Connecting Generative AI and Robotics

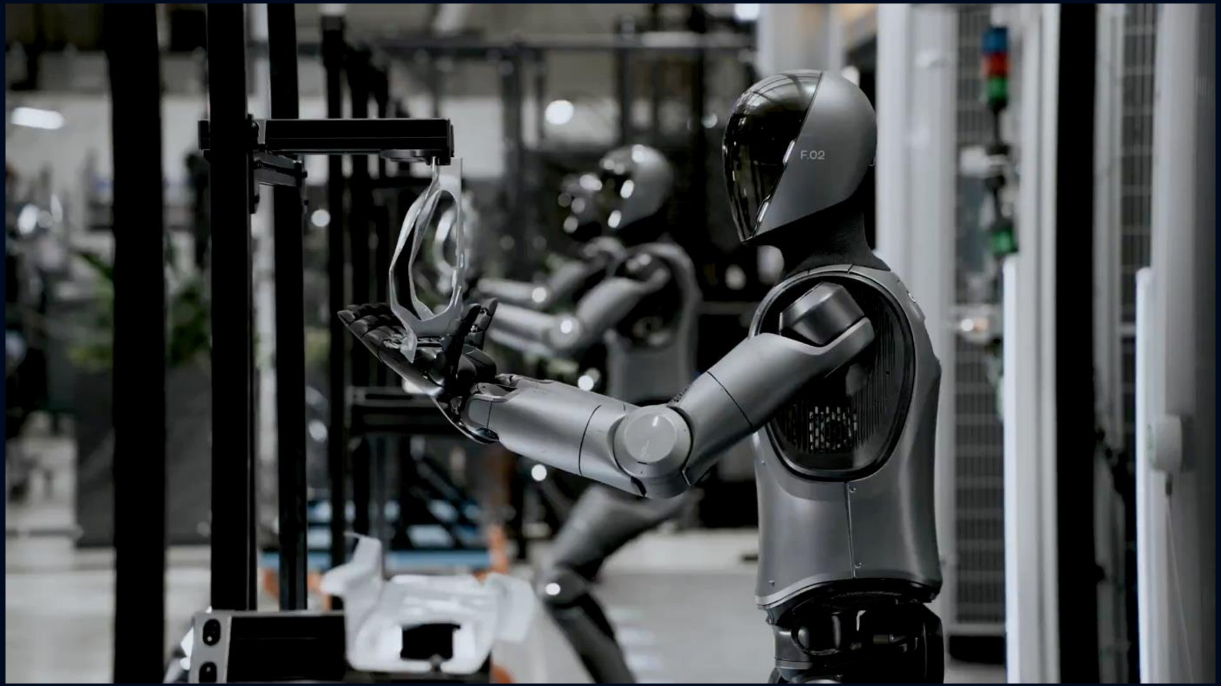
Edward Johns

The Robot Learning Lab
at
Imperial College London

22nd November 2024

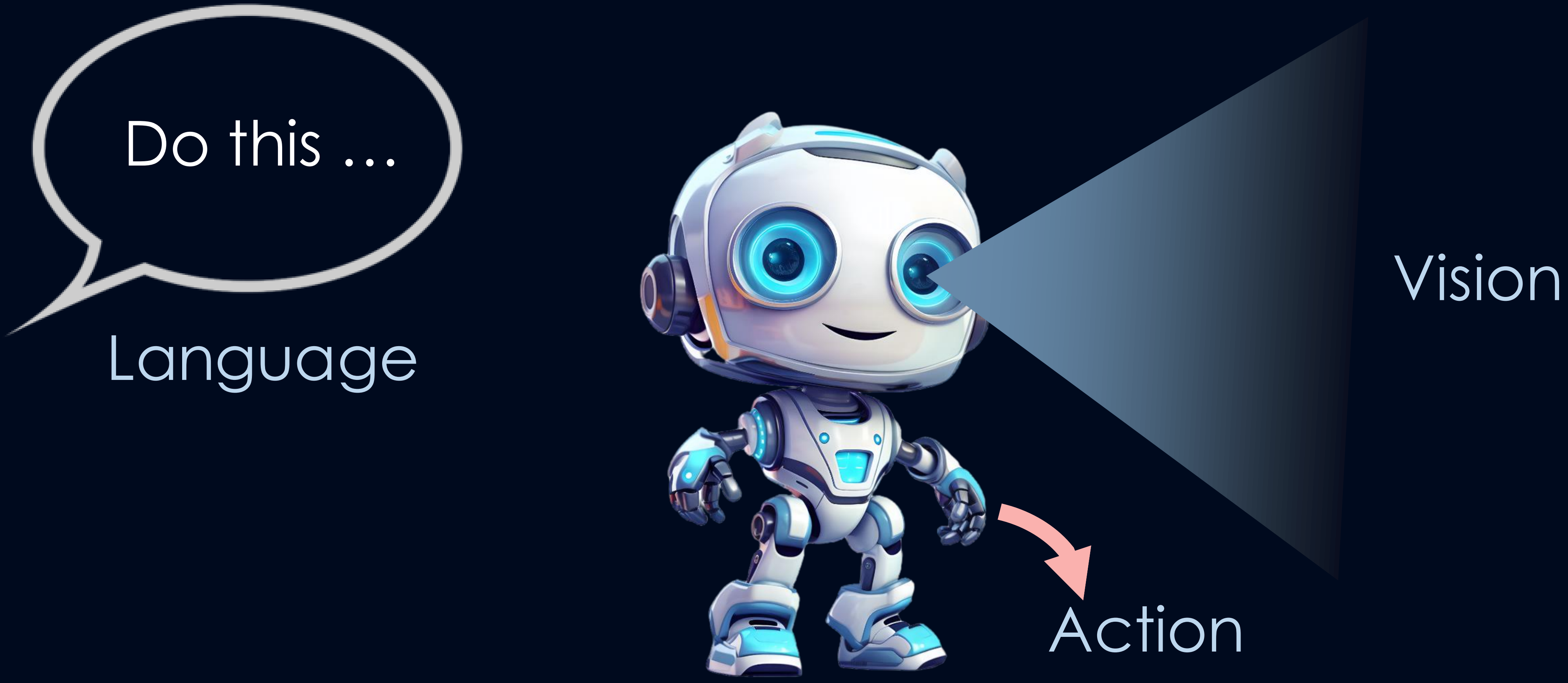
Physical Intelligence
October 31st 2024

Figure
November 19th 2024



T. Zhao et al., 2023

Vision & Language & Action



Think of a Random Task …

Think of a Random Task …

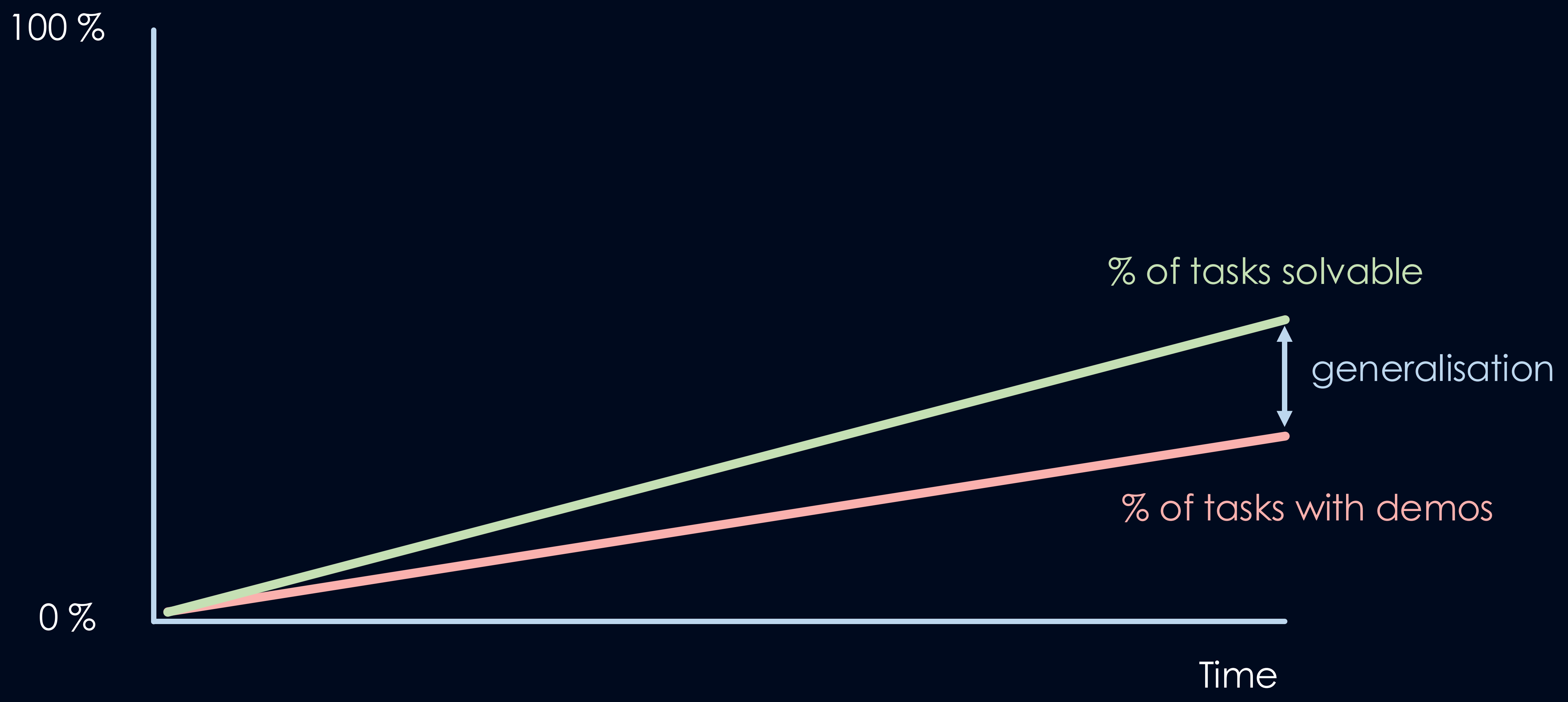
Teltonika Electronics Manufacturing Services
2021

Kuangwu (Foshan) Audio Equipment Co.
2024

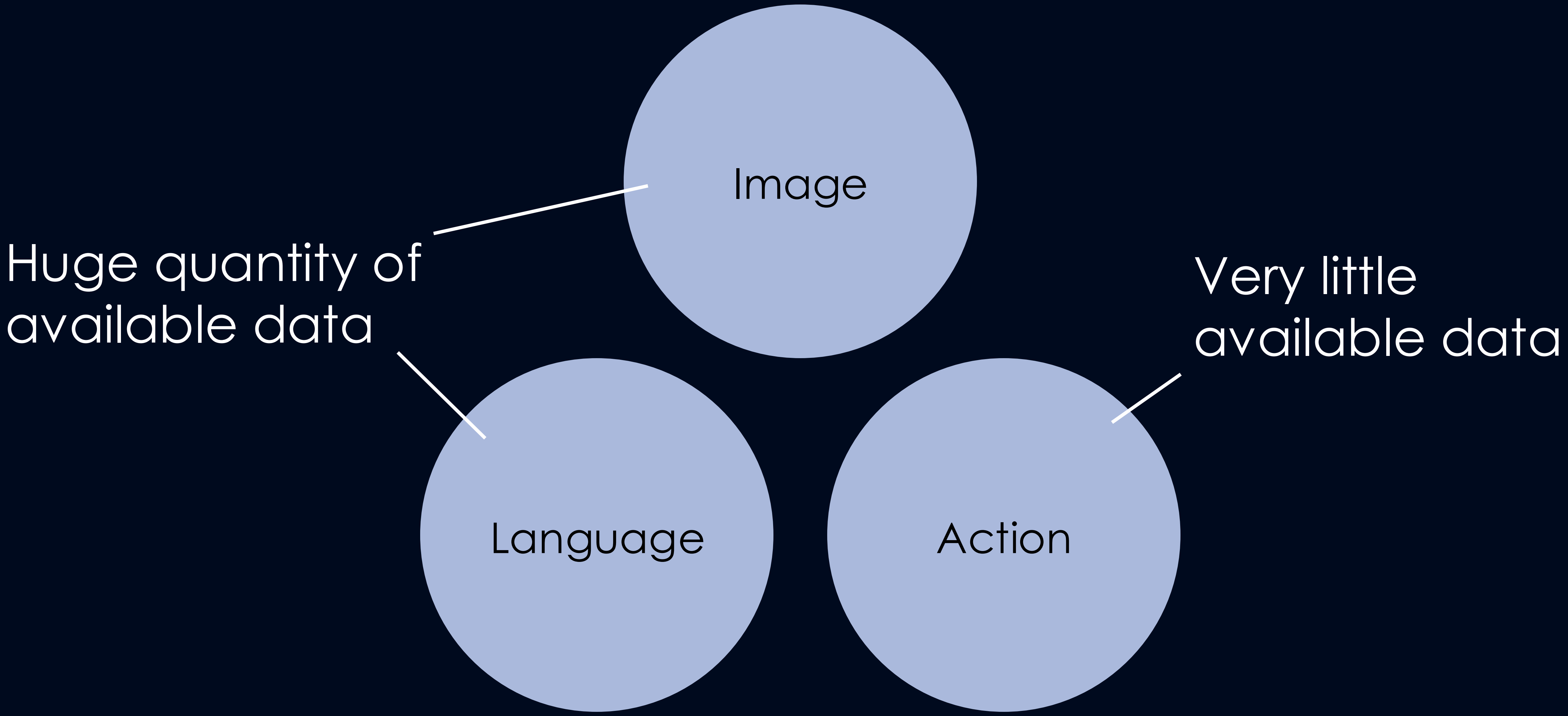
We're Going to Need a Lot of Demonstrations…

Object Instances X Object Poses X Tasks X Environments X Robot Embodiments

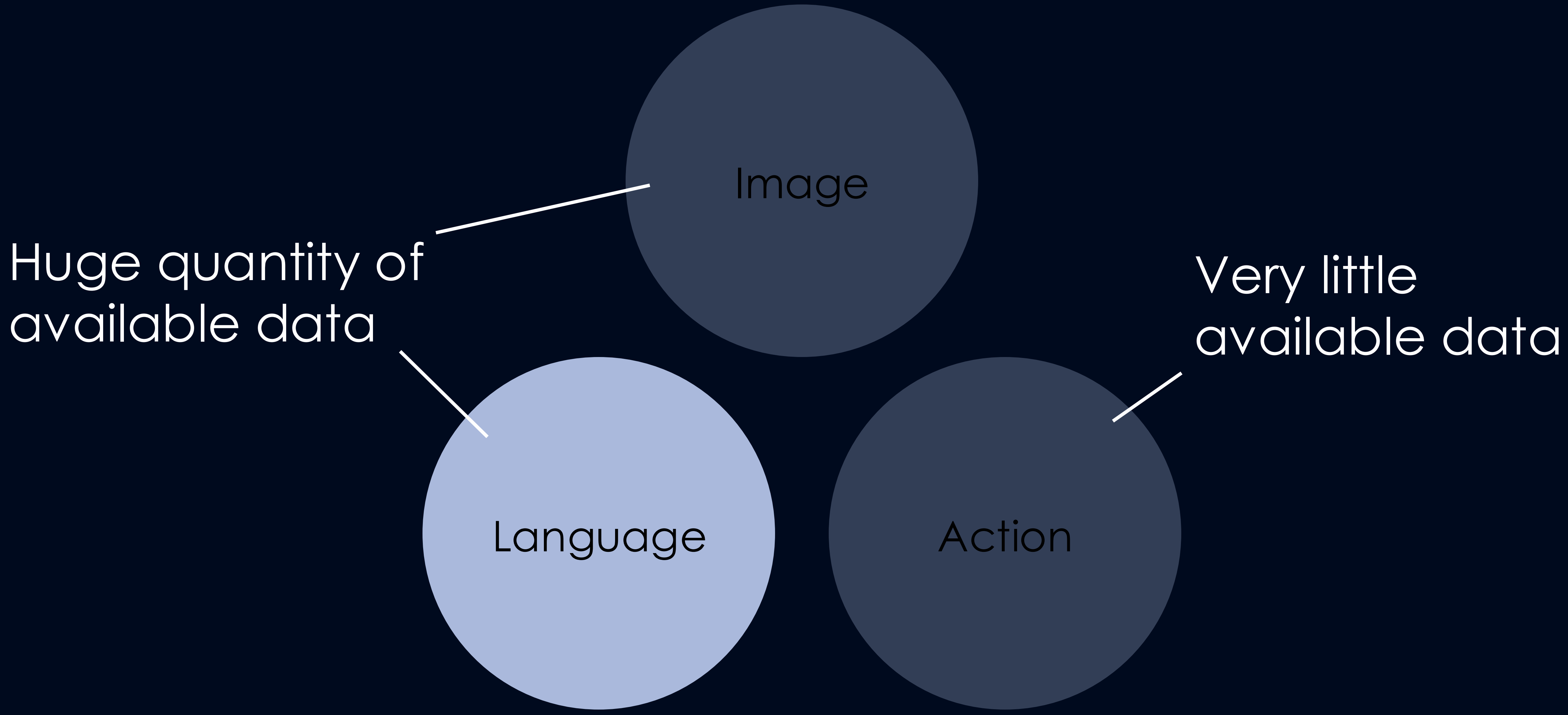
Looking Ahead



Vision & Language & Action



Vision & Language & Action



Can Large Language Models Alone Solve Robotics Tasks?

Can Large Language Models Alone Solve Robotics Tasks?

" Wipe the plate with the sponge "

Kwon, Di Palo, and Johns, "Language Models as Zero-Shot Trajectory Generators", RA-Letters 2024

Can Large Language Models Alone Solve Robotics Tasks?

" Wipe the plate with the sponge "

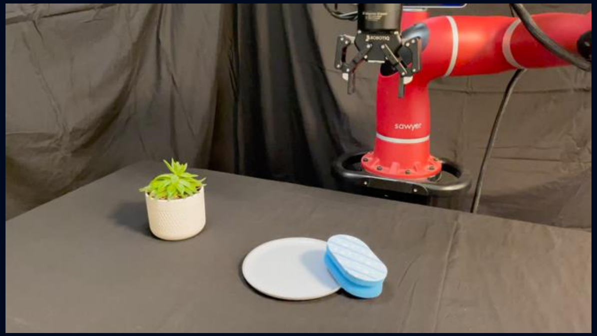
Single task-agnostic prompt

````
# INPUT: [INSERT EE POSITION], [INSERT TASK]
MAIN_PROMPT = \
"""You are a sentient AI that can control a robot arm by generating Python code which outputs a list of trajectory points for the robot arm end-effector to follow to complete a given user command.
Each element in the trajectory list is an end-effector pose, and should be of length 4, comprising a 3D position and a rotation value.

AVAILABLE FUNCTIONS:
You must remember that this conversation is a monologue, and that you are in control. I am not able to assist you with any questions, and you must output the final code yourself by making use of the available
information, common sense, and general knowledge.
You are, however, able to call any of the following Python functions, if required, as often as you want:
1. detect_object(object_or_object_part: str) -> None: This function will not return anything, but only print the position, orientation, and dimensions of any object or object part in the environment. This information
will be printed for as many instances of the queried object or object part in the environment. If there are multiple objects or object parts to detect, call one function for each object or object part, all before
executing any trajectories. The unit is in metres.
2. execute_trajectory(trajectory: list) -> None: This function will execute the list of trajectory points on the robot arm end-effector, and will also not return anything.
3. open_gripper() -> None: This function will open the gripper on the robot arm, and will also not return anything.
4. close_gripper() -> None: This function will close the gripper on the robot arm, and will also not return anything.
5. task_completed() -> None: Call this function only when the task has been completed. This function will also not return anything.
When calling any of the functions, make sure to stop generation after each function call and wait for it to be executed, before calling another function and continuing with your plan.

ENVIRONMENT SET-UP:
The 3D coordinate system of the environment is as follows:
    1. The x-axis is in the horizontal direction, increasing to the right.
    2. The y-axis is in the depth direction, increasing away from you.
    3. The z-axis is in the vertical direction, increasing upwards.
The robot arm end-effector is currently positioned at [INSERT EE POSITION], with the rotation value at 0, and the gripper open.
The robot arm is in a top-down set-up, with the end-effector facing down onto a tabletop. The end-effector is therefore able to rotate about the z-axis, from -pi to pi radians.
The end-effector gripper has two fingers, and they are currently parallel to the x-axis.
The gripper can only grasp objects along sides which are shorter than 0.08.
Negative rotation values represent clockwise rotation, and positive rotation values represent anticlockwise rotation. The rotation values should be in radians.

COLLISION AVOIDANCE:
If the task requires interaction with multiple objects:
1. Make sure to consider the object widths, lengths, and heights so that an object does not collide with another object or with the tabletop, unless necessary.
2. It may help to generate additional trajectories and add specific waypoints (calculated from the given object information) to clear objects and the tabletop and avoid collisions, if necessary.

VELOCITY CONTROL:
1. The default speed of the robot arm end-effector is 100 points per trajectory.
2. If you need to make the end-effector follow a particular trajectory more quickly, then generate fewer points for the trajectory, and vice versa.

CODE GENERATION:
When generating the code for the trajectory, do the following:
1. Describe briefly the shape of the motion trajectory required to complete the task.
2. The trajectory could be broken down into multiple steps. In that case, each trajectory step (at default speed) should contain at least 100 points. Define general functions which can be reused for the different
trajectory steps whenever possible, but make sure to define new functions whenever a new motion is required. Output a step-by-step reasoning before generating the code.
3. If the trajectory is broken down into multiple steps, make sure to chain them such that the start point of trajectory_2 is the same as the end point of trajectory_1 and so on, to ensure a smooth overall trajectory.
Call the execute_trajectory function after each trajectory step.
4. When defining the functions, specify the required parameters, and document them clearly in the code. Make sure to include the orientation parameter.
5. If you want to print the calculated value of a variable to use later, make sure to use the print function to three decimal places, instead of simply writing the variable name. Do not print any of the trajectory
variables, since the output will be too long.
6. Mark any code clearly with the ```python and ``` tags.

INITIAL PLANNING 1:
If the task requires interaction with an object part (as opposed to the object as a whole), describe which part of the object would be most suitable for the gripper to interact with.
Then, detect the necessary objects in the environment. Stop generation after this step to wait until you obtain the printed outputs from the detect_object function calls.

INITIAL PLANNING 2:
Then, output Python code to decide which object to interact with, if there are multiple instances of the same object.
Then, describe how best to approach the object (for example, approaching the midpoint of the object, or one of its edges, etc.), depending on the nature of the task, or the object dimensions, etc.
Then, output a detailed step-by-step plan for the trajectory, including when to lower the gripper to make contact with the object, if necessary.
Finally, perform each of these steps one by one. Name each trajectory variable with the trajectory number.
Stop generation after each code block to wait for it to finish executing before continuing with your plan.

The user command is "[INSERT TASK]".
"""
````

+

Object detection

(But no trajectory optimisers, demonstrations, or action primitives)



Kwon, Di Palo, and Johns, "Language Models as Zero-Shot Trajectory Generators", RA-Letters 2024

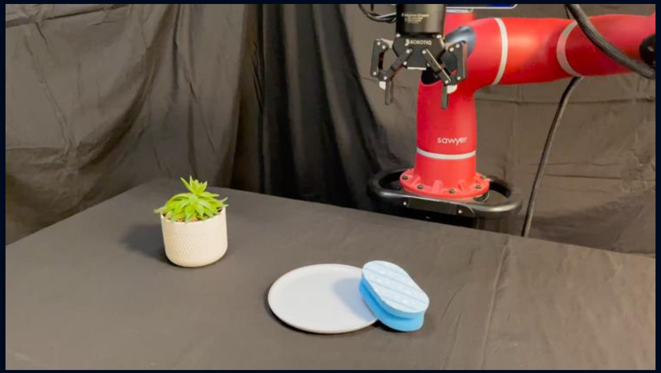
Can Large Language Models Alone Solve Robotics Tasks?

Task-agnostic prompt:

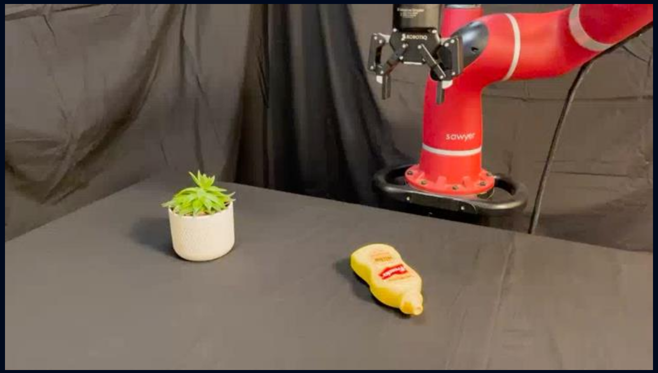
```
# INPUT: [INSERT EE POSITION], [INSERT TASK]
MAIN_PROMPT = \
"""You are a sentient AI that can control a robot arm by generating Python code which outputs a list of trajectory points for the robot arm end-effector to follow to complete a given user command.
Each element in the trajectory list is an end-effector pose, and should be of length 4, comprising a 3D position and a rotation value.

AVAILABLE FUNCTIONS:
You must remember that this conversation is a monologue, and that you are in control. I am not able to assist you with any questions, and you must output the final code yourself by making use of the available
information, common sense, and general knowledge.
You are, however, able to call any of the following Python functions, if required, as often as you want:
1. detect_object(object_or_object_part: str) -> None: This function will not return anything, but only print the position, orientation, and dimensions of any object or object part in the environment. This information
will be printed for as many instances of the queried object or object part in the environment. If there are multiple objects or object parts to detect, call one function for each object or object part, all before
executing any trajectories. The unit is in metres.
2. execute_trajectory(trajectory: list) -> None: This function will execute the list of trajectory points on the robot arm end-effector, and will also not return anything.
3. open_gripper() -> None: This function will open the gripper on the robot arm, and will also not return anything.
4. close_gripper() -> None: This function will close the gripper on the robot arm, and will also not return anything.
5. task_completed() -> None: Call this function only when the task has been completed. This function will also not return anything.
When calling any of the functions, make sure to stop generation after each function call and wait for it to be executed, before calling another function and continuing with your plan.

ENVIRONMENT SET-UP:
The 3D coordinate system of the environment is as follows:
    1. The x-axis is in the horizontal direction, increasing to the right.
    2. The y-axis is in the depth direction, increasing away from you.
    3. The z-axis is in the vertical direction, increasing upwards.
The robot arm end-effector is currently positioned at [INSERT EE POSITION], with the rotation value at 0, and the gripper open.
The robot arm is in a top-down set-up, with the end-effector facing down onto a tabletop. The end-effector is therefore able to rotate about the z-axis, from -pi to pi radians.
The end-effector gripper has two fingers, and they are currently parallel to the x-axis.
The gripper can only grasp objects along sides which are shorter than 0.08.
Negative rotation values represent clockwise rotation, and positive rotation values represent anticlockwise rotation. The rotation values should be in radians.

COLLISION AVOIDANCE:
If the task requires interaction with multiple objects:
1. Make sure to consider the object widths, lengths, and heights so that an object does not collide with another object or with the tabletop, unless necessary.
2. It may help to generate additional trajectories and add specific waypoints (calculated from the given object information) to clear objects and the tabletop and avoid collisions, if necessary.

VELOCITY CONTROL:
1. The default speed of the robot arm end-effector is 100 points per trajectory.
2. If you need to make the end-effector follow a particular trajectory more quickly, then generate fewer points for the trajectory, and vice versa.

CODE GENERATION:
When generating the code for the trajectory, do the following:
1. Describe briefly the shape of the motion trajectory required to complete the task.
2. The trajectory could be broken down into multiple steps. In that case, each trajectory step (at default speed) should contain at least 100 points. Define general functions which can be reused for the different
trajectory steps whenever possible, but make sure to define new functions whenever a new motion is required. Output a step-by-step reasoning before generating the code.
3. If the trajectory is broken down into multiple steps, make sure to chain them such that the start point of trajectory_2 is the same as the end point of trajectory_1 and so on, to ensure a smooth overall trajectory.
Call the execute_trajectory function after each trajectory step.
4. When defining the functions, specify the required parameters, and document them clearly in the code. Make sure to include the orientation parameter.
5. If you want to print the calculated value of a variable to use later, make sure to use the print function to three decimal places, instead of simply writing the variable name. Do not print any of the trajectory
variables, since the output will be too long.
6. Mark any code clearly with the ```python and ``` tags.

INITIAL PLANNING 1:
If the task requires interaction with an object part (as opposed to the object as a whole), describe which part of the object would be most suitable for the gripper to interact with.
Then, detect the necessary objects in the environment. Stop generation after this step to wait until you obtain the printed outputs from the detect_object function calls.

INITIAL PLANNING 2:
Then, output Python code to decide which object to interact with, if there are multiple instances of the same object.
Then, describe how best to approach the object (for example, approaching the midpoint of the object, or one of its edges, etc.), depending on the nature of the task, or the object dimensions, etc.
Then, output a detailed step-by-step plan for the trajectory, including when to lower the gripper to make contact with the object, if necessary.
Finally, perform each of these steps one by one. Name each trajectory variable with the trajectory number.
Stop generation after each code block to wait for it to finish executing before continuing with your plan.

The user command is "[INSERT TASK]".
"""
```

Kwon, Di Palo, and Johns, "Language Models as Zero-Shot Trajectory Generators", RA-Letters 2024

Can Large Language Models Alone Solve Robotics Tasks?

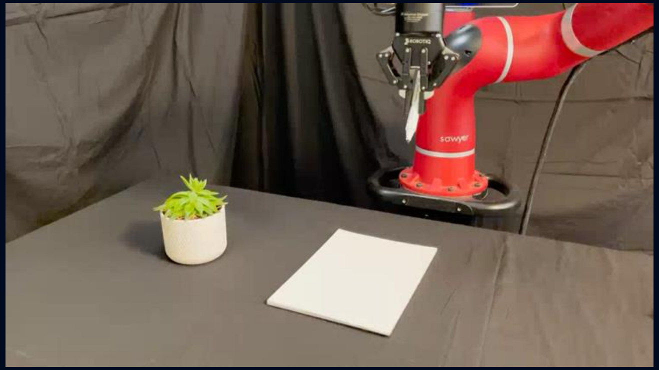


" Wipe the plate with the sponge "



" Shake the mustard bottle "

" Place the apple in the bowl "



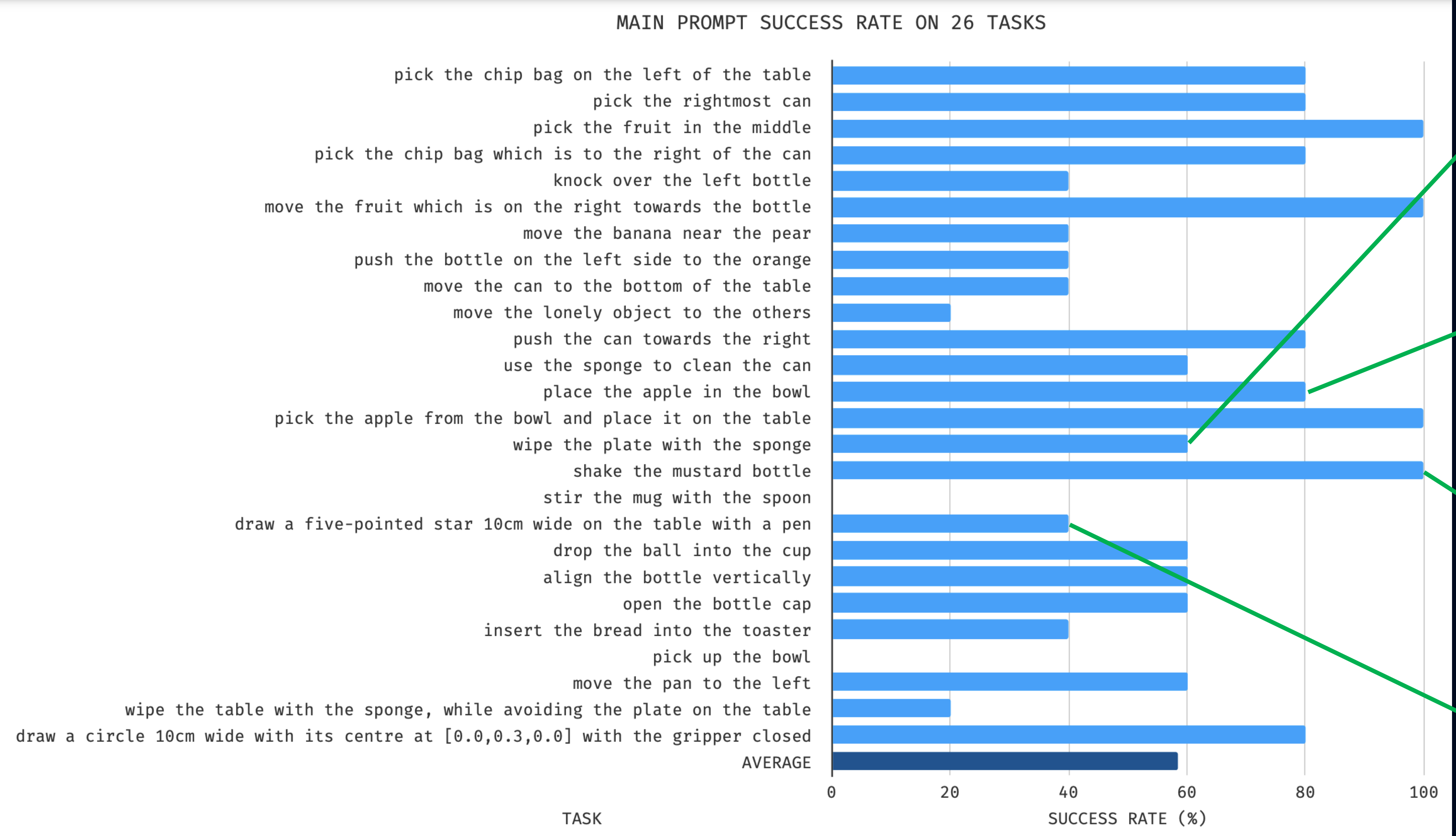
" Draw a five-pointed star 10cm wide on the table with a pen "

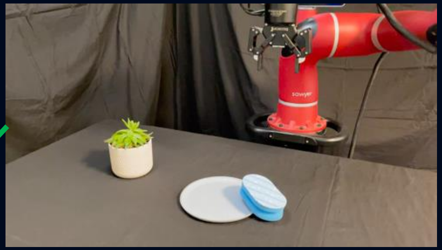
" Open the bottle cap "

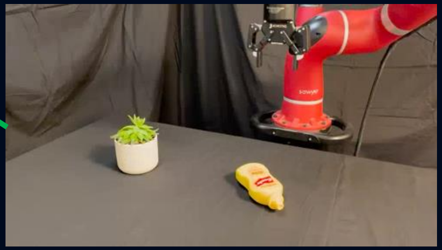
" Move the lonely object to the others "

Kwon, Di Palo, and Johns, "Language Models as Zero-Shot Trajectory Generators", RA-Letters 2024

Can Large Language Models Alone Solve Robotics Tasks?





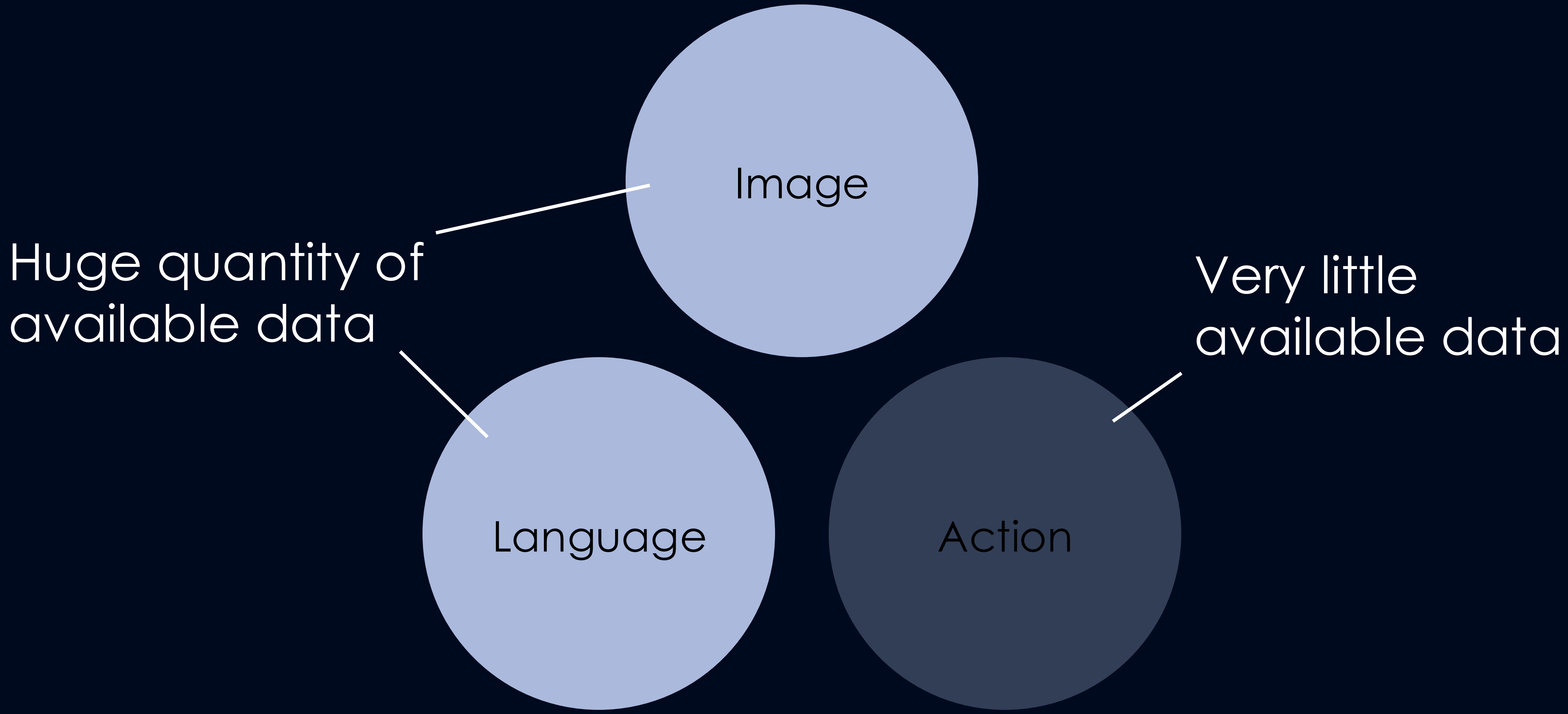


Kwon, Di Palo, and Johns, "Language Models as Zero-Shot Trajectory Generators", RA-Letters 2024

Can Large Language Models Alone Solve Robotics Tasks?

Kwon, Di Palo, and Johns, "Language Models as Zero-Shot Trajectory Generators", RA-Letters 2024

Vision & Language & Action



OpenAI's DALL-E 2 Arrives

"An astronaut riding a horse in a photorealistic style"

"Teddy bears shopping for groceries in Ancient Egypt"

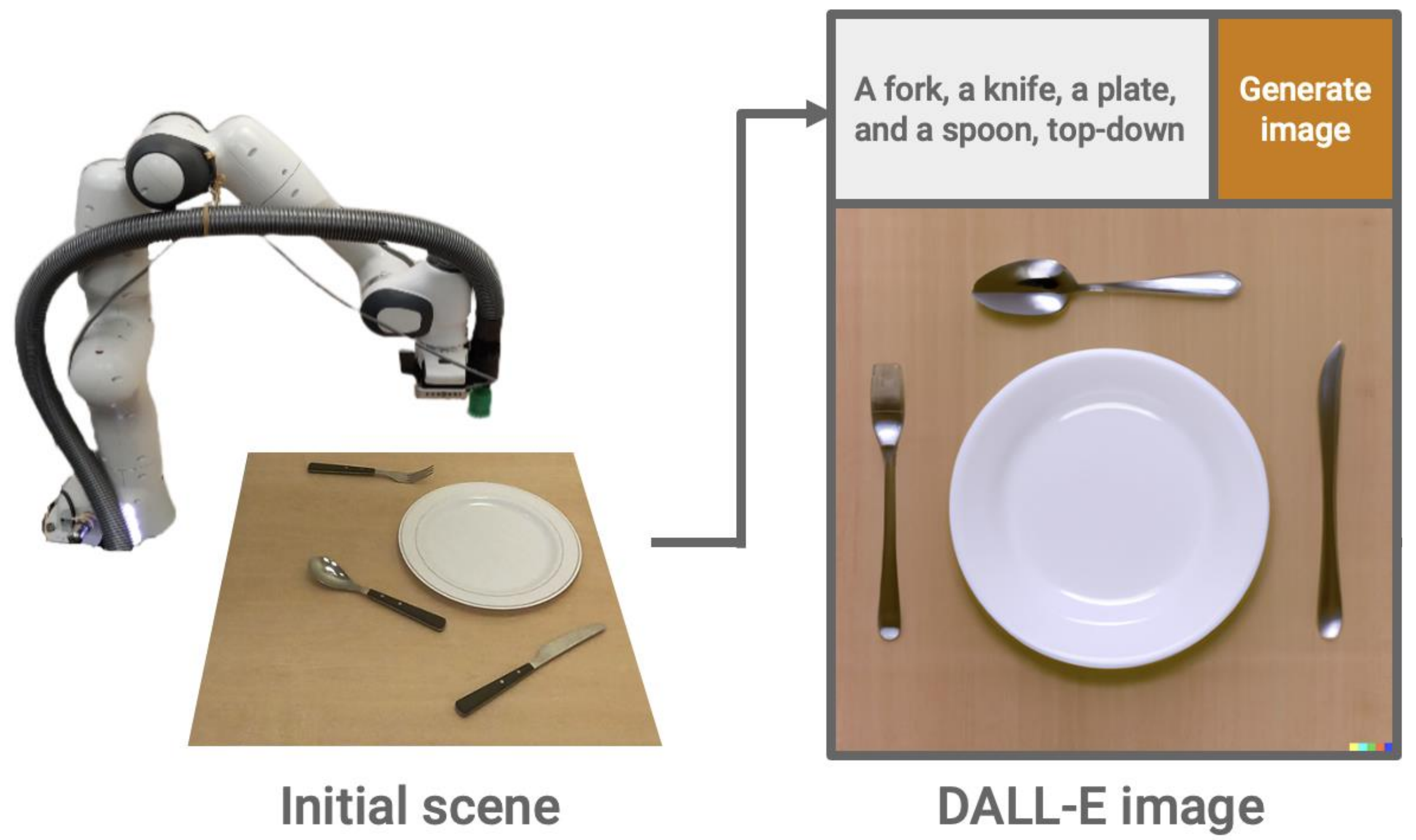
Generative AI as Imagination Engines

Kapelyukh, Vosylius, and Johns, "DALL-E-Bot: Introducing Web-Scale Diffusion Models to Robotics", in RA-Letters 2023

Generative AI as Imagination Engines

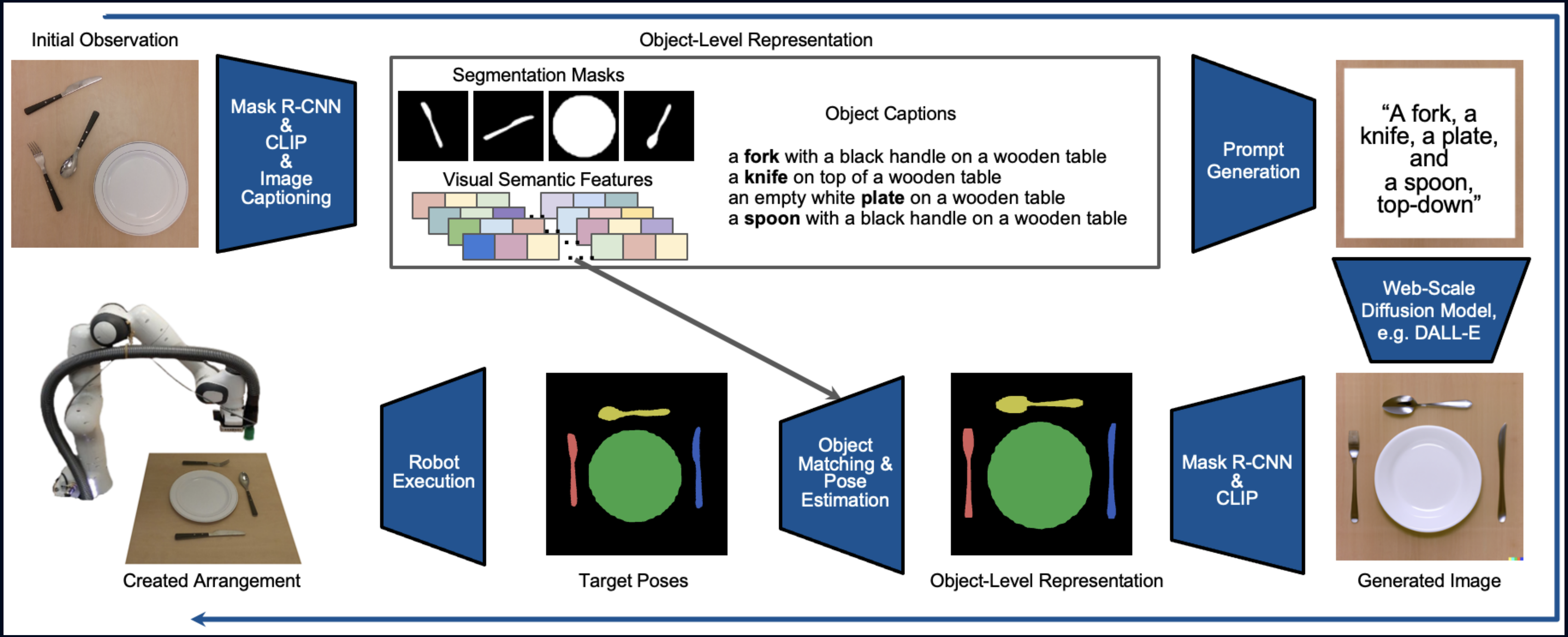
Kapelyukh, Vosylius, and Johns, "DALL-E-Bot: Introducing Web-Scale Diffusion Models to Robotics", in RA-Letters 2023

Generative AI as Imagination Engines



Kapelyukh, Vosylius, and Johns, "DALL-E-Bot: Introducing Web-Scale Diffusion Models to Robotics", in RA-Letters 2023

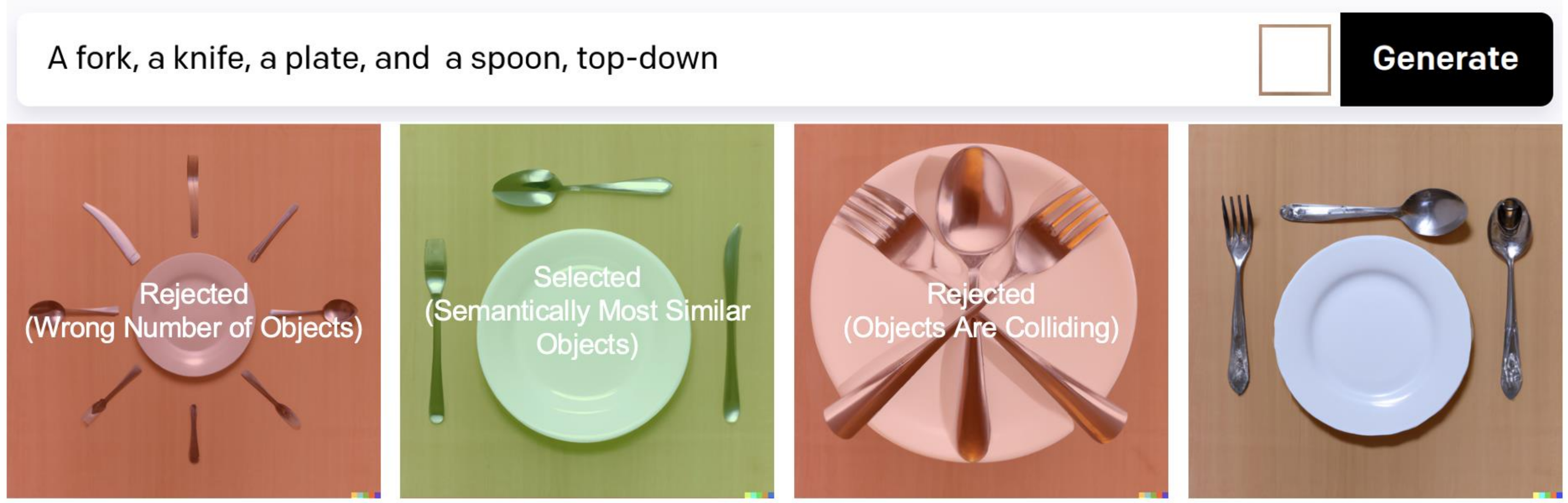
DALL-E-Bot



Kapelyukh, Vosylius, and Johns, "DALL-E-Bot: Introducing Web-Scale Diffusion Models to Robotics", in RA-Letters 2023

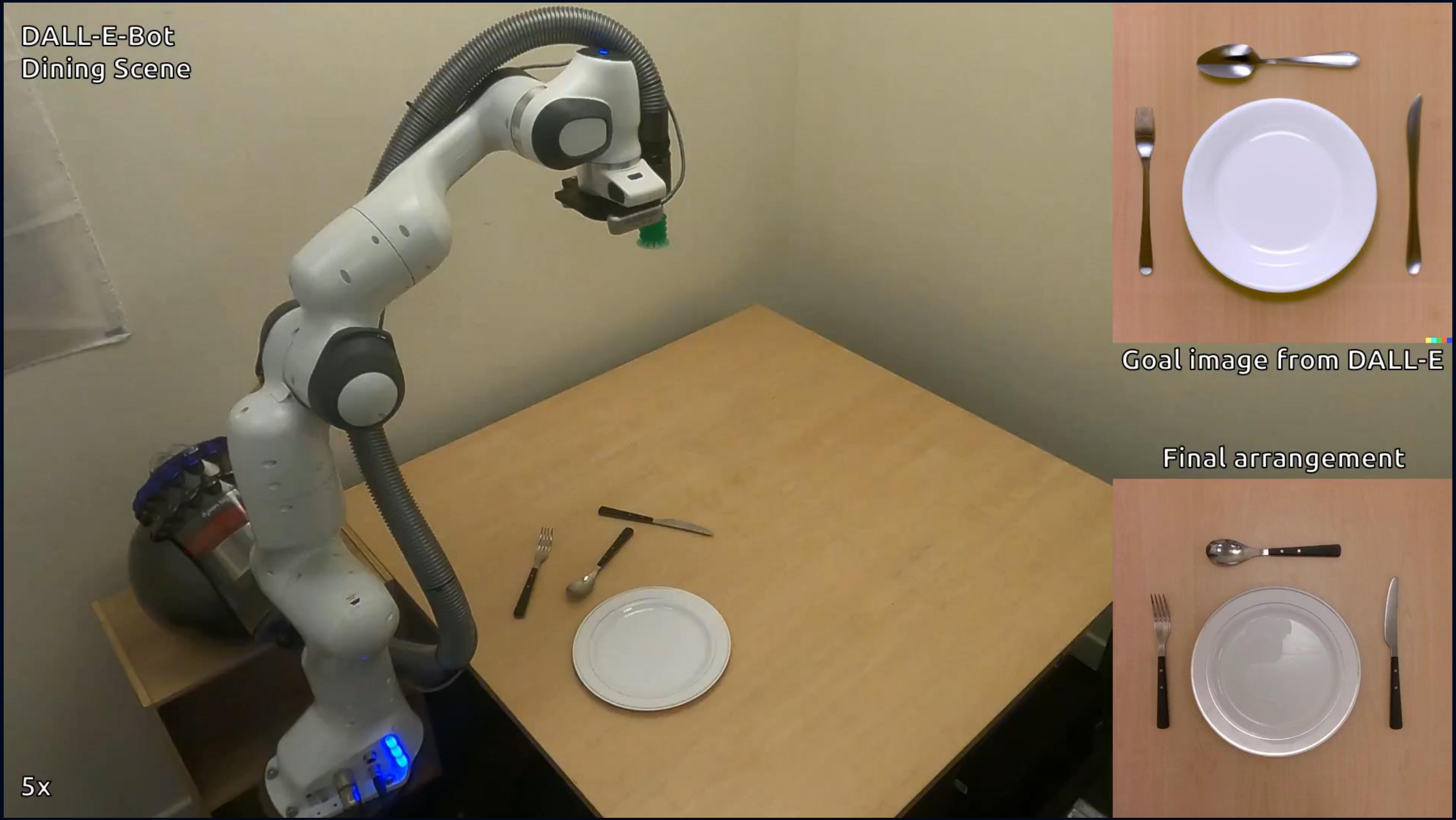
DALL-E-Bot

Initial observation



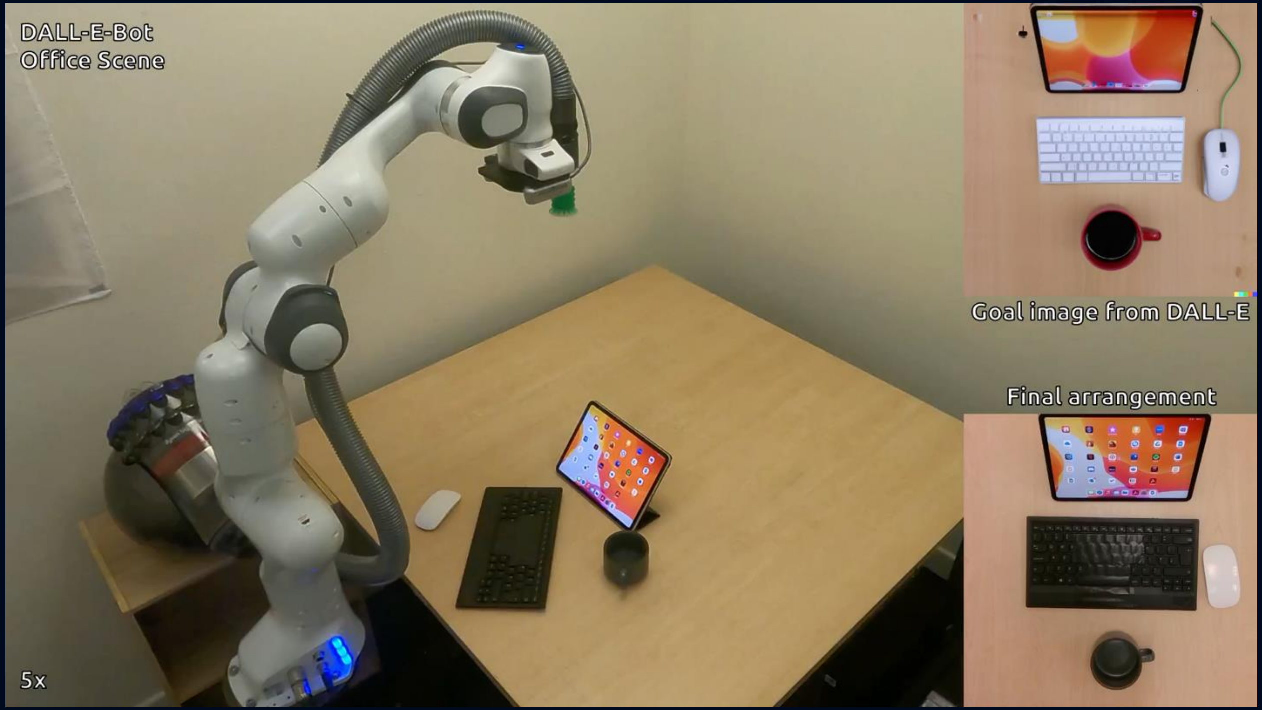
Kapelyukh, Vosylius, and Johns, "DALL-E-Bot: Introducing Web-Scale Diffusion Models to Robotics", in RA-Letters 2023

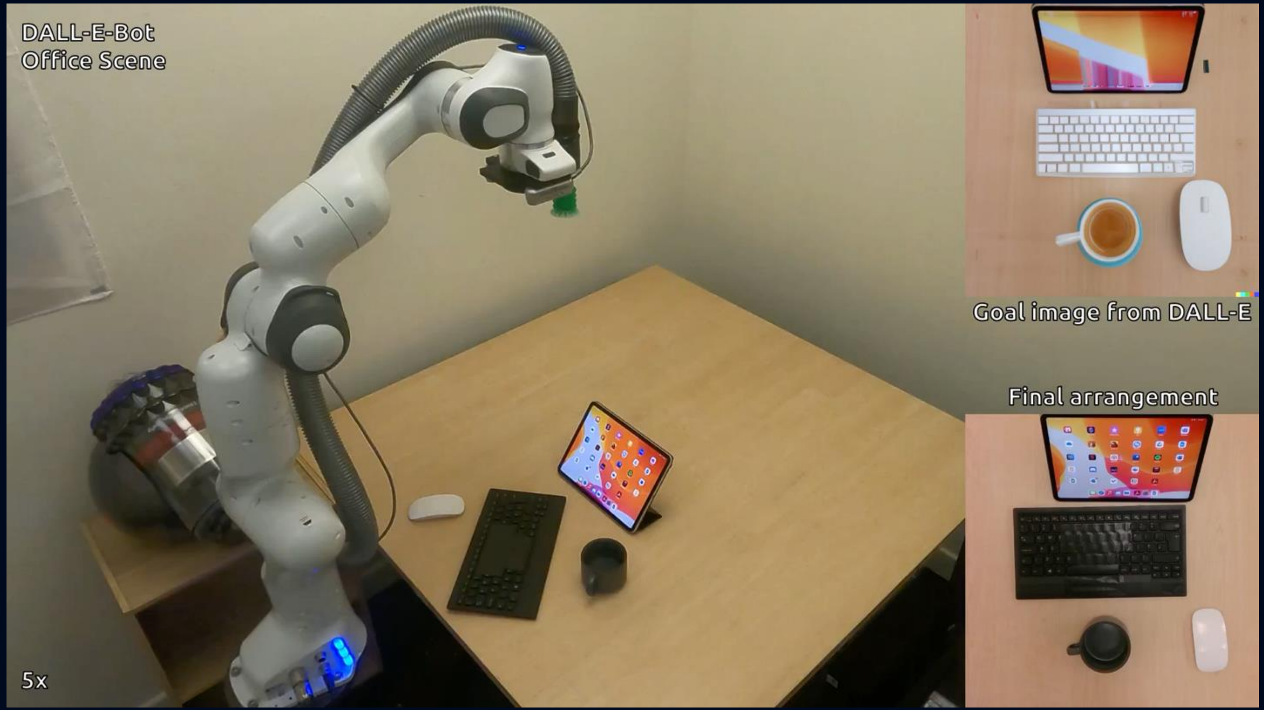
DALL-E-Bot



Kapelyukh, Vosylius, and Johns, "DALL-E-Bot: Introducing Web-Scale Diffusion Models to Robotics", in RA-Letters 2023

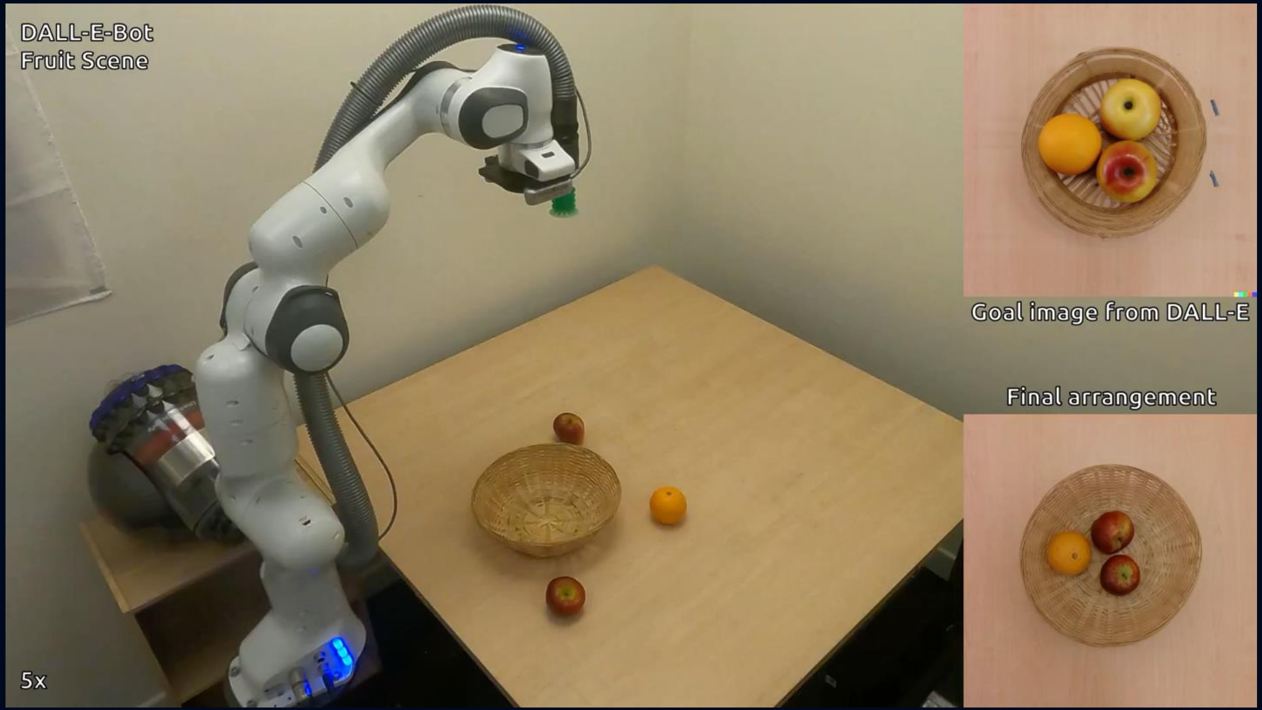
DALL-E-Bot





Kapelyukh, Vosylius, and Johns, "DALL-E-Bot: Introducing Web-Scale Diffusion Models to Robotics", in RA-Letters 2023

DALL-E-Bot



Kapelyukh, Vosylius, and Johns, "DALL-E-Bot: Introducing Web-Scale Diffusion Models to Robotics", in RA-Letters 2023

DALL-E-Bot

Random | Geometric | DALL-E-Bot with autoregression | DALL-E-Bot without filtering | DALL-E-Bot

Dining scene

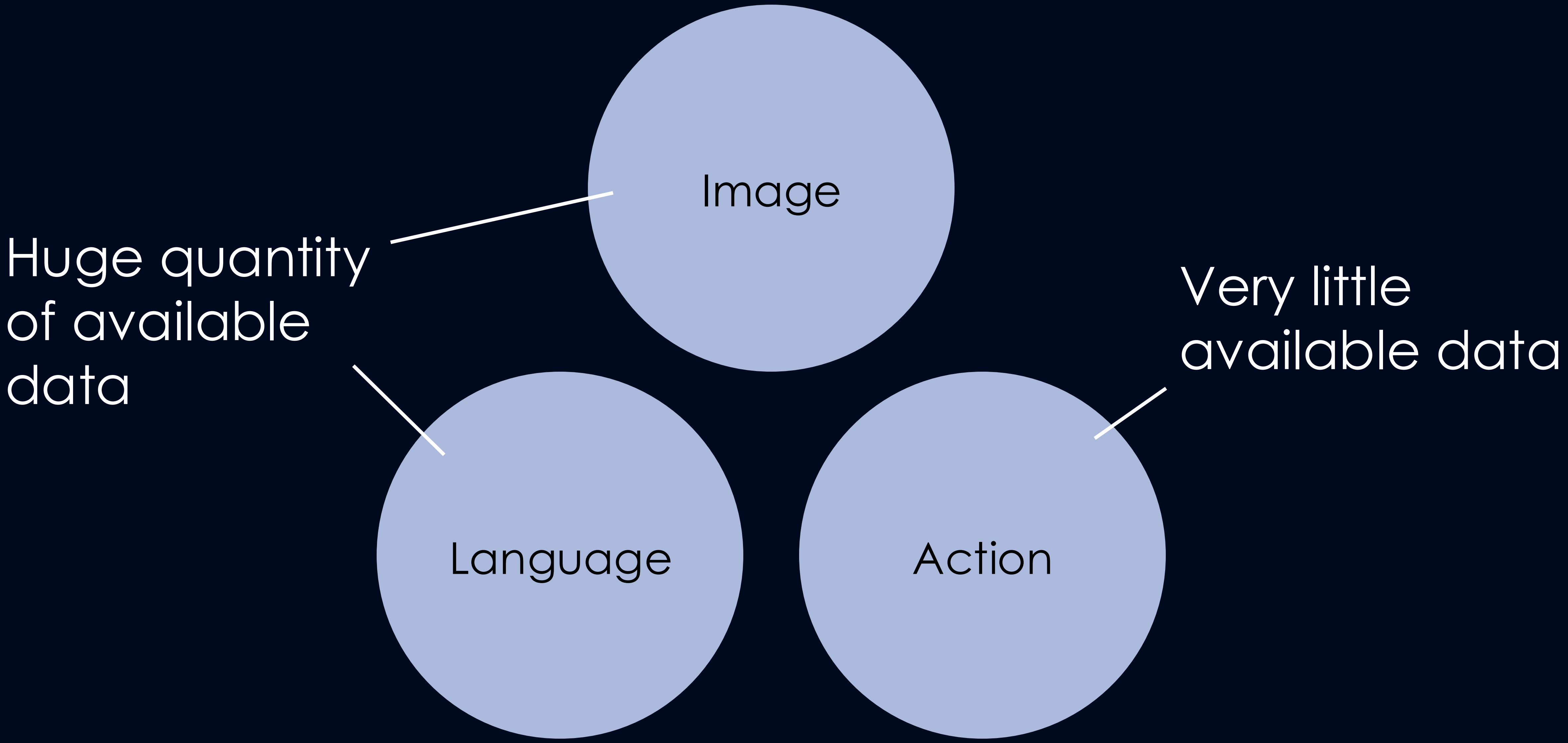
Office scene

Fruit scene

DALL-E image | Final scene | DALL-E image | Final scene | DALL-E image | Final scene

Kapelyukh, Vosylius, and Johns, "DALL-E-Bot: Introducing Web-Scale Diffusion Models to Robotics", in RA-Letters 2023

Vision & Language & Action



In-Context Learning in LLMs

Complete the pattern:

(1, 1), (1, 2), (2, 2), (2, 1)
(1, 3), (1, 6), (4, 6), (4, 3)
(5, 1), (5, 3), (7, 3), (7, 1)
(6, 4), (6, 6), (8, 6),

ChatGPT

?

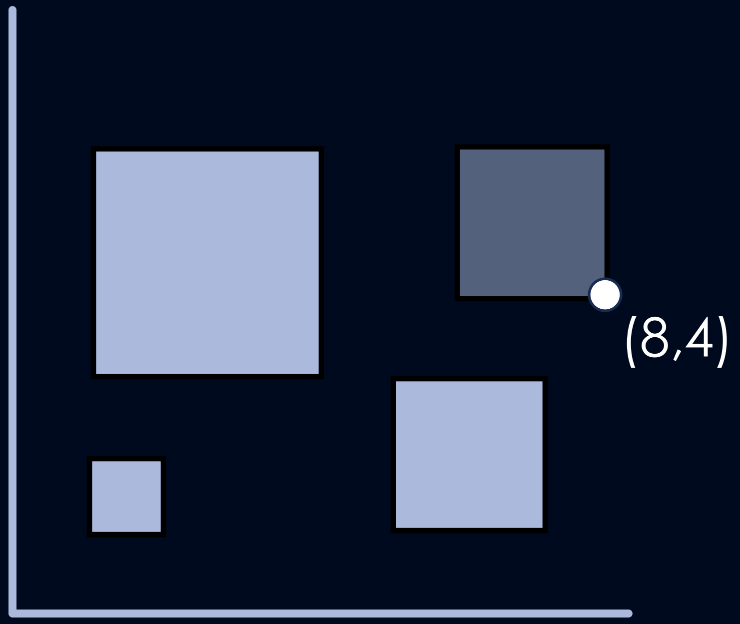
In-Context Learning in LLMs

Complete the pattern:

(1, 1), (1, 2), (2, 2), (2, 1)
(1, 3), (1, 6), (4, 6), (4, 3)
(5, 1), (5, 3), (7, 3), (7, 1)
(6, 4), (6, 6), (8, 6),

ChatGPT

(8,4)



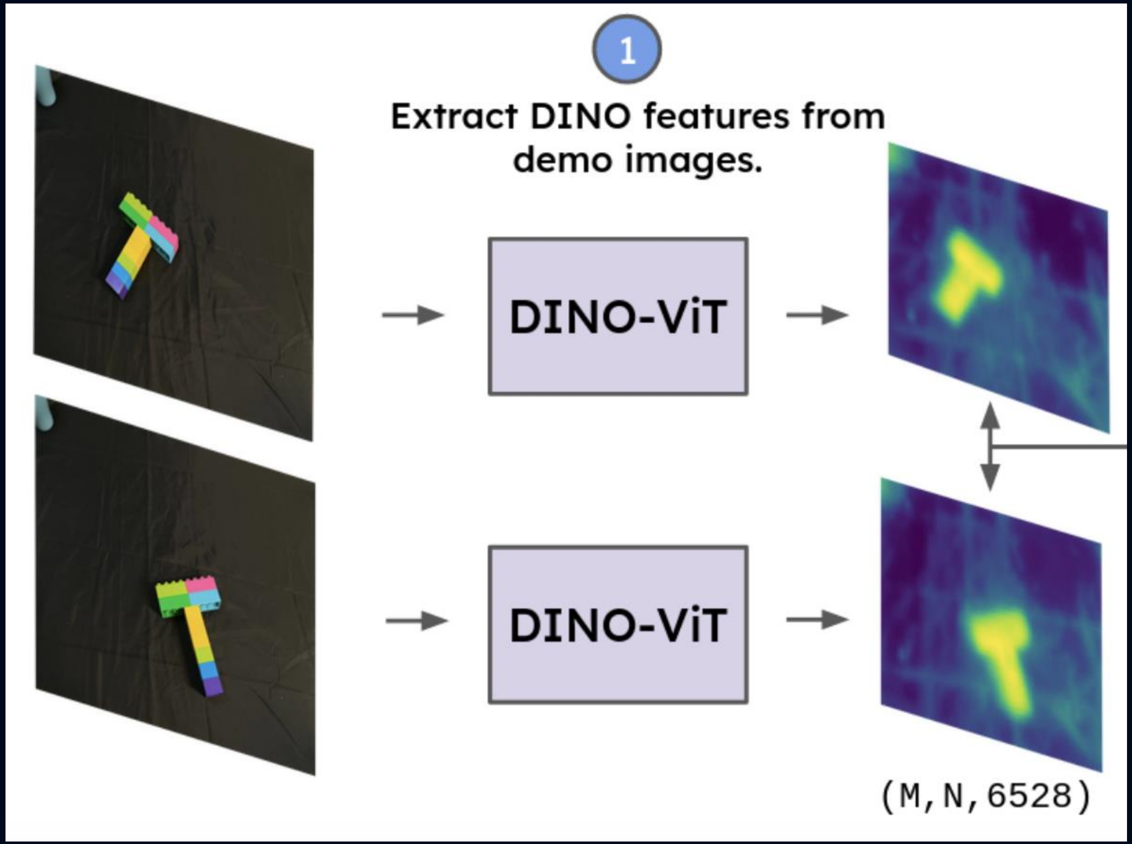
In-Context Learning in Robotics?

Complete the pattern:

Demonstration 1:
(o_1,a_1), (o_2,a_2), (o_3,a_3), …
Demonstration 2:
(o_1,a_1), (o_2,a_2), (o_3,a_3), …
…
+
Test:
$(o_{t,}$ …)

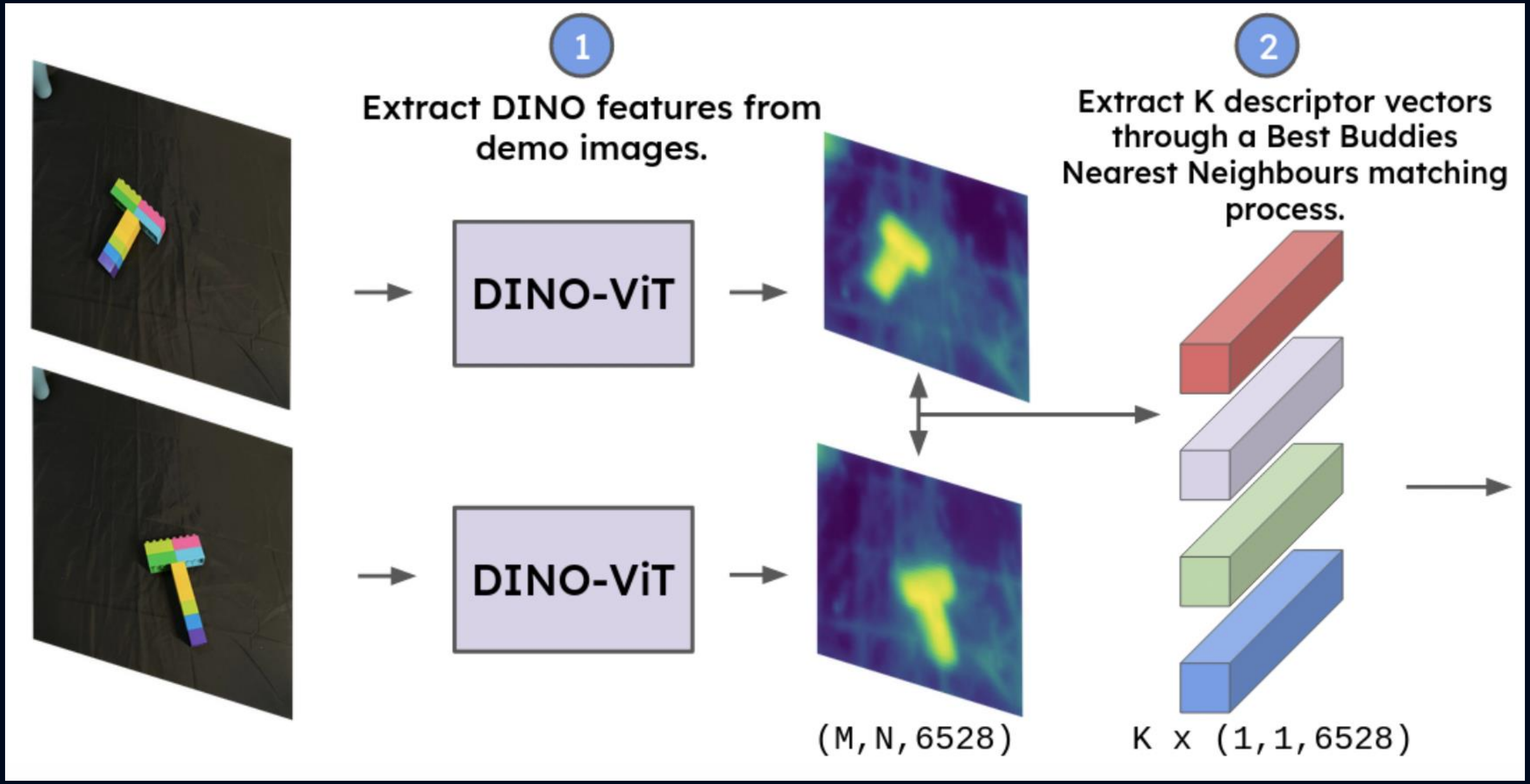
→ ChatGPT → a_t

Keypoint Action Tokens



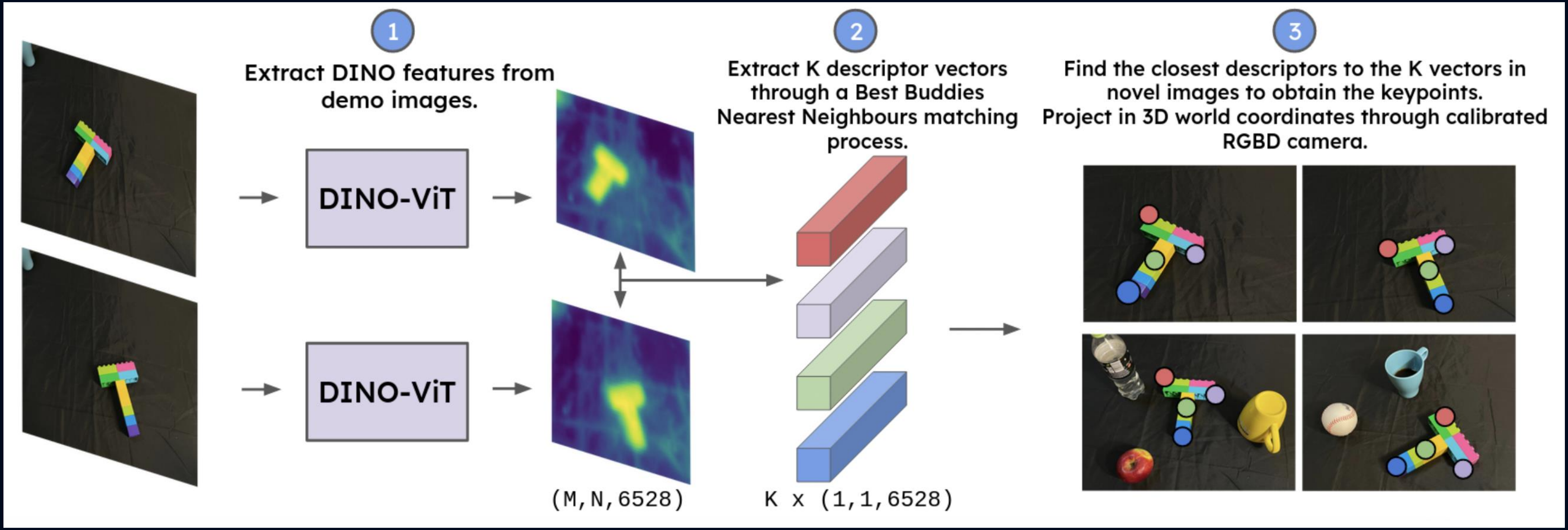
Di Palo and Johns, "Keypoint Action Tokens Enable In-Context Imitation Learning in Robotics", RSS 2024

Keypoint Action Tokens



Di Palo and Johns, "Keypoint Action Tokens Enable In-Context Imitation Learning in Robotics", RSS 2024

Keypoint Action Tokens



Di Palo and Johns, "Keypoint Action Tokens Enable In-Context Imitation Learning in Robotics", RSS 2024

Keypoint Action Tokens

Di Palo and Johns, "Keypoint Action Tokens Enable In-Context Imitation Learning in Robotics", RSS 2024

Keypoint Action Tokens

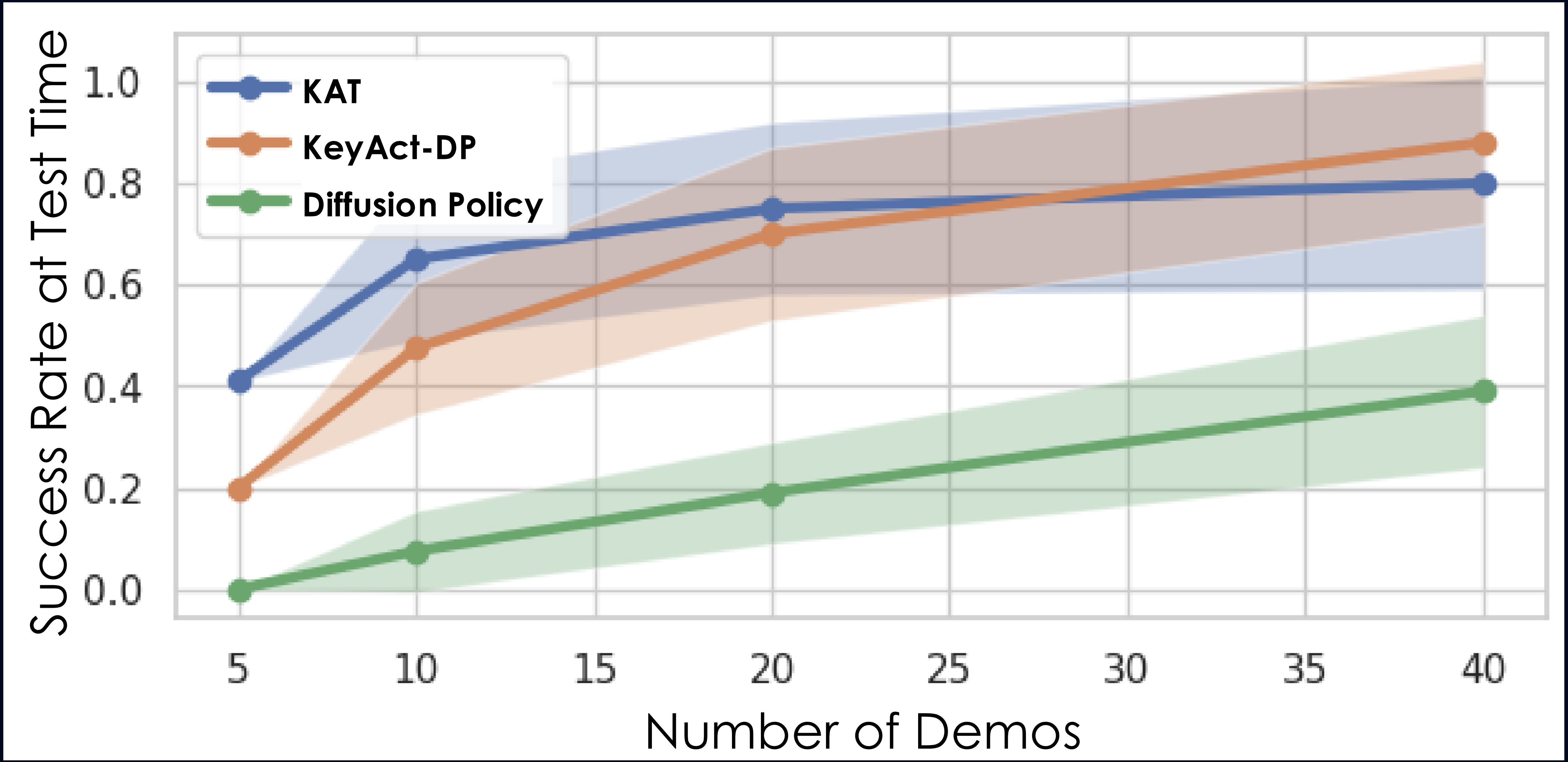
Di Palo and Johns, "Keypoint Action Tokens Enable In-Context Imitation Learning in Robotics", RSS 2024

Keypoint Action Tokens

10x

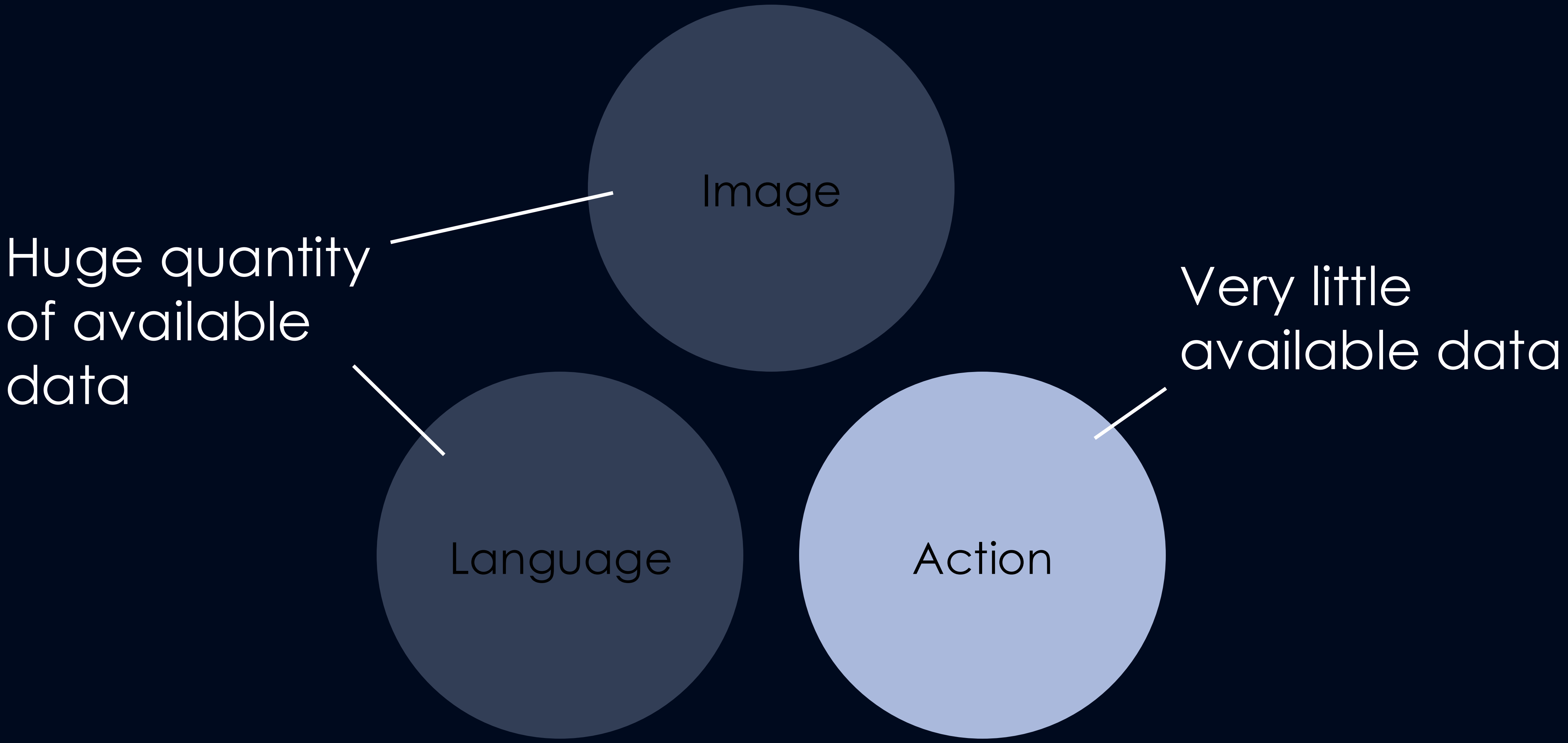
Di Palo and Johns, "Keypoint Action Tokens Enable In-Context Imitation Learning in Robotics", RSS 2024

Keypoint Action Tokens



Di Palo and Johns, "Keypoint Action Tokens Enable In-Context Imitation Learning in Robotics", RSS 2024

Vision & Language & Action



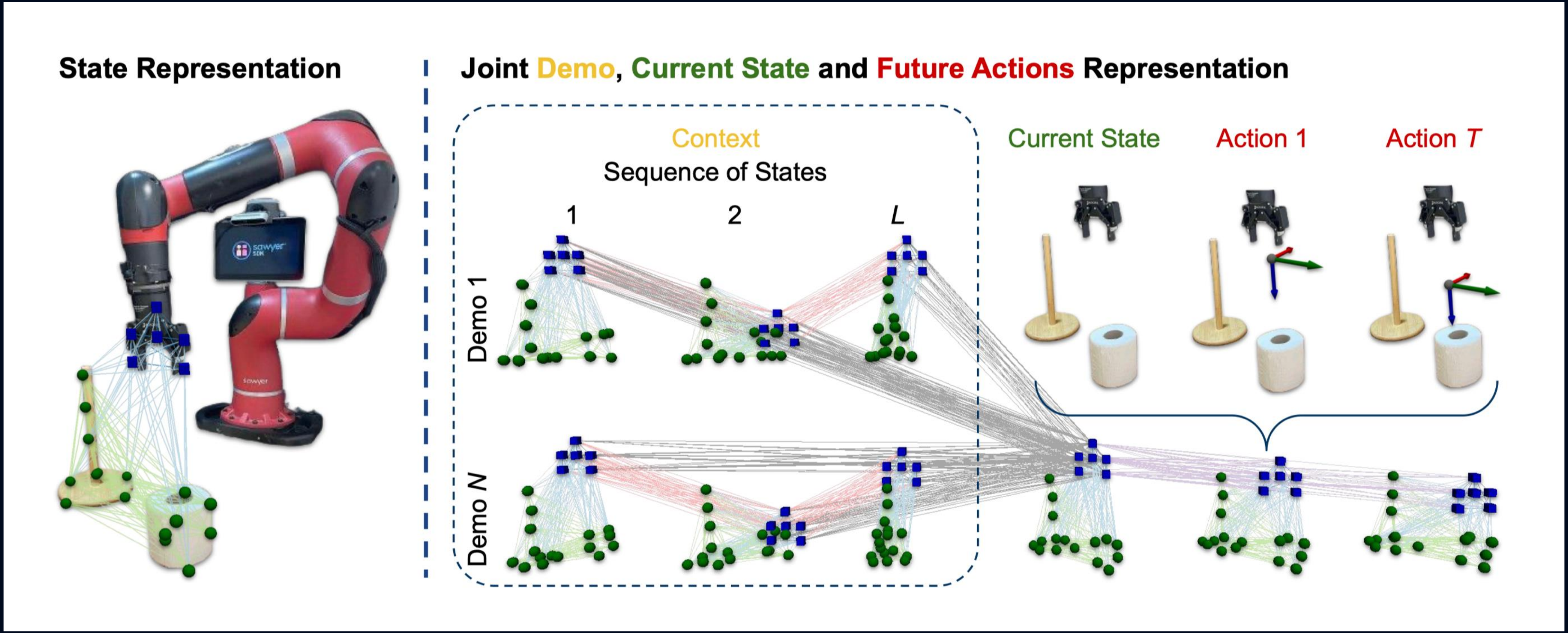
In-Context Learning in Robotics

Diffusion

1. What are the optimal inductive biases?

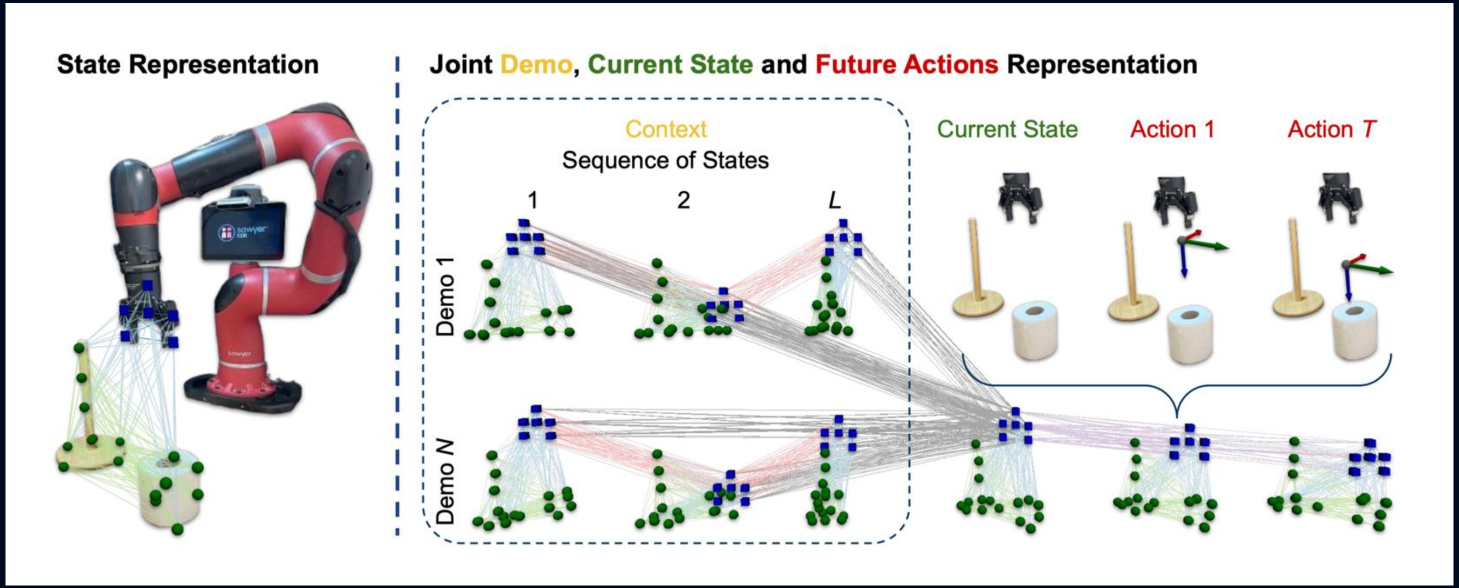
2. How can we generate the training data?

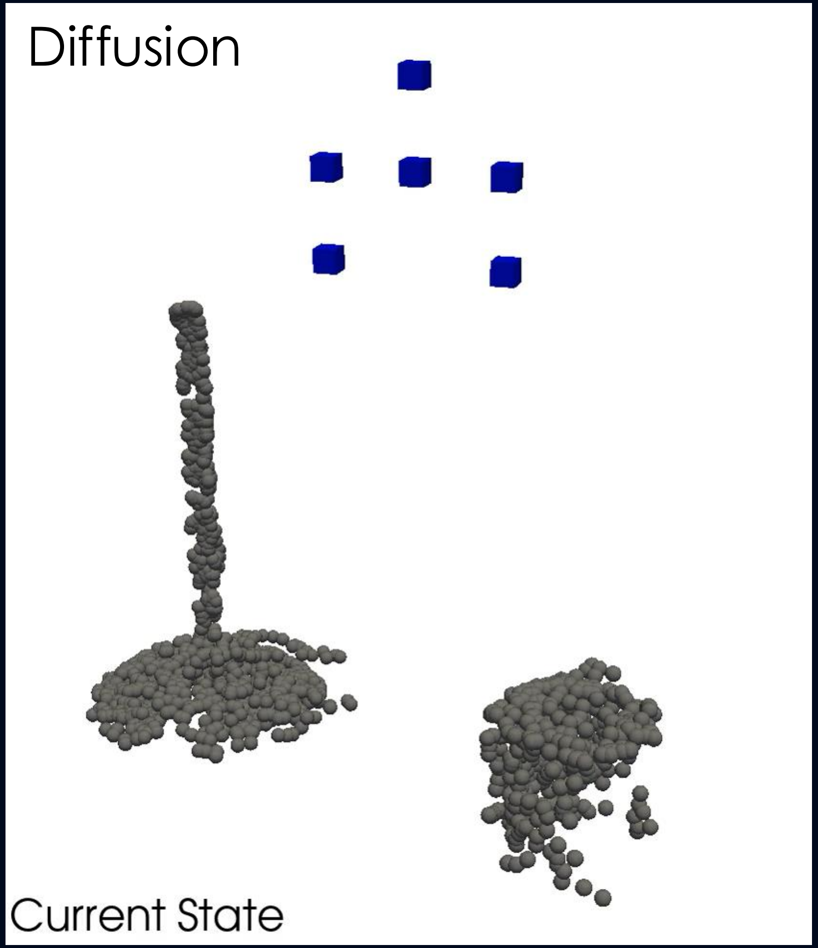
Instant Policy



Vosylius and Johns, "Instant Policy: In-Context Imitation Learning via Graph Diffusion", 2024

Instant Policy





Vosylius and Johns, "Instant Policy: In-Context Imitation Learning via Graph Diffusion", 2024

Instant Policy

The only training data we need: random, simulated "pseudo-demonstrations"

pseudo-task 1

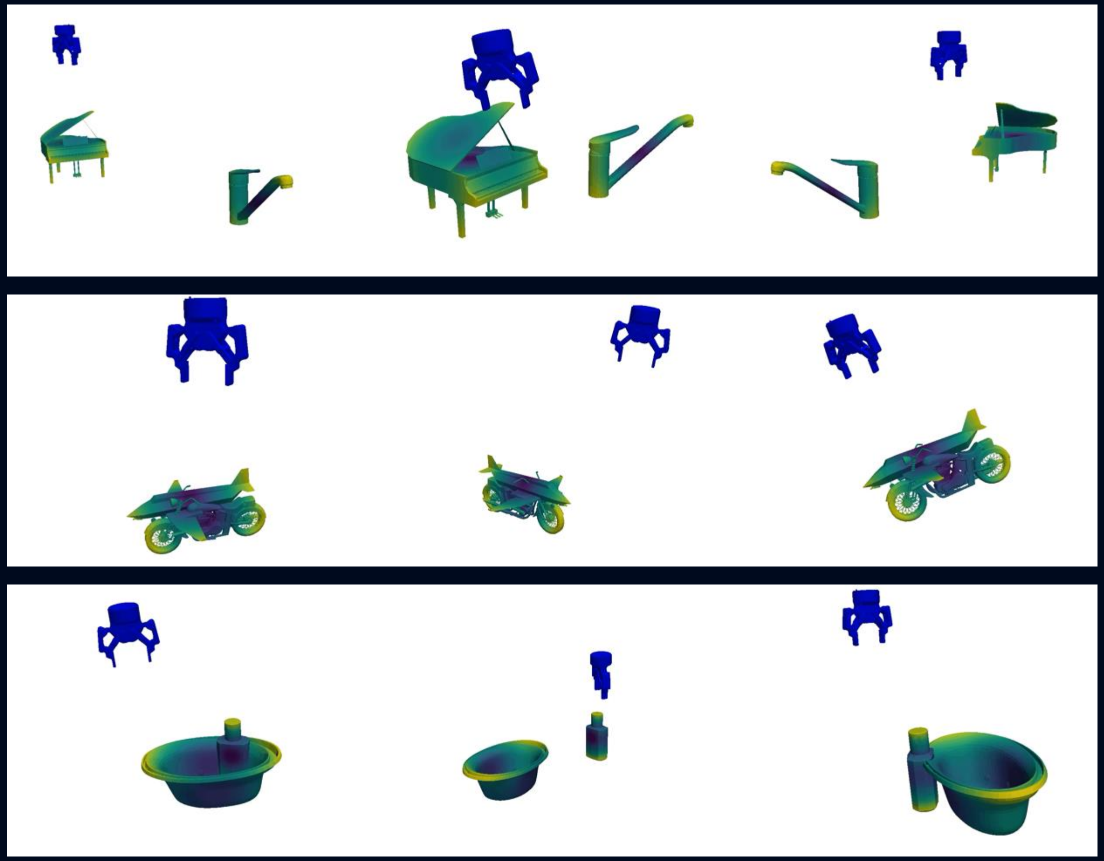
pseudo-task 2

pseudo-task 3

.

.

.

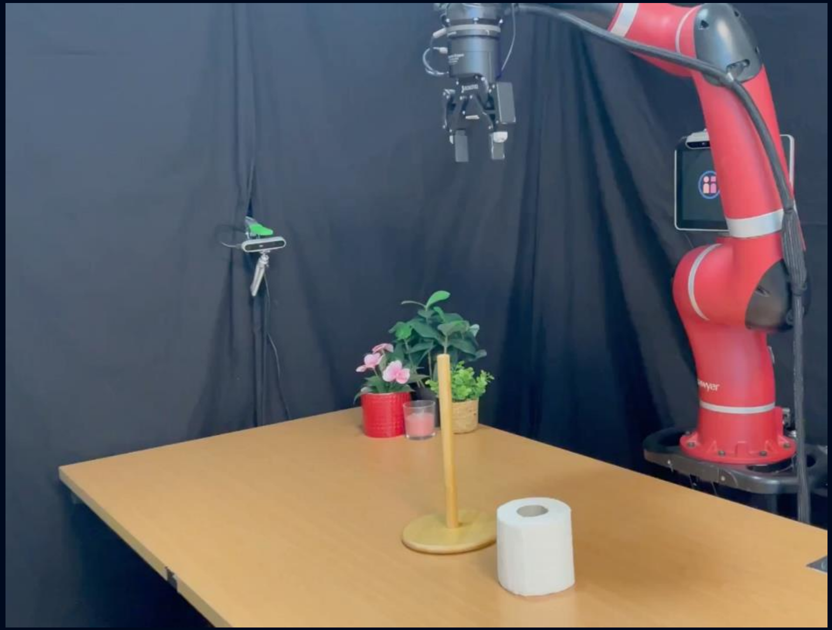


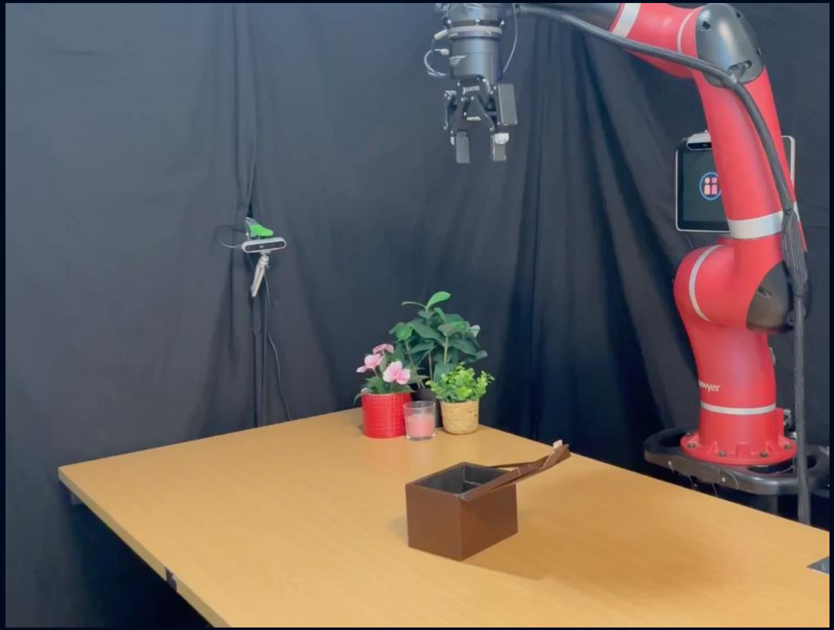
Vosylius and Johns, "Instant Policy: In-Context Imitation Learning via Graph Diffusion", 2024

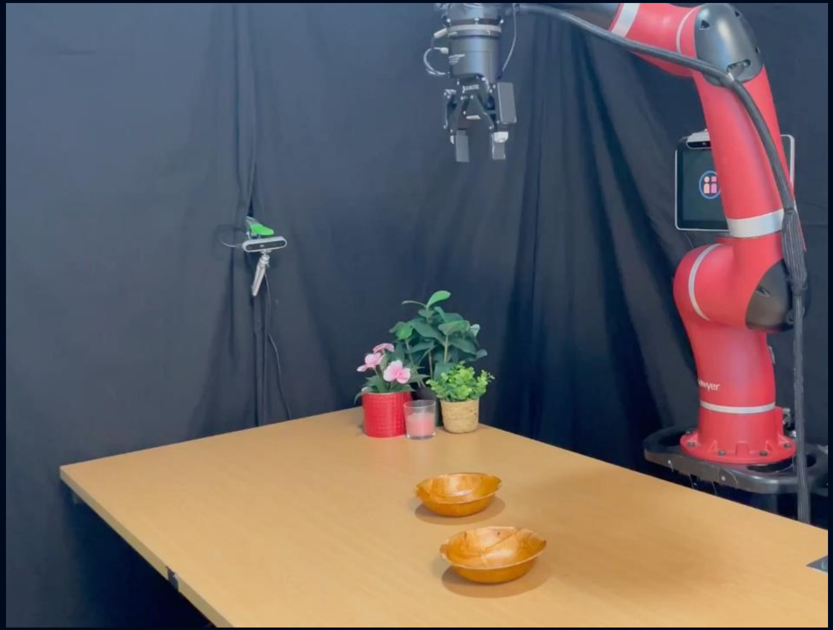
Instant Policy

Vosylius and Johns, "Instant Policy: In-Context Imitation Learning via Graph Diffusion", 2024

Instant Policy







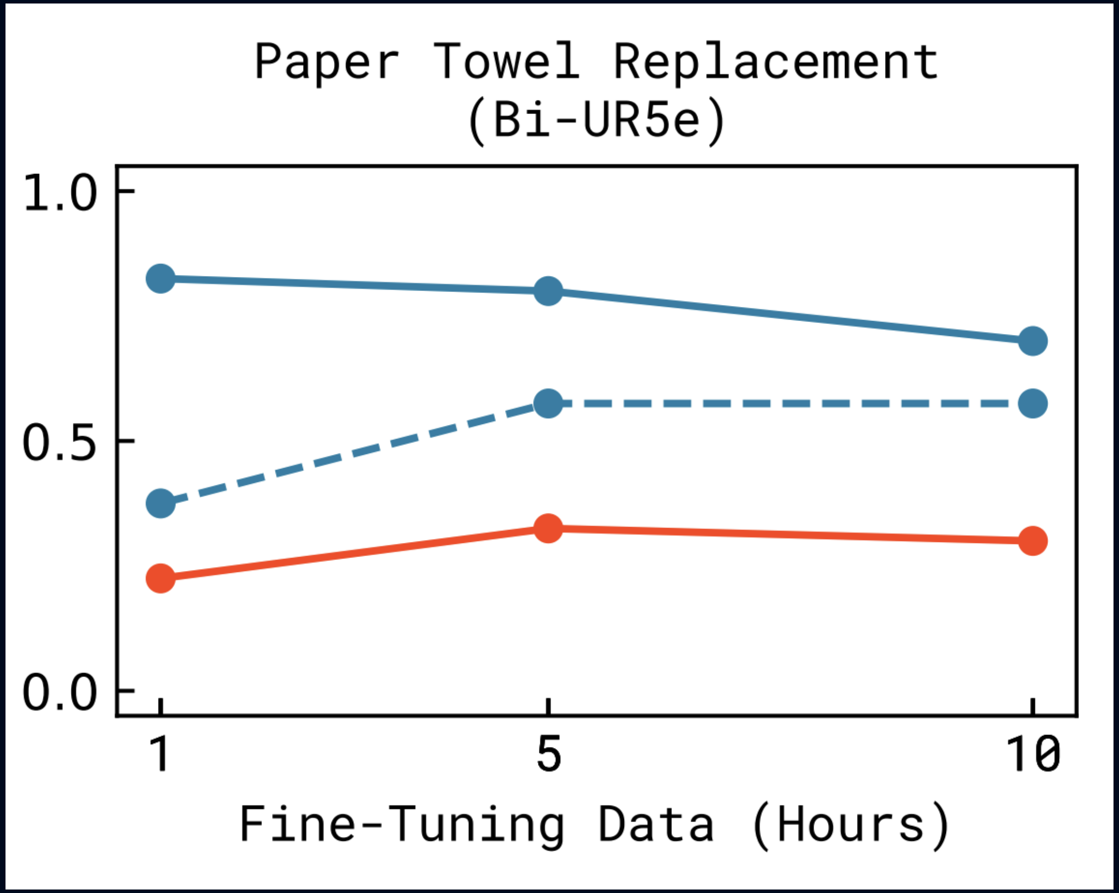
Vosylius and Johns, "Instant Policy: In-Context Imitation Learning via Graph Diffusion", 2024

Instant Policy

Vosylius and Johns, "Instant Policy: In-Context Imitation Learning via Graph Diffusion", 2024

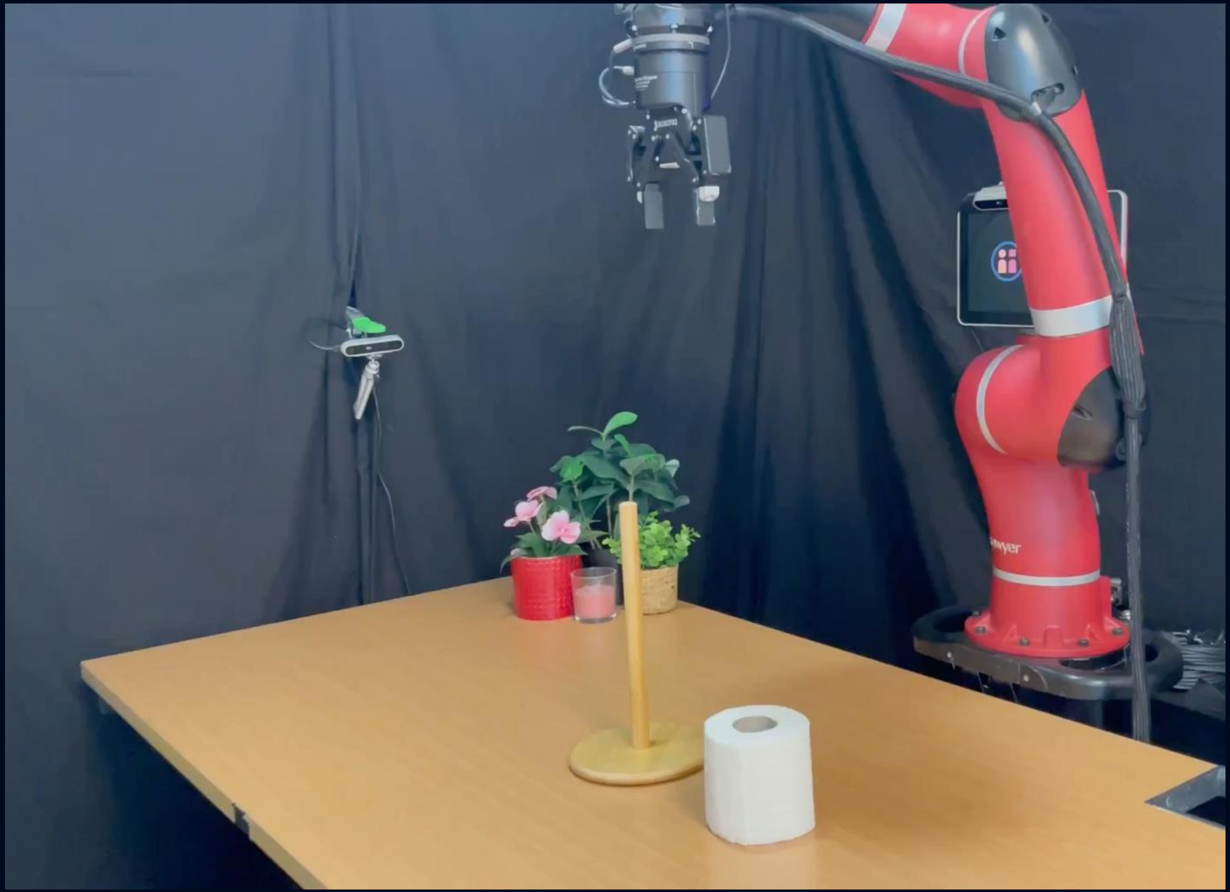
Fine-Tuning vs In-Context Learning

Physical Intelligence, 2024



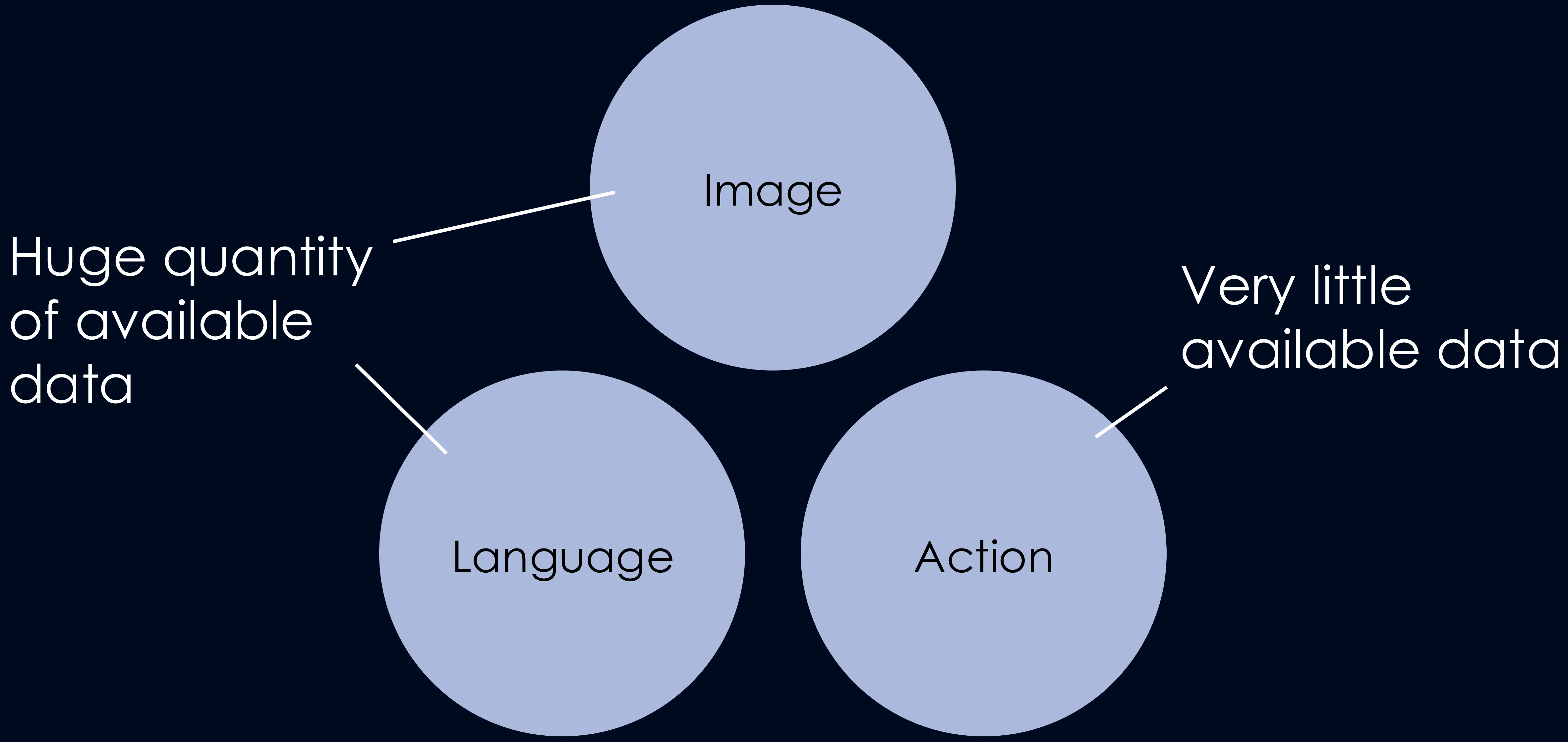
Fine-Tuning vs In-Context Learning

Physical Intelligence, 2024



Vitalis and Johns, 2024

Vision & Language & Action



Acknowledgements

Kamil Dreczkowski | Shikun Liu | Norman Di Palo | Vitalis Vosylius | Ivan Kapelyukh | Georgios Papagiannis | Pietro Vitiello | Yifei Ren | John Kwon

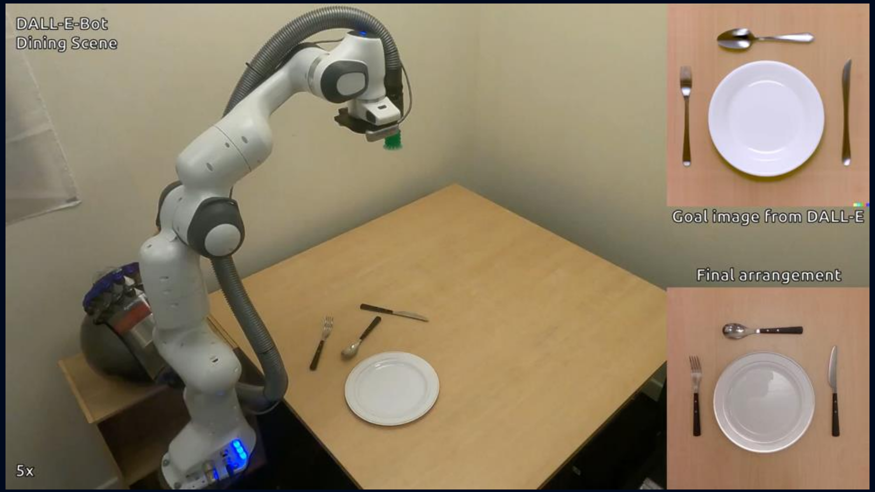
EPSRC
Engineering and Physical Sciences Research Council

dyson

Royal Academy of Engineering

The Robot Learning Lab
at
Imperial College London

Questions?



www.robot-learning.uk