Gemini Robotics-ER 1.5

Preview: The Gemini Robotics-ER 1.5 model is currently in preview.

Gemini Robotics-ER 1.5 is a vision-language model (VLM) that brings Gemini's agentic capabilities to robotics. It's designed for advanced reasoning in the physical world, allowing robots to interpret complex visual data, perform spatial reasoning, and plan actions from natural language commands.

Key features and benefits:

- **Enhanced autonomy:** Robots can reason, adapt, and respond to changes in open-ended environments.
- **Natural language interaction:** Makes robots easier to use by enabling complex task assignments using natural language.
- **Task orchestration:** Deconstructs natural language commands into subtasks and integrates with existing robot controllers and behaviors to complete long-horizon tasks.
- **Versatile capabilities:** Locates and identifies objects, understands object relationships, plans grasps and trajectories, and interprets dynamic scenes.

This document describes what the model does (#how-it-works) and takes you through several examples (#agentic-capabilities) that highlight the model's agentic capabilities.

If you want to jump right in, you can try out the model in Google AI Studio.

✦ **Try in Google AI Studio**
(https://aistudio.google.com/prompts/new_chat?model=gemini-robotics-er-1.5-preview)

Safety

While Gemini Robotics-ER 1.5 was built with safety in mind, it is your responsibility to maintain a safe environment around the robot. Generative AI models can make mistakes, and physical robots can cause damage. Safety is a priority, and making generative AI models safe when

used with real-world robotics is an active and critical area of our research. To learn more, visit the Google DeepMind robotics safety page (https://deepmind.google/models/gemini-robotics/safety).

Getting started: Finding objects in a scene

The following example demonstrates a common robotics use case. It shows how to pass an image and a text prompt to the model using the `generateContent` (/api/generate-content#method:-models.generatecontent) method to get a list of identified objects with their corresponding 2D points. The model returns points for items it identified in an image, returning their normalized 2D coordinates and labels.

You can use this output with a robotics API or call a vision-language-action (VLA) model or any other third-party user-defined functions to generate actions for a robot to perform.

Python REST (#rest)
(#python)

```
from google import genai
from google.genai import types

# Initialize the GenAI client and specify the model
MODEL_ID = "gemini-robotics-er-1.5-preview"
PROMPT = """
          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.
        """
client = genai.Client(api_key=YOUR_API_KEY)

# Load your image
with open("my-image.png", 'rb') as f:
    image_bytes = f.read()

image_response = client.models.generate_content(
    model=MODEL_ID,
    contents=[
        types.Part.from_bytes(
            data=image_bytes,
```

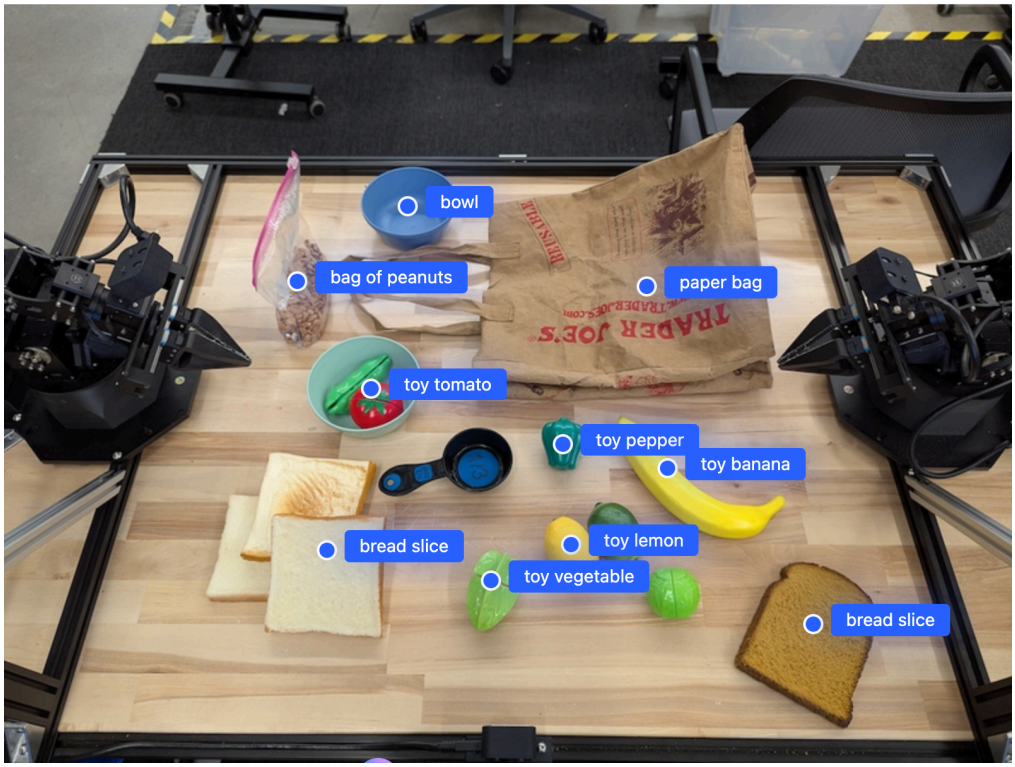
```
            mime_type='image/png',
        ),
        PROMPT
    ],
    config = types.GenerateContentConfig(
        temperature=0.5,
        thinking_config=types.ThinkingConfig(thinking_budget=0)
    )
)

print(image_response.text)
```

The output will be a JSON array containing objects, each with a `point` (normalized `[y, x]` coordinates) and a `label` identifying the object.

JSON (#json)

```
[
  {"point": [376, 508], "label": "small banana"},
  {"point": [287, 609], "label": "larger banana"},
  {"point": [223, 303], "label": "pink starfruit"},
  {"point": [435, 172], "label": "paper bag"},
  {"point": [270, 786], "label": "green plastic bowl"},
  {"point": [488, 775], "label": "metal measuring cup"},
  {"point": [673, 580], "label": "dark blue bowl"},
  {"point": [471, 353], "label": "light blue bowl"},
  {"point": [492, 497], "label": "bread"},
  {"point": [525, 429], "label": "lime"}
]
```

The following image is an example of how these points can be displayed:



How it works

Gemini Robotics-ER 1.5 allows your robots to contextualize and work in the physical world using spatial understanding. It takes image/video/audio input and natural language prompts to:

- **Understand objects and scene context**: Identifies objects, and reasons about their relationship to the scene, including their affordances.
- **Understand task instructions**: Interprets tasks given in natural language, like "find the banana".
- **Reason spatially and temporally**: Understand sequences of actions and how objects interact with a scene over time.
- **Provide structured output**: Returns coordinates (points or bounding boxes) representing object locations.

This enables robots to "see" and "understand" their environment programmatically.

Gemini Robotics-ER 1.5 is also agentic, which means it can break down complex tasks (like "put the apple in the bowl") into sub-tasks to orchestrate long horizon tasks:

- **Sequencing subtasks**: Decomposes commands into a logical sequence of steps.
- **Function calls/Code execution**: Executes steps by calling your existing robot functions/tools or executing generated code.

Read more about how function calling with Gemini works on the Function Calling page (/gemini-api/docs/function-calling?example=meeting#how-it-works).

Using the thinking budget with Gemini Robotics-ER 1.5

Gemini Robotics-ER 1.5 has a flexible thinking budget that gives you control over latency versus accuracy tradeoffs. For spatial understanding tasks like object detection, the model can achieve high performance with a small thinking budget. More complex reasoning tasks like counting and weight estimation benefit from a larger thinking budget. This lets you balance the need for low-latency responses with high-accuracy results for more challenging tasks.

To learn more about thinking budgets, see the Thinking (/gemini-api/docs/thinking) core capabilities page.

Agentic capabilities for robotics

This section walks through various capabilities of Gemini Robotics-ER 1.5, demonstrating how to use the model for robotic perception, reasoning, and planning applications.

The examples in this section demonstrate capabilities from pointing and finding objects in an image to planning trajectories and orchestrating long horizon tasks. For simplicity, the code snippets have been reduced to show the prompt and the call to `generate_content` API. The full runnable code as well as additional examples can be found in the Robotics cookbook (https://github.com/google-gemini/cookbook/blob/main/quickstarts/gemini-robotics-er.ipynb).

Pointing to objects

Pointing and finding objects in images or video frames is a common use case for vision-and-language models (VLMs) in robotics. The following example asks the model to find specific objects within an image and return their coordinates in an image.

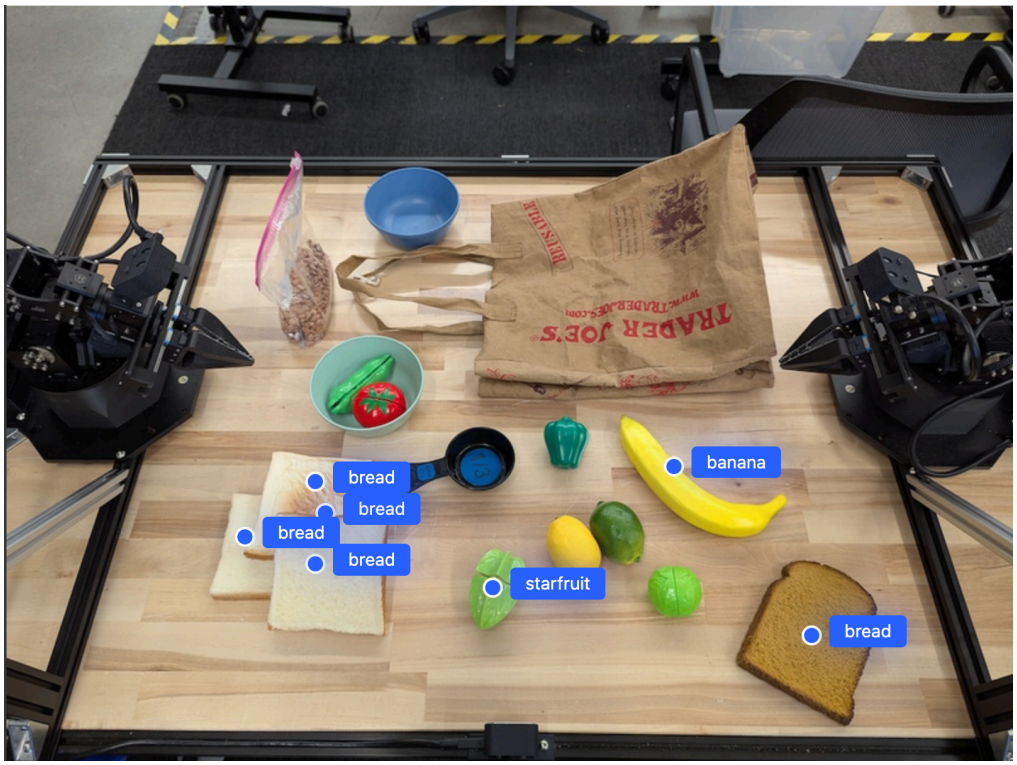
Python (#python)

```
from google import genai
from google.genai import types

# Initialize the GenAI client and specify the model
MODEL_ID = "gemini-robotics-er-1.5-preview"
client = genai.Client(api_key=YOUR_API_KEY)

# Load your image and set up your prompt
with open('path/to/image-with-objects.jpg', 'rb') as f:
    image_bytes = f.read()

queries = [
    "bread",
    "starfruit",
    "banana",
]

prompt = f"""
    Get all points matching the following objects: {', '.join(queries)}. Tl
    label returned should be an identifying name for the object detected.
    The answer should follow the json format:

    [{{"point": , "label": }}, ...]. The points are in

    [y, x] format normalized to 0-1000.
    """

image_response = client.models.generate_content(
  model=MODEL_ID,
  contents=[
    types.Part.from_bytes(
      data=image_bytes,
      mime_type='image/jpeg',
    ),
    prompt
  ],
  config = types.GenerateContentConfig(
      temperature=0.5,
      thinking_config=types.ThinkingConfig(thinking_budget=0)
  )
)
```

```
  print(image_response.text)
```

The output would be similar to the getting started example, a JSON containing the coordinates of the objects found and their labels.


```
[
  {"point": [671, 317], "label": "bread"},
  {"point": [738, 307], "label": "bread"},
  {"point": [702, 237], "label": "bread"},
  {"point": [629, 307], "label": "bread"},
  {"point": [833, 800], "label": "bread"},
  {"point": [609, 663], "label": "banana"},
  {"point": [770, 483], "label": "starfruit"}
]
```

Use the following prompt to request the model to interpret abstract categories like "fruit" instead of specific objects and locate all instances in the image.

Python
(#python)

```
prompt = f"""
        Get all points for fruit. The label returned should be an identify
        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."""
```

Visit the image understanding (/gemini-api/docs/image-understanding) page for other image processing techniques.

Tracking objects in a video

Gemini Robotics-ER 1.5 can also analyze video frames to track objects over time. See Video inputs (/gemini-api/docs/video-understanding#supported-formats) for a list of supported video formats.

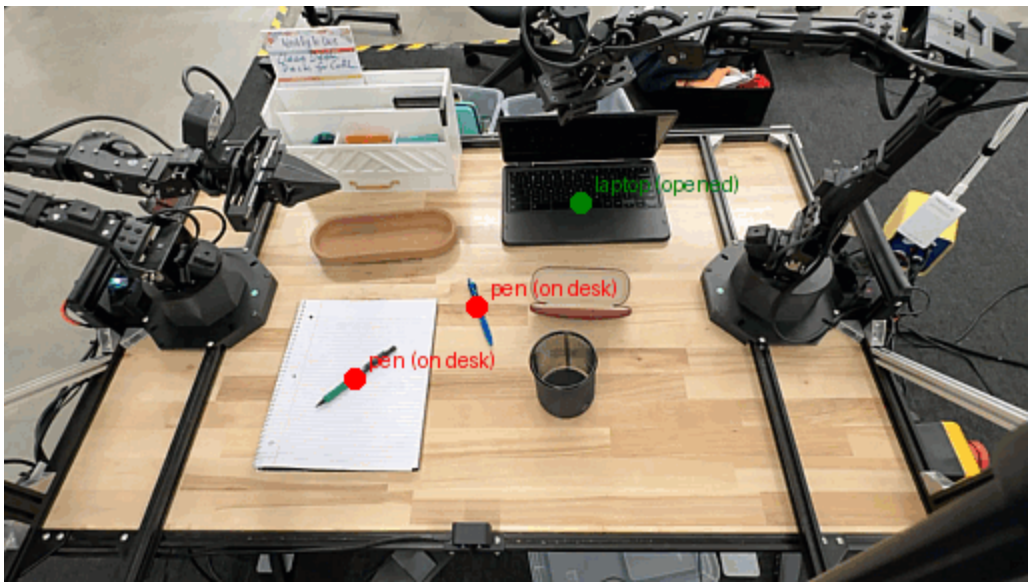
The following is the base prompt used to find specific objects in each frame that the model analyzes:

Python (#python)

```
# Define the objects to find
queries = [
    "pen (on desk)",
    "pen (in robot hand)",
    "laptop (opened)",
    "laptop (closed)",
]

base_prompt = f"""
  Point to the following objects in the provided image: {', '.join(queries
  The answer should follow the json format:

  [{{"point": , "label": }}, ...].

  The points are in [y, x] format normalized to 0-1000.
  If no objects are found, return an empty JSON list [].
```

```
    """
```

The output shows a pen and laptop being tracked across the video frames.



For the full runnable code, see the Robotics cookbook (https://github.com/google-gemini/cookbook/blob/main/quickstarts/gemini-robotics-er.ipynb).

Object detection and bounding boxes

Beyond single points, the model can also return 2D bounding boxes, providing a rectangular region enclosing an object.

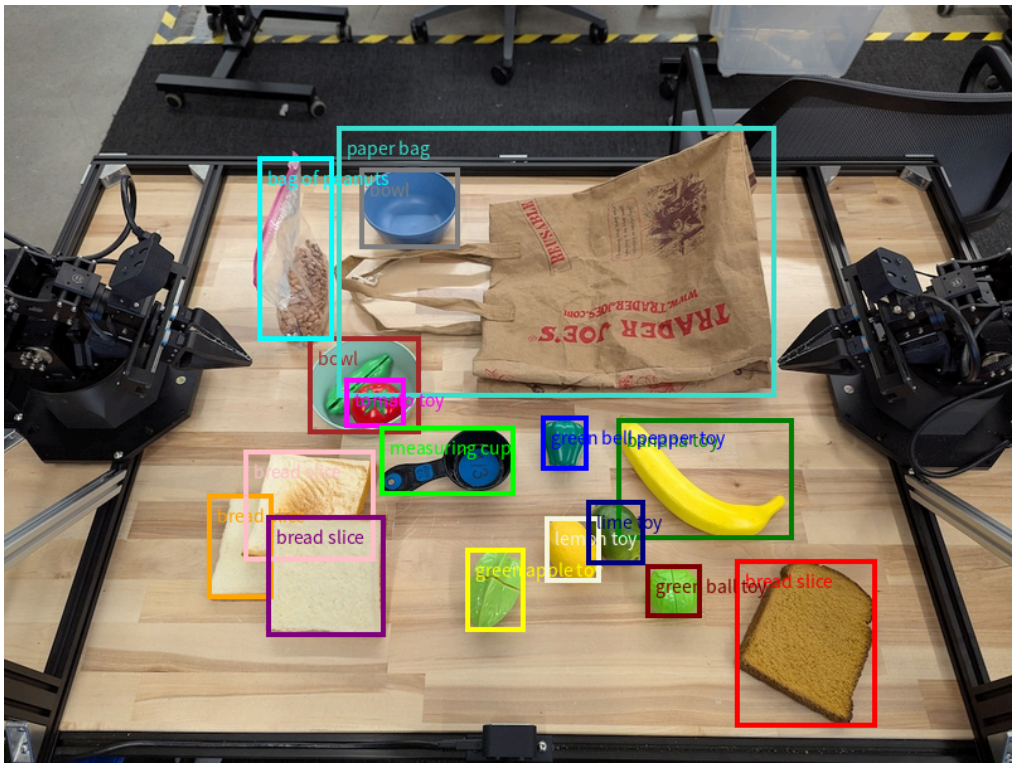
This example requests 2D bounding boxes for identifiable objects on a table. The model is instructed to limit the output to 25 objects and to name multiple instances uniquely.

Python (#python)

```
from google import genai
from google.genai import types

# Initialize the GenAI client and specify the model
MODEL_ID = "gemini-robotics-er-1.5-preview"
client = genai.Client(api_key=YOUR_API_KEY)

# Load your image and set up your prompt
```

```
with open('path/to/image-with-objects.jpg', 'rb') as f:
    image_bytes = f.read()

prompt = """
      Return bounding boxes as a JSON array with labels. Never return mask
      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 characteristic
      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
      """

image_response = client.models.generate_content(
  model=MODEL_ID,
  contents=[
    types.Part.from_bytes(
      data=image_bytes,
      mime_type='image/jpeg',
    ),
    prompt
  ],
  config = types.GenerateContentConfig(
      temperature=0.5,
      thinking_config=types.ThinkingConfig(thinking_budget=0)
  )
)

print(image_response.text)
```

The following displays the boxes returned from the model.



For the full runnable code, see the Robotics cookbook (https://github.com/google-gemini/cookbook/blob/main/quickstarts/gemini-robotics-er.ipynb). The Image understanding (/gemini-api/docs/image-understanding) page also has additional examples of visual tasks like segmentation and object detection.

Additional bounding box examples can be found in the Image understanding (/gemini-api/docs/image-understanding) page.

Trajectories

Gemini Robotics-ER 1.5 can generate sequences of points that define a trajectory, useful for guiding robot movement.

This example requests a trajectory to move a red pen to an organizer, including the starting point and a series of intermediate points.

Python (#python)

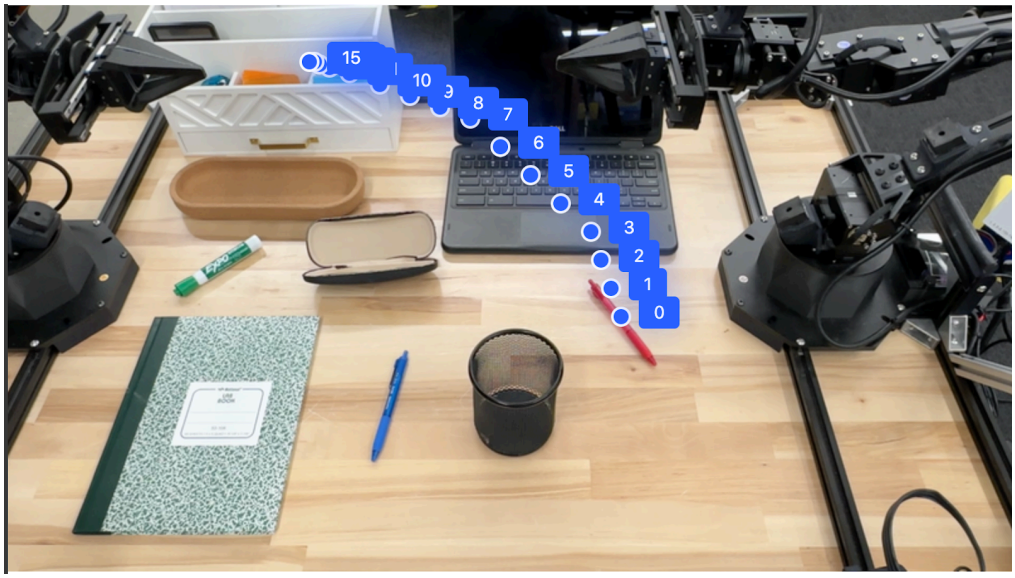
```
from google import genai
from google.genai import types
```

```
# Initialize the GenAI client and specify the model
MODEL_ID = "gemini-robotics-er-1.5-preview"
client = genai.Client(api_key=YOUR_API_KEY)

# Load your image and set up your prompt
with open('path/to/image-with-objects.jpg', 'rb') as f:
    image_bytes = f.read()

points_data = []
prompt = """
        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.
        """

image_response = client.models.generate_content(
  model=MODEL_ID,
  contents=[
    types.Part.from_bytes(
      data=image_bytes,
      mime_type='image/jpeg',
    ),
    prompt
  ],
  config = types.GenerateContentConfig(
      temperature=0.5,
  )
)

print(image_response.text)
```

The response is a set of coordinates that describe the trajectory of the path that the red pen should follow to complete the task of moving it on top of the organizer:

```
[
  {"point": [550, 610], "label": "0"},
  {"point": [500, 600], "label": "1"},
  {"point": [450, 590], "label": "2"},
```

```
  {"point": [400, 580], "label": "3"},
  {"point": [350, 550], "label": "4"},
  {"point": [300, 520], "label": "5"},
  {"point": [250, 490], "label": "6"},
  {"point": [200, 460], "label": "7"},
  {"point": [180, 430], "label": "8"},
  {"point": [160, 400], "label": "9"},
  {"point": [140, 370], "label": "10"},
  {"point": [120, 340], "label": "11"},
  {"point": [110, 320], "label": "12"},
  {"point": [105, 310], "label": "13"},
  {"point": [100, 305], "label": "14"},
  {"point": [100, 300], "label": "15"}
]
```



Orchestration

Gemini Robotics-ER 1.5 can perform higher-level spatial reasoning, inferring actions or identifying optimal locations based on contextual understanding.

Making room for a laptop

This example shows how Gemini Robotics-ER can reason about a space. The prompt asks the model to identify which object needs to be moved to create space for another item.

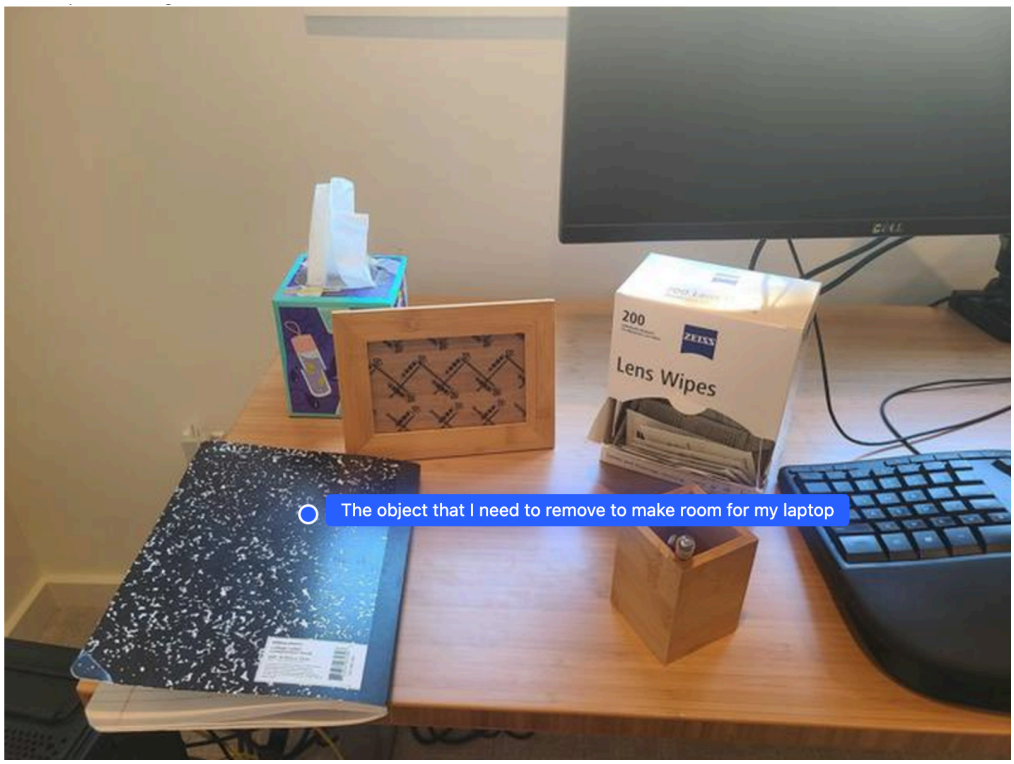
Python
(#python)

```
from google import genai
from google.genai import types

# Initialize the GenAI client and specify the model
MODEL_ID = "gemini-robotics-er-1.5-preview"
client = genai.Client(api_key=YOUR_API_KEY)

# Load your image and set up your prompt
with open('path/to/image-with-objects.jpg', 'rb') as f:
    image_bytes = f.read()

prompt = """
          Point to the object that I need to remove to make room for my la
          The answer should follow the json format: [{"point": <point>,
          "label": <label1>}, ...]. The points are in [y, x] format normal
        """

image_response = client.models.generate_content(
  model=MODEL_ID,
  contents=[
    types.Part.from_bytes(
      data=image_bytes,
      mime_type='image/jpeg',
    ),
    prompt
  ],
  config = types.GenerateContentConfig(
      temperature=0.5,
      thinking_config=types.ThinkingConfig(thinking_budget=0)
  )
)

print(image_response.text)
```

The response contains a 2D coordinate of the object that answers the user's question, in this case, the object that should move to make room for a laptop.

```
[
  {"point": [672, 301], "label": "The object that I need to remove to make room
```

]



Packing a Lunch

The model can also provide instructions for multi-step tasks and point to relevant objects for each step. This example shows how the model plans a series of steps to pack a lunch bag.

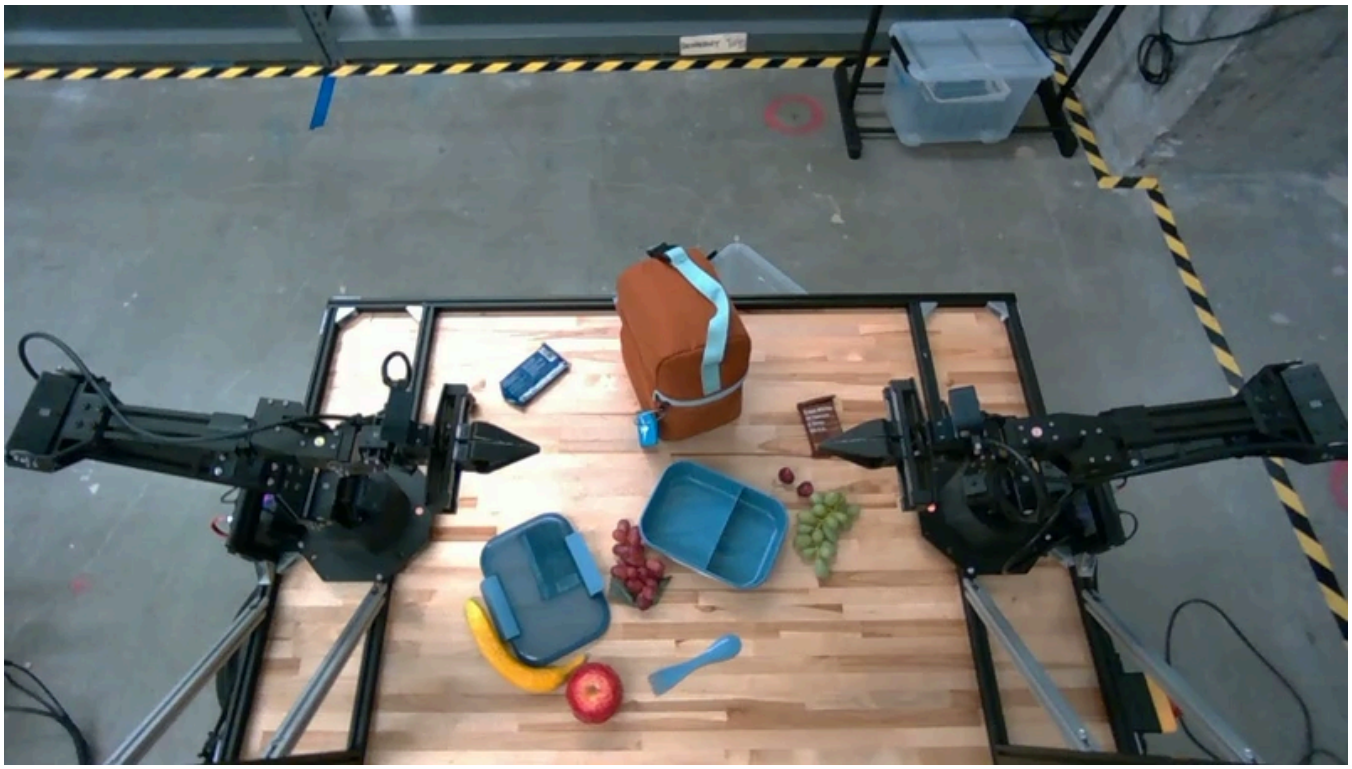
Python
(#python)

```
from google import genai
from google.genai import types

# Initialize the GenAI client and specify the model
MODEL_ID = "gemini-robotics-er-1.5-preview"
client = genai.Client(api_key=YOUR_API_KEY)

# Load your image and set up your prompt
with open('path/to/image-of-lunch.jpg', 'rb') as f:
    image_bytes = f.read()

prompt = """
```

```
          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.
        """

image_response = client.models.generate_content(
  model=MODEL_ID,
  contents=[
    types.Part.from_bytes(
      data=image_bytes,
      mime_type='image/jpeg',
    ),
    prompt
  ],
  config = types.GenerateContentConfig(
      temperature=0.5,
      thinking_config=types.ThinkingConfig(thinking_budget=0)
  )
)

print(image_response.text)
```

The response of this prompt is a set of step by step instructions on how to pack a lunch bag from the image input.

Input image

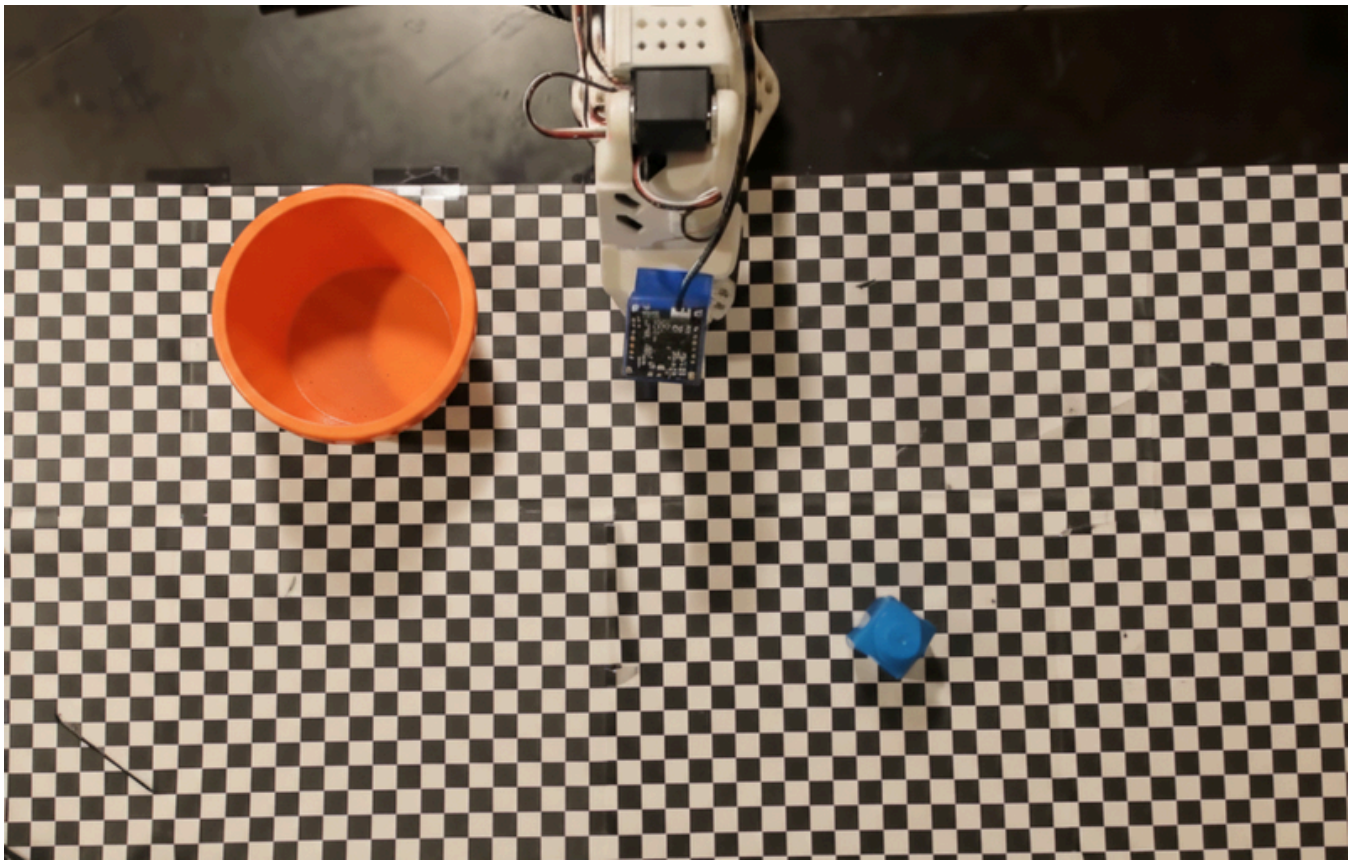


Model output

```
Based on the image, here is a plan to pack the lunch box and lunch bag:

1.  **Pack the fruit into the lunch box.** Place the [apple](apple), [banana](ba
2.  **Add the spoon to the lunch box.** Put the [blue spoon](blue spoon) inside
3.  **Close the lunch box.** Secure the lid on the [blue lunch box](blue lunch b
4.  **Place the lunch box inside the lunch bag.** Put the closed [blue lunch box
5.  **Pack the remaining items into the lunch bag.** Place the [blue snack bar](

Here is the list of objects and their locations:
*   [{"point": [899, 440], "label": "apple"}]
*   [{"point": [814, 363], "label": "banana"}]
*   [{"point": [727, 470], "label": "red grapes"}]
*   [{"point": [675, 608], "label": "green grapes"}]
*   [{"point": [706, 529], "label": "blue lunch box"}]
*   [{"point": [864, 517], "label": "blue spoon"}]
*   [{"point": [499, 401], "label": "blue snack bar"}]
*   [{"point": [614, 705], "label": "brown snack bar"}]
*   [{"point": [448, 501], "label": "brown lunch bag"}]
```

Calling a custom robot API

This example demonstrates task orchestration with a custom robot API. It introduces a mock API designed for a pick-and-place operation. The task is to pick up a blue block and place it in an orange bowl:



Similar to the other examples on this page, the full runnable code is available in the Robotics cookbook (https://github.com/google-gemini/cookbook/blob/main/quickstarts/gemini-robotics-er.ipynb).

First step is to locate both of the items with the following prompt:

Python
(#python)

```
prompt = """
            Locate and point to the blue block and the orange bowl. The lal
            returned should be an identifying name for the object detected
            The answer should follow the json format: [{"point": <point>,
            The points are in [y, x] format normalized to 0-1000.
          """
```

The model response includes the normalized coordinates of the block and the bowl:

```
[
  {"point": [389, 252], "label": "orange bowl"},
  {"point": [727, 659], "label": "blue block"}
]
```

This example uses the following mock robot API:

Python
(#python)

```
def move(x, y, high):
  print(f"moving to coordinates: {x}, {y}, {15 if high else 5}")

def setGripperState(opened):
  print("Opening gripper" if opened else "Closing gripper")

def returnToOrigin():
  print("Returning to origin pose")
```

The next step is calling a sequence of API functions with the necessary logic to execute the action. The following prompt includes a description of the robot API that the model should use when orchestrating this task.

Python
(#python)

```
prompt = 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' s
      to True means the robot arm should be lifted above the scene for
      avoiding obstacles during motion. 'high' set to False means the robo
      arm should have the gripper placed on the surface for interacting wi
      objects.

    def setGripperState(opened):
      # Opens the gripper if opened set to true, otherwise closes the grip
```

```
    def returnToOrigin():
      # Returns the robot to an initial state. Should be called as a clean
      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.
    For example:
    Reasoning: To pick up the block, I will first move the arm to a high
    position above the block, open the gripper, move down to the block,
    close the gripper, lift the arm, move to a high position above the bowl,
    move down to the bowl, open the gripper, and then lift the arm back to
    a high position.
"""
```

The following shows a possible output of the model based on the prompt and the mock robot API. The output includes the model thinking process and the tasks that it planned out as a result. It also shows the output of the robot function calls that the model sequenced together.

```
Reasoning: To perform the pick and place operation, I will first move the
robotic arm to a high position above the blue block to ensure clearance. Then, I
will open the gripper in preparation for grasping. Next, I will lower the arm to
the surface level to position the gripper around the block and close the gripper
to secure it. After picking up the block, I will lift the arm to clear the
surface and move towards the orange bowl, maintaining a high position to avoid
collisions. Once positioned above the bowl, I will lower the arm to place the
block inside. I will then open the gripper to release the block. Finally, I will
lift the arm out of the bowl and call `returnToOrigin` to reset the robot's
state.

[
  {
```

```
    "function": "move",
    "args": [
      163,
      427,
      true
    ]
  },
  {
    "function": "setGripperState",
    "args": [
      true
    ]
  },
  {
    "function": "move",
    "args": [
      163,
      427,
      false
    ]
  },
  {
    "function": "setGripperState",
    "args": [
      false
    ]
  },
  {
    "function": "move",
    "args": [
      163,
      427,
      true
    ]
  },
  {
    "function": "move",
    "args": [
      -247,
      90,
      true
    ]
  },
  {
    "function": "move",
```

```
    "args": [
      -247,
      90,
      false
    ]
  },
  {
    "function": "setGripperState",
    "args": [
      true
    ]
  },
  {
    "function": "move",
    "args": [
      -247,
      90,
      true
    ]
  },
  {
    "function": "returnToOrigin",
    "args": []
  }
]
```

```
Executing Function Calls:
moving to coordinates: 163, 427, 15
Opening gripper
moving to coordinates: 163, 427, 5
Closing gripper
moving to coordinates: 163, 427, 15
moving to coordinates: -247, 90, 15
moving to coordinates: -247, 90, 5
Opening gripper
moving to coordinates: -247, 90, 15
Returning to origin pose
```

Code execution

Gemini Robotics-ER 1.5 can suggest and execute Python code to perform tasks that require dynamic actions, such as zooming into an image region for better detail.

This example demonstrates how the model can suggest using the code execution (/gemini-api/docs/code-execution) tool to "zoom in" on a specific area of an image, which it then carries out to answer the user's question.

Python (#python)

```
from google import genai
from google.genai import types

client = genai.Client(api_key=YOUR_API_KEY)

MODEL_ID = "gemini-robotics-er-1.5-preview"

# Load your image and set up your prompt
with open('path/to/image-of-object.jpg', 'rb') as f:
    image_bytes = f.read()
prompt = """
          What is the air quality reading? Using the code execution feature
          zoom in on the image to take a closer look.
        """

response = client.models.generate_content(
    model=MODEL_ID,
    contents=[
        types.Part.from_bytes(
            data=image_bytes,
            mime_type='image/jpeg',
        ),
        prompt
    ],
    config = types.GenerateContentConfig(
        temperature=0.5,
        tools=[types.Tool(code_execution=types.ToolCodeExecution)]
    )
)

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)
```

The following shows a possible model response for this task. The output shows the model generating code for zooming into the image to read the screen better. It also shows the thought process around the adjustments it made to the cropping. For the complete sample output, see the Robotics cookbook (https://github.com/google-gemini/cookbook/blob/main/quickstarts/gemini-robotics-er.ipynb).

```
# Detect relevant objects and output in json format
# [{"box_2d": [379, 407, 481, 553], "label": "display"}]
from PIL import Image

# Open the image
img = Image.open("input_file_0.jpeg")

# Define the coordinates for cropping (normalized to 0-1000)
# Based on visual inspection, the display is roughly centered around [430, 430]
# Let's try to zoom in on the display area.
# Bounding box for display: [ymin, xmin, ymax, xmax] in pixels.
# From visual inspection of the original image (size 90489 bytes, resolution
unknown but let's assume a standard aspect ratio and roughly 1000x1000 for
relative positioning),
# the display is located at approximately y=380-480, x=410-540.
# Let's define a larger area around it for context.
# Let's say we want to crop from y=350 to y=550 and x=350 to x=650.

...

thought
The crop was too tight and only captured part of the display. Let's adjust the b
The previous crop was `img.crop((400, 350, 600, 500))`. This means xmin=400, ymi
The display is centered around x=480, y=430.
The previous crop captured the left side of the display.
Let's expand the x range to the right and y range to cover the whole circle.
New bounding box estimate: ymin=370, xmin=400, ymax=500, xmax=600.
Let's try again with a wider crop.
# Detect relevant objects and output in json format
# [{"box_2d": [379, 407, 481, 553], "label": "display"}]
from PIL import Image

# Open the image
```

```
img = Image.open("input_file_0.jpeg")

# Define the coordinates for cropping (normalized to 0-1000)
…

The reading of 14 falls into the "Moderate" category.
The prompt asks for the "air quality reading", which can be interpreted as
either the numerical value or the categorical description. Given the numerical
value is explicitly displayed, it's the primary reading.

Let's provide the numerical value as the reading.
The reading is 014.
Based on the display on the air purifier, the air quality reading is **014**.
```

Best practices

To optimize the performance and accuracy of your robotics applications, it's crucial to understand how to interact with the Gemini model effectively. This section outlines best practices and key strategies for crafting prompts, handling visual data, and structuring tasks to achieve the most reliable results.

1. Use clear and simple language.
    - **Embrace natural language**: The Gemini model is designed to comprehend natural, conversational language. Structure your prompts in a way that is semantically clear and mirrors how a person would naturally give instructions.
    - **Use everyday terminology**: Opt for common, everyday language over technical or specialized jargon. If the model is not responding as expected to a particular term, try rephrasing it with a more common synonym.
2. Optimize the visual input.
    - **Zoom in for detail**: When dealing with objects that are small or difficult to discern in a wider shot, use a bounding box function to isolate the object of interest. You can then crop the image to this selection and send the new, focused image to the model for a more detailed analysis.
    - **Experiment with lighting and color**: The model's perception can be affected by challenging lighting conditions and poor color contrast.

3. Break down complex problems into smaller steps. By addressing each smaller step individually, you can guide the model to a more precise and successful outcome.
4. Improve accuracy through consensus. For tasks that require a high degree of precision, you can query the model multiple times with the same prompt. By averaging the returned results, you can arrive at a "consensus" that is often more accurate and reliable.

Limitations

Consider the following limitations when developing with Gemini Robotics-ER 1.5:

- **Preview status:** The model is currently in **Preview**. APIs and capabilities may change, and it may not be suitable for production-critical applications without thorough testing.
- **Latency:** Complex queries, high-resolution inputs, or extensive `thinking_budget` can lead to increased processing times.
- **Hallucinations:** Like all large language models, Gemini Robotics-ER 1.5 can occasionally "hallucinate" or provide incorrect information, especially for ambiguous prompts or out-of-distribution inputs.
- **Dependence on prompt quality:** The quality of the model's output is highly dependent on the clarity and specificity of the input prompt. Vague or poorly structured prompts can lead to suboptimal results.
- **Computational cost:** Running the model, especially with video inputs or high `thinking_budget`, consumes computational resources and incurs costs. See the Thinking (/gemini-api/docs/thinking) page for more details.
- **Input types:** See the following topics for details on limitations for each mode.
    - Image inputs (/gemini-api/docs/image-understanding#technical-details-image)
    - Video inputs (/gemini-api/docs/video-understanding#supported-formats)
    - Audio inputs (/gemini-api/docs/audio#supported-formats)

Privacy Notice

You acknowledge that the models referenced in this document (the "Robotics Models") leverage video and audio data in order to operate and move your hardware in accordance with your instructions. You therefore may operate the Robotics Models such that data from identifiable persons, such as voice, imagery, and likeness data ("Personal Data"), will be collected by the Robotics Models. If you elect to operate the Robotics Models in a manner that collects Personal Data, you agree that you will not permit any identifiable persons to interact with, or be present in the area surrounding, the Robotics Models, unless and until such identifiable persons have been sufficiently notified of and consented to the fact that their Personal Data may be provided to and used by Google as outlined in the Gemini API Additional Terms of Service found at https://ai.google.dev/gemini-api/terms (https://ai.google.dev/gemini-api/terms) (the "Terms"), including in accordance with the section entitled "How Google Uses Your Data". You will ensure that such notice permits the collection and use of Personal Data as outlined in the Terms, and you will use commercially reasonable efforts to minimize the collection and distribution of Personal Data by using techniques such as face blurring and operating the Robotics Models in areas not containing identifiable persons to the extent practicable.

Pricing

For detailed information on pricing and available regions, refer to the pricing (/gemini-api/docs/pricing) page.

Model versions

Property	Description	
Model code	`gemini-robotics-er-1.5-preview`	
Supported data types	**Inputs** Text, images, video, audio	**Output** Text

Token limits[*] (/gemini-api/docs/tokens)	**Input token limit** 1,048,576	**Output token limit** 65,536
Capabilities	**Audio generation** Not supported	**Batch API** Not supported
	Caching Not supported	**Code execution** Supported
	Function calling Supported	**Grounding with Google Maps** Not supported
	Image generation Not supported	**Live API** Not supported
	Search grounding Supported	**Structured outputs** Supported
	Thinking Supported	**URL context** Supported
Versions	Read the model version patterns (/gemini-api/docs/models/gemini#model-versions) for more details. • Preview: `gemini-robotics-er-1.5-preview`	
Latest update	September 2025	
Knowledge	January 2025	

cutoff

Next steps

- Explore other capabilities and continue experimenting with different prompts and inputs to discover more applications for Gemini Robotics-ER 1.5. See the Robotics cookbook (https://github.com/google-gemini/cookbook/blob/main/quickstarts/gemini-robotics-er.ipynb) for more examples.
- Learn about how Gemini Robotics models were built with safety in mind, visit the Google DeepMind robotics safety page (https://deepmind.google/models/gemini-robotics/safety).
- Read about the latest updates on Gemini Robotics models on the Gemini Robotics landing page (https://deepmind.google/robotics).

Last updated 2025-10-24 UTC.