\_time = end\_time - start\_time  
  
print(f"\nTotal processing time: {elapsed\_time:.4f} seconds")  
print(f"Success? {'Yes' if response.text == '(1)' else 'No'}")

#### Output Description

The model returns:

* **Binary Response**: “(1)” for successful completion or “(2)” for failure
* **Multi-View Analysis**: Synthesizes information across all 8 images
* **Temporal Understanding**: Compares before and after states
* **Camera Fusion**: Handles both static and moving camera perspectives
* **Processing Time**: 8-15 seconds due to processing 8 images

**Example Output**:

(1)

Interpretation: Yes, the task was completed successfully - the mango is now in the brown container.

#### Use Case

* **Task Verification**: Confirm robotic tasks were completed correctly
* **Multi-Camera Systems**: Use multiple viewpoints for robust detection
* **Quality Control**: Automated inspection with high confidence
* **Failure Detection**: Identify when tasks fail for recovery

### Example 23: Image Enhancement with Code Execution

**Objective**: Use code execution to enhance image analysis.

#### Procedure Description

This example demonstrates the code execution capability:

1. **Load Image**: Air quality sensor display
2. **Reading Challenge**: Small text that’s hard to read
3. **Code Execution Tool**: Model can generate and run Python code
4. **Image Processing**: Code zooms into specific regions
5. **Enhanced Reading**: Better accuracy through image manipulation

#### Code Snippet

img = get\_image\_resized("air\_quality.jpeg")  
  
prompt = textwrap.dedent("""\  
 What is the air quality reading? Using the code execution feature, zoom in  
 on the image to take a closer look.""")  
  
  
start\_time = time.time()  
response = client.models.generate\_content(  
 model=MODEL\_ID,  
 contents=[img, prompt],  
 config=types.GenerateContentConfig(  
 temperature=0.5,  
 tools=[types.Tool(code\_execution=types.ToolCodeExecution)],  
 ),  
)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
  
for part in response.candidates[0].content.parts:  
 if part.text is not None:  
 print(part.text)  
 if part.executable\_code is not None:  
 print(part.executable\_code.code)  
 if part.code\_execution\_result is not None:  
 print(part.code\_execution\_result.output)  
  
img

#### Output Description

The model returns:

* **Generated Code**: Python code to crop/zoom the image
* **Code Execution Result**: Output from running the code
* **Enhanced Reading**: More accurate reading of the display
* **Multi-Part Response**: Text explanation + code + execution result
* **Processing Time**: 5-10 seconds

**Example Output**:

Text: "I'll zoom in on the display to read the air quality value more clearly."  
  
Executable Code:  
```python  
from PIL import Image  
import numpy as np  
  
# Crop the display region  
display\_region = img.crop((300, 200, 500, 350))  
display\_region = display\_region.resize((400, 300))  
display\_region.show()

Code Execution Result: “Display region isolated successfully”

Final Answer: “The air quality reading is 45 AQI (Moderate)”

#### Use Case  
  
- \*\*OCR Enhancement\*\*: Improve text reading accuracy  
- \*\*Image Analysis\*\*: Perform complex image processing  
- \*\*Adaptive Behavior\*\*: Model can write code to solve problems  
- \*\*Multi-Step Processing\*\*: Combine vision with computation  
  
---  
  
### Example 24: Segmentation Masks  
  
\*\*Objective\*\*: Generate pixel-precise segmentation masks for objects.  
  
#### Procedure Description  
  
This example demonstrates precise object segmentation:  
  
1. \*\*Load Image\*\*: Robot gripper holding a mango  
2. \*\*Multi-Object Segmentation\*\*: Request masks for mango and gripper fingers  
3. \*\*Mask Generation\*\*: Model returns base64-encoded PNG masks  
4. \*\*Mask Parsing\*\*: Decode and process the segmentation masks  
5. \*\*Visualization\*\*: Overlay colored masks on the original image  
  
#### Code Snippet  
  
```python  
img = get\_image\_resized("mango.png")  
  
prompt = textwrap.dedent("""\  
 Provide the segmentation masks for the following objects in this image:  
 mango, left robot gripper finger, right robot gripper finger.  
  
 The answer should follow the JSON format:  
 [  
 {  
 "box\_2d": [ymin, xmin, ymax, xmax],  
 "label": "<label for the object>",  
 "mask": "data:image/png;base64,<base64 encoded PNG mask>"  
 },  
 ...  
 ]  
  
 The box\_2d coordinates should be normalized to 0-1000 and must be integers.  
 The mask should be a base64 encoded PNG image where non-zero pixels indicate  
 the mask.""")  
  
start\_time = time.time()  
config=types.GenerateContentConfig(temperature=0.5)  
print("Raw Model Response Text:")  
json\_output = call\_gemini\_robotics\_er(img, prompt, config)  
  
print(f"\nTotal processing time: {(time.time() - start\_time):.4f} seconds")  
  
try:  
 segmentation\_masks = parse\_segmentation\_masks(  
 json\_output, img\_height=img.size[1], img\_width=img.size[0]  
 )  
 print(f"Successfully parsed {len(segmentation\_masks)} segmentation masks.")  
  
 annotated\_img = plot\_segmentation\_masks(  
 img.convert("RGBA"), segmentation\_masks  
 )  
 display.display(annotated\_img)  
  
except json.JSONDecodeError as e:  
 print(f"Error decoding JSON response: {e}")  
except Exception as e:  
 print(f"An error occurred during mask processing or plotting: {e}")

#### Output Description

The model returns:

* **JSON Array**: Each object with box\_2d, label, and base64 mask
* **Bounding Boxes**: Rough region for each object
* **Pixel Masks**: Base64-encoded PNG where white pixels indicate the object
* **Multi-Object**: Separate mask for each requested object
* **Processing Time**: 5-8 seconds
* **Visualization**: Image with colored transparent overlays for each mask

**Example Output Format**:

[  
 {  
 "box\_2d": [320, 280, 520, 480],  
 "label": "mango",  
 "mask": "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAA..."  
 },  
 {  
 "box\_2d": [380, 200, 480, 280],  
 "label": "left robot gripper finger",  
 "mask": "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAA..."  
 },  
 {  
 "box\_2d": [380, 480, 480, 560],  
 "label": "right robot gripper finger",  
 "mask": "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAA..."  
 }  
]

#### Use Case

* **Precise Manipulation**: Know exact object boundaries for grasping
* **Quality Inspection**: Detect defects or damage on specific parts
* **Scene Understanding**: Separate overlapping or touching objects
* **Training Data**: Generate segmentation masks for ML training

### Example 25: Robot Function Call Generation

**Objective**: Generate executable robot function calls for pick-and-place tasks.

#### Procedure Description

This example demonstrates end-to-end task execution:

1. **Object Localization**: First uses pointing to find blue block and orange bowl
2. **Coordinate Extraction**: Extracts point coordinates from detection results
3. **Relative Coordinates**: Calculates coordinates relative to robot origin
4. **Function Call Planning**: Model generates sequence of robot function calls
5. **Reasoning Included**: Explains the pick-and-place strategy
6. **Execution**: Calls are executed in sequence with print statements

#### Code Snippet

# Step 1: Locate objects  
img = get\_image\_resized("soarm-block.png")  
points\_data = []  
  
prompt = textwrap.dedent("""\  
 Locate and point to the blue block and the orange bowl. The label returned  
 should be an identifying name for the object detected.  
  
 The answer should follow the JSON format:  
 [{"point": <point>, "label": <label1>}, ...]  
  
 The points are in [y, x] format normalized to 0-1000.""")  
  
start\_time = time.time()  
json\_output = call\_gemini\_robotics\_er(img, prompt)  
data = json.loads(json\_output)  
points\_data.extend(data)  
  
# Step 2: Extract coordinates and calculate relative positions  
robot\_origin\_y = 300  
robot\_origin\_x = 500  
  
blue\_block\_point = None  
orange\_bowl\_point = None  
  
for item in points\_data:  
 if item.get("label") == "blue block":  
 blue\_block\_point = item.get("point")  
 elif item.get("label") == "orange bowl":  
 orange\_bowl\_point = item.get("point")  
  
if blue\_block\_point and orange\_bowl\_point:  
 block\_y, block\_x = blue\_block\_point  
 bowl\_y, bowl\_x = orange\_bowl\_point  
  
 block\_relative\_x = block\_x - robot\_origin\_x  
 block\_relative\_y = block\_y - robot\_origin\_y  
 bowl\_relative\_x = bowl\_x - robot\_origin\_x  
 bowl\_relative\_y = bowl\_y - robot\_origin\_y  
  
 # Step 3: Generate function calls  
 prompt = textwrap.dedent(f"""\  
 You are a robotic arm with six degrees-of-freedom. You have the following  
 functions available to you:  
  
 def move(x, y, high):  
 # Moves the arm to the given coordinates. The boolean value 'high' set  
 # to True means the robot arm should be lifted above the scene for  
 # avoiding obstacles during motion. 'high' set to False means the robot  
 # arm should have the gripper placed on the surface for interacting with  
 # objects.  
  
 def setGripperState(opened):  
 # Opens the gripper if opened set to true, otherwise closes the gripper  
  
 def returnToOrigin():  
 # Returns the robot to an initial state. Should be called as a cleanup  
 # operation.  
  
 The origin point for calculating the moves is at normalized point  
 y={robot\_origin\_y}, x={robot\_origin\_x}. Use this as the new (0,0) for  
 calculating moves, allowing x and y to be negative.  
  
 Perform a pick and place operation where you pick up the blue block at  
 normalized coordinates ({block\_x}, {block\_y}) (relative coordinates:  
 {block\_relative\_x}, {block\_relative\_y}) and place it into the orange bowl  
 at normalized coordinates ({bowl\_x}, {bowl\_y}) (relative coordinates:  
 {bowl\_relative\_x}, {bowl\_relative\_y}).  
 Provide the sequence of function calls as a JSON list of objects, where  
 each object has a "function" key (the function name) and an "args" key  
 (a list of arguments for the function).  
  
 Also, include your reasoning before the JSON output.""")  
  
 json\_output = call\_gemini\_robotics\_er(img, prompt, config)  
  
 # Step 4: Execute function calls  
 try:  
 function\_calls = json.loads(json\_output)  
  
 print("\nExecuting Function Calls:")  
 for call in function\_calls:  
 function\_name = call.get("function")  
 arguments = call.get("args", [])  
  
 if function\_name == "move":  
 move(\*arguments)  
 elif function\_name == "setGripperState":  
 setGripperState(\*arguments)  
 elif function\_name == "returnToOrigin":  
 returnToOrigin()  
  
 except json.JSONDecodeError:  
 print("Error: Could not parse JSON response from the model.")

#### Output Description

The model returns:

* **Reasoning**: Explanation of the pick-and-place strategy
* **Function Sequence**: JSON array of function calls with arguments
* **Proper Ordering**: Moves are sequenced logically (approach→grasp→lift→move→place→open→return)
* **High/Low Moves**: Uses ‘high’ parameter to avoid collisions during transit
* **Gripper Control**: Opens and closes gripper at appropriate times

**Example Output Format**:

Reasoning:  
To pick up the blue block and place it in the orange bowl, I will:  
1. Move to a high position above the block  
2. Open the gripper  
3. Move down to the block  
4. Close the gripper around the block  
5. Lift the block (high move)  
6. Move to a high position above the bowl  
7. Move down to place in the bowl  
8. Open the gripper to release  
9. Return to origin  
  
Function Calls:  
```json  
[  
 {"function": "move", "args": [80, -20, true]},  
 {"function": "setGripperState", "args": [true]},  
 {"function": "move", "args": [80, -20, false]},  
 {"function": "setGripperState", "args": [false]},  
 {"function": "move", "args": [80, -20, true]},  
 {"function": "move", "args": [-50, 120, true]},  
 {"function": "move", "args": [-50, 120, false]},  
 {"function": "setGripperState", "args": [true]},  
 {"function": "returnToOrigin", "args": []}  
]

**Execution Output**:

Executing Function Calls:  
moving to coordinates: 80, -20, 15  
Opening gripper  
moving to coordinates: 80, -20, 5  
Closing gripper  
moving to coordinates: 80, -20, 15  
moving to coordinates: -50, 120, 15  
moving to coordinates: -50, 120, 5  
Opening gripper  
Returning to origin pose

#### Use Case

* **Task Automation**: Convert high-level instructions to executable commands
* **Pick-and-Place**: Automate object manipulation tasks
* **Human-Robot Interface**: Natural language to robot commands
* **Code Generation**: Automatically generate robot control programs

## Advanced Features

### 1. Thinking Mode

The model supports configurable thinking for complex reasoning:

config = types.GenerateContentConfig(  
 temperature=0.5,  
 thinking\_config=types.ThinkingConfig(  
 thinking\_budget=-1 # -1 = unlimited, 0 = disabled, >0 = limited tokens  
 )  
)

**When to Use**: - Complex spatial reasoning tasks - Multi-step planning - Weight/constraint estimation - Ambiguous scenarios requiring analysis

### 2. Code Execution

Enable the model to write and execute code:

config = types.GenerateContentConfig(  
 temperature=0.5,  
 tools=[types.Tool(code\_execution=types.ToolCodeExecution)]  
)

**Use Cases**: - Image enhancement (zoom, crop, adjust) - Mathematical calculations - Data transformation - Custom processing pipelines

### 3. Multi-Turn Conversations

Maintain context across multiple exchanges:

conversation\_history = [  
 {"role": "user", "parts": [{"text": first\_prompt}]},  
 {"role": "model", "parts": [{"text": first\_response.text}]},  
]  
  
chat = client.chats.create(model=MODEL\_ID, history=conversation\_history)  
response = chat.send\_message([follow\_up\_prompt, image])

**Benefits**: - Progressive refinement - Follow-up questions - Task continuation - Context preservation

### 4. File Upload

Process large video files:

myfile = client.files.upload(file="/path/to/video.mp4")  
while myfile.state == "PROCESSING":  
 time.sleep(1)  
 myfile = client.files.get(name=myfile.name)  
  
response = client.models.generate\_content(  
 model=MODEL\_ID,  
 contents=[myfile, prompt]  
)

**Supported Formats**: - Video: MP4, MOV, AVI - Audio: MP3, WAV - Large images

### 5. Batch Processing with Parallelization

Process multiple queries efficiently:

with concurrent.futures.ThreadPoolExecutor() as executor:  
 results = executor.map(process\_query, queries)

**Performance Gains**: - 4-5x speedup for independent queries - Optimal resource utilization - Reduced total processing time

## Best Practices

### 1. Coordinate System Understanding

**Key Points**: - Always use normalized coordinates (0-1000) - Format is [y, x] - y first, then x - Convert to pixels: pixel = (normalized / 1000) \* dimension - Top-left origin (0, 0)

### 2. Prompt Engineering

**Effective Prompts**: - Be specific about desired output format - Request JSON for programmatic parsing - Specify coordinate normalization requirements - Include examples when possible - Use clear, unambiguous language

**Example**:

prompt = textwrap.dedent("""\  
 Point to the red cup on the table.  
   
 Return the answer in JSON format:  
 [{"point": [y, x], "label": "red cup"}]  
   
 Coordinates should be normalized to 0-1000 range.  
 """)

### 3. Error Handling

**Robust Implementation**:

try:  
 json\_output = call\_gemini\_robotics\_er(img, prompt)  
 data = json.loads(json\_output)  
 # Process data  
except json.JSONDecodeError as e:  
 print(f"JSON parsing error: {e}")  
 # Fallback logic  
except Exception as e:  
 print(f"Unexpected error: {e}")  
 # Error recovery

### 4. Performance Optimization

**Tips**: - Use parallel processing for independent queries - Reduce image resolution when appropriate - Sample video frames instead of processing every frame - Cache repeated queries - Adjust thinking budget based on task complexity

### 5. Temperature Settings

**Recommendations**: - temperature=0.0-0.3: Deterministic, consistent results (detection tasks) - temperature=0.4-0.7: Balanced creativity and consistency (planning tasks) - temperature=0.8-1.0: Creative, varied outputs (exploration tasks)

### 6. Image Preparation

**Best Practices**: - Resize large images (800-1200px width optimal) - Use good lighting conditions - Minimize motion blur - Ensure objects are visible - Use appropriate camera angles

## Troubleshooting

### Issue: Inconsistent Detection Results

**Possible Causes**: - High temperature setting - Poor image quality - Ambiguous prompt

**Solutions**: - Lower temperature to 0.2-0.3 - Improve image lighting and resolution - Make prompt more specific - Add examples to prompt

### Issue: Slow Processing Times

**Possible Causes**: - Large images - High thinking budget - Sequential processing

**Solutions**: - Resize images to 800-1200px - Reduce thinking budget if not needed - Use parallel processing with ThreadPoolExecutor - Sample frames for video instead of processing all

### Issue: JSON Parsing Errors

**Possible Causes**: - Model returned text before/after JSON - Invalid JSON format - Incomplete response

**Solutions**: - Use parse\_json() helper function to extract JSON - Add explicit JSON format requirements to prompt - Implement try-except blocks - Request “Never return code fencing” in prompt

### Issue: Incorrect Coordinate Values

**Possible Causes**: - Misunderstanding of coordinate system - Wrong conversion formula - Image resize not accounted for

**Solutions**: - Verify coordinates are in 0-1000 range - Check [y, x] order (not [x, y]) - Apply same resize to visualization as input - Test with known reference points

### Issue: Missing Objects in Detection

**Possible Causes**: - Objects too small - Poor contrast - Occlusion - Ambiguous prompt

**Solutions**: - Use higher resolution images - Improve lighting - Try multiple camera angles - Be more specific in object description - Use abstract categories if specific names don’t work

### Issue: Video Upload Failures

**Possible Causes**: - File too large - Unsupported format - Network issues

**Solutions**: - Compress video before upload - Convert to MP4 format - Check file size limits - Retry with exponential backoff - Use frame sampling instead

## Conclusion

The **Gemini Robotics ER 1.5 Preview** model represents a powerful tool for building intelligent robotic systems. Its combination of vision understanding, spatial reasoning, and natural language processing enables a wide range of applications from simple object detection to complex task orchestration.

### Key Takeaways

1. **Versatile Capabilities**: From point detection to function call generation
2. **Multi-Modal Understanding**: Process images, video, and text together
3. **Structured Outputs**: JSON format for easy integration
4. **Advanced Reasoning**: Thinking mode and code execution for complex tasks
5. **Scalable Performance**: Parallel processing and optimization techniques

### Next Steps

* Experiment with different prompts and parameters
* Combine multiple capabilities for complex workflows
* Integrate with your robot control systems
* Build multi-turn conversational interfaces
* Optimize for your specific use cases

### Resources

* [Google AI Gemini API Documentation](https://ai.google.dev/gemini-api/docs/robotics-overview)
* [API Reference](https://ai.google.dev/api/python)
* [Google AI Studio](https://aistudio.google.com/)
* [Community Forums](https://discuss.ai.google.dev/)

**Document Version**: 1.0  
**Last Updated**: 2024  
**Model Version**: gemini-robotics-er-1.5-preview  
**Author**: Generated from official Google Colab notebook

*This documentation is based on the official Gemini Robotics ER 1.5 Preview notebook provided by Google. All code examples, procedures, and outputs are derived from the notebook content and Google AI documentation.*