Cloud Computing 2025
Valencia, Spain

\Orchestrating a brighter world NEC

LLM-based Distributed Code Generation and Cost-Efficient Execution in the Cloud

K. Rao, G. Coviello, G. Mellone, C. G. De Vita, S. Chakradhar

Kunal Rao
Researcher
NEC Laboratories America, Inc
kunal@nec-labs.com

Brief bio

- Work as Researcher at NEC Laboratories America, Inc. in Princeton, NJ

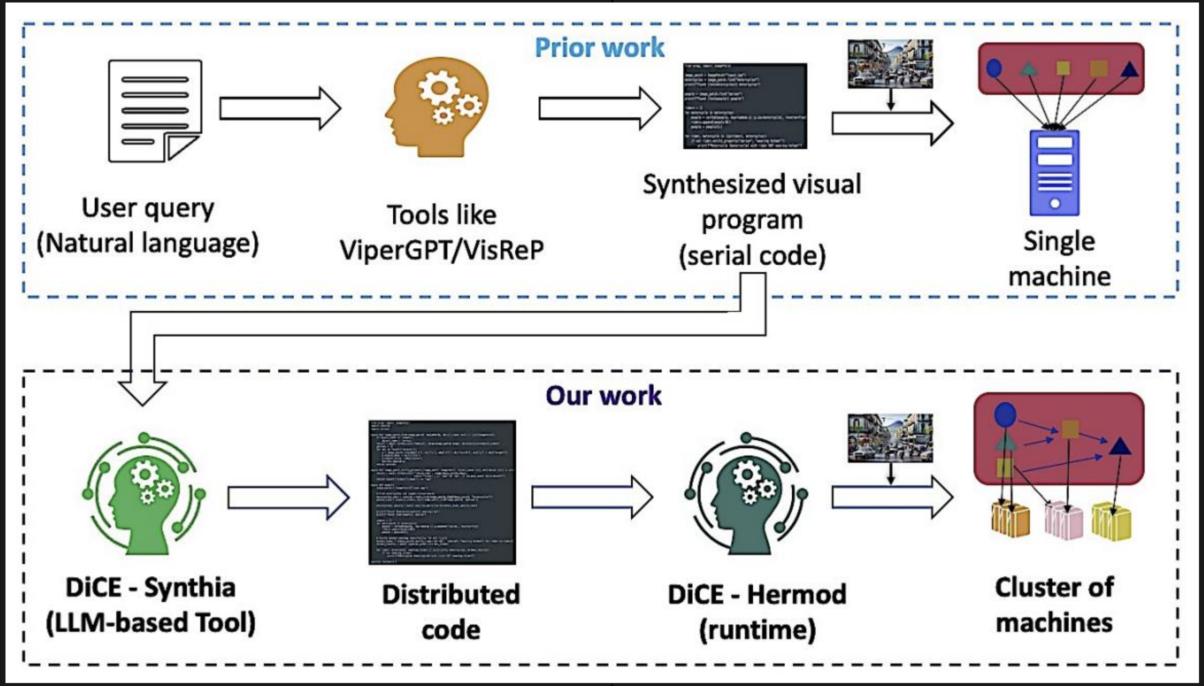
- Current research: Generative AI, Edge and Cloud computing, leveraging AI/ML models to solve systems problems, gaining utility from AI/ML models for real-world applications

- Past research: High Performance Computing, GPGPU and Xeon Phi computing, Graph analytics, Video Analytics (Computer Vision, Applied Machine Learning)

- 27 granted patents and several are published and pending, 28 published papers

DiCE and DiCE-M system

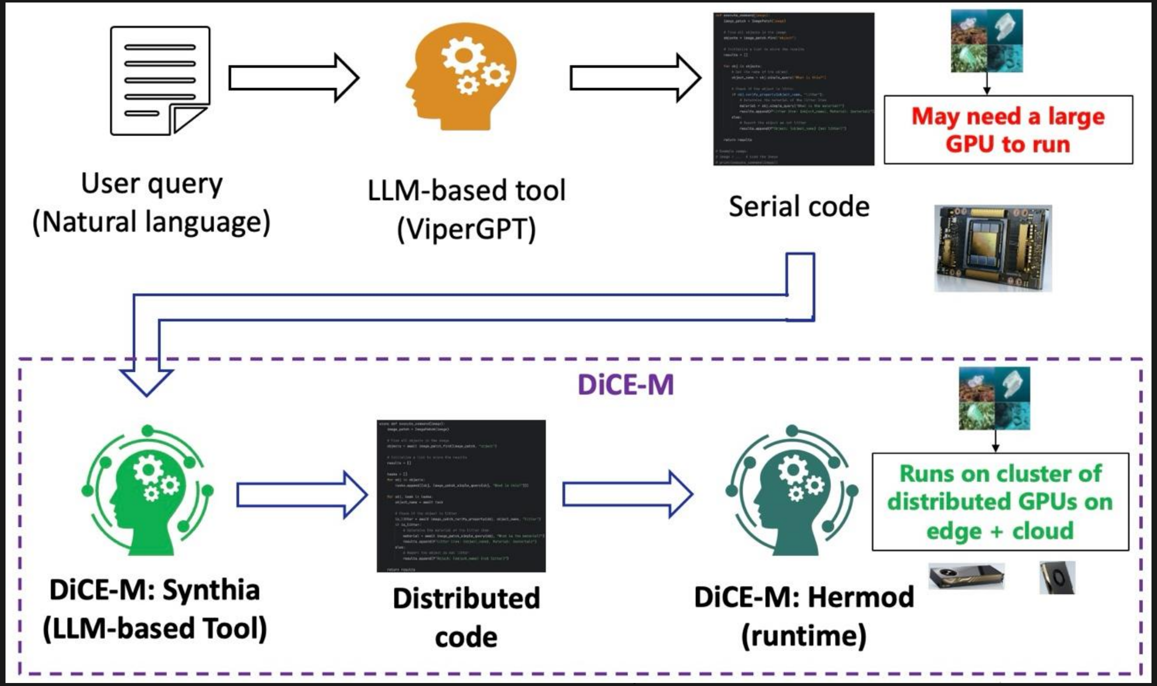
GQA dataset



DiCE: https://ieeexplore.ieee.org/document/10795392/

PICOM 2024

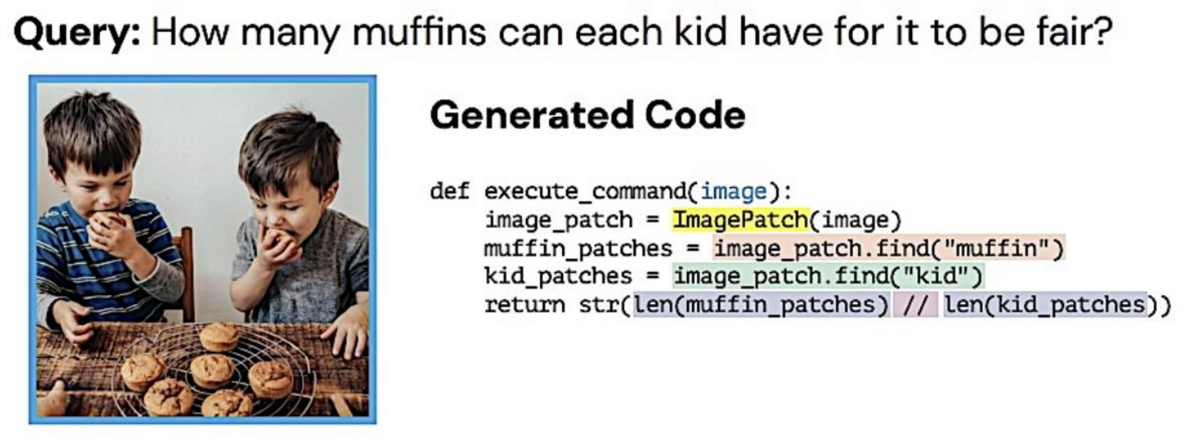
Real-world marine application
(runs on edge and cloud)



DiCE-M: https://ieeexplore.ieee.org/document/10818183

SEC 2024

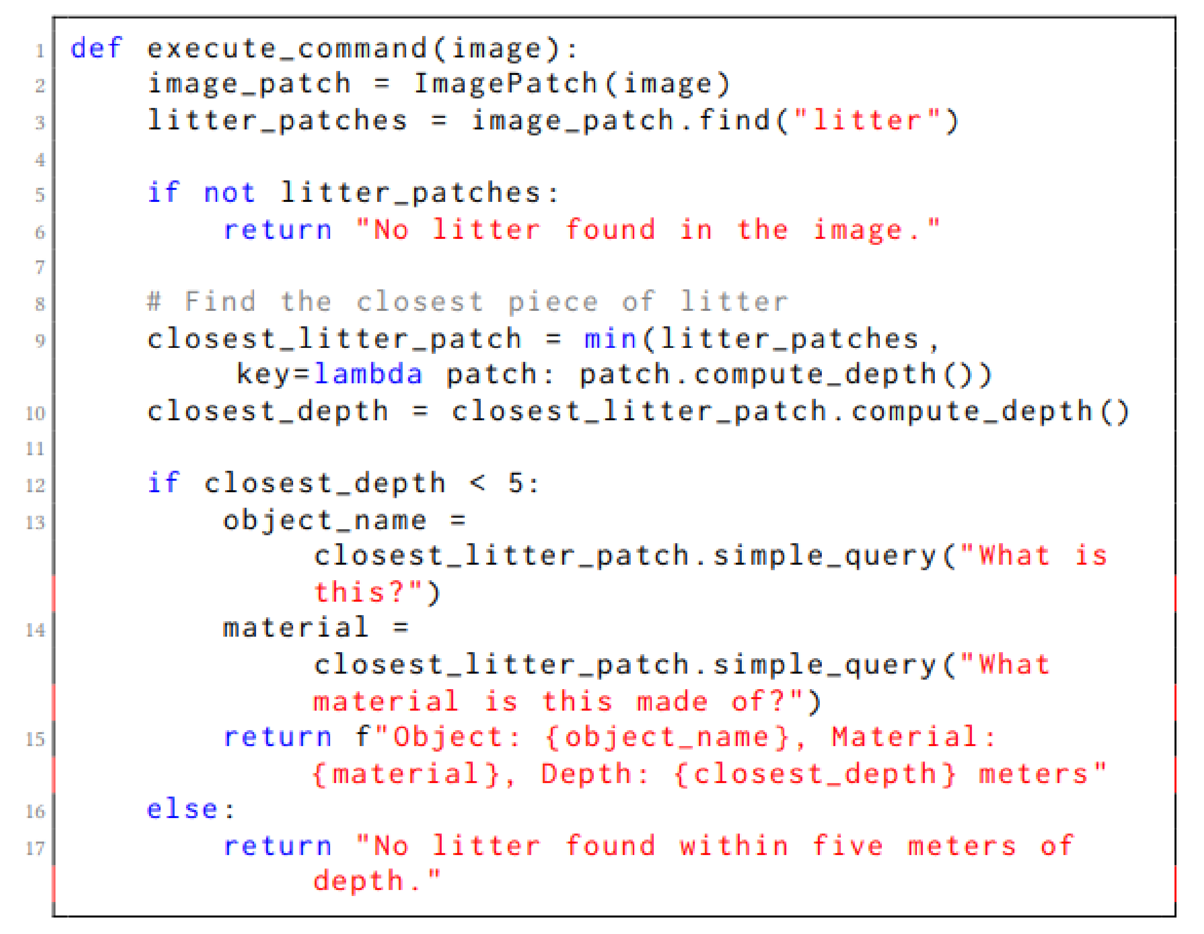
Example Serial code (for DiCE and DiCE-M)



Query: How many muffins can each kid have for it to be fair?

Generated Code

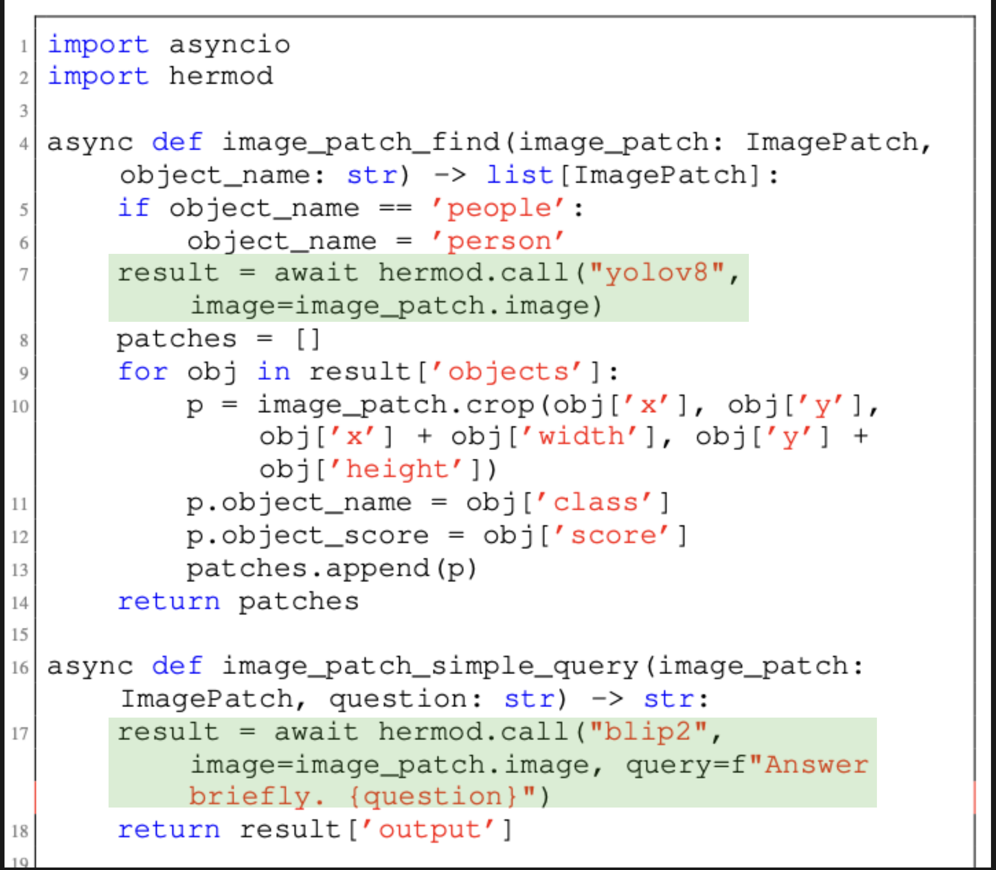
```
def execute_command(image):
    image_patch = ImagePatch(image)
    muffin_patches = image_patch.find("muffin")
    kid_patches = image_patch.find("kid")
    return str(len(muffin_patches) // len(kid_patches))
```



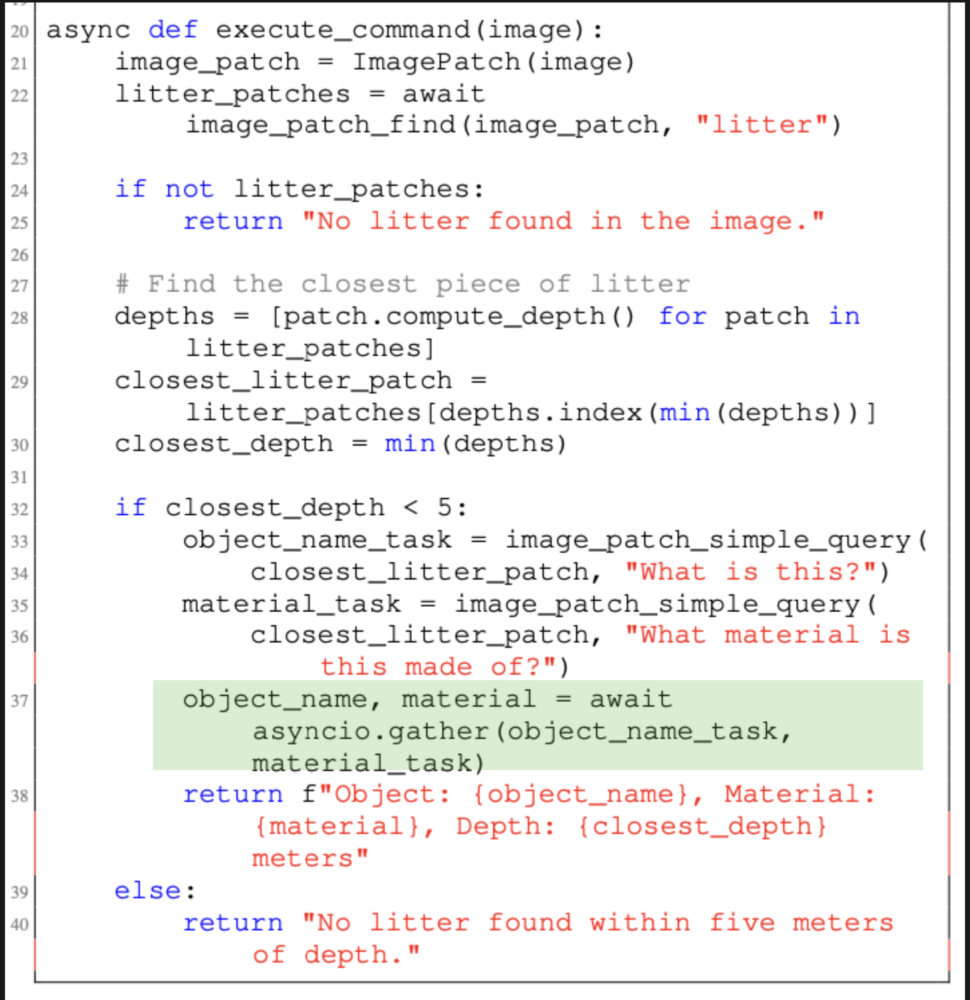
```
def execute_command(image):
    image_patch = ImagePatch(image)
    litter_patches = image_patch.find("litter")

    if not litter_patches:
        return "No litter found in the image."

    # Find the closest piece of litter
    closest_litter_patch = min(litter_patches,
         key=lambda patch: patch.compute_depth())
    closest_depth = closest_litter_patch.compute_depth()

    if closest_depth < 5:
        object_name =
             closest_litter_patch.simple_query("What is
             this?")
        material =
             closest_litter_patch.simple_query("What
             material is this made of?")
        return f"Object: {object_name}, Material:
             {material}, Depth: {closest_depth} meters"
    else:
        return "No litter found within five meters of
             depth."
```

DiCE and DiCE-M depend on ViperGPT and don't address cost implications

Example Distributed code for DiCE-M



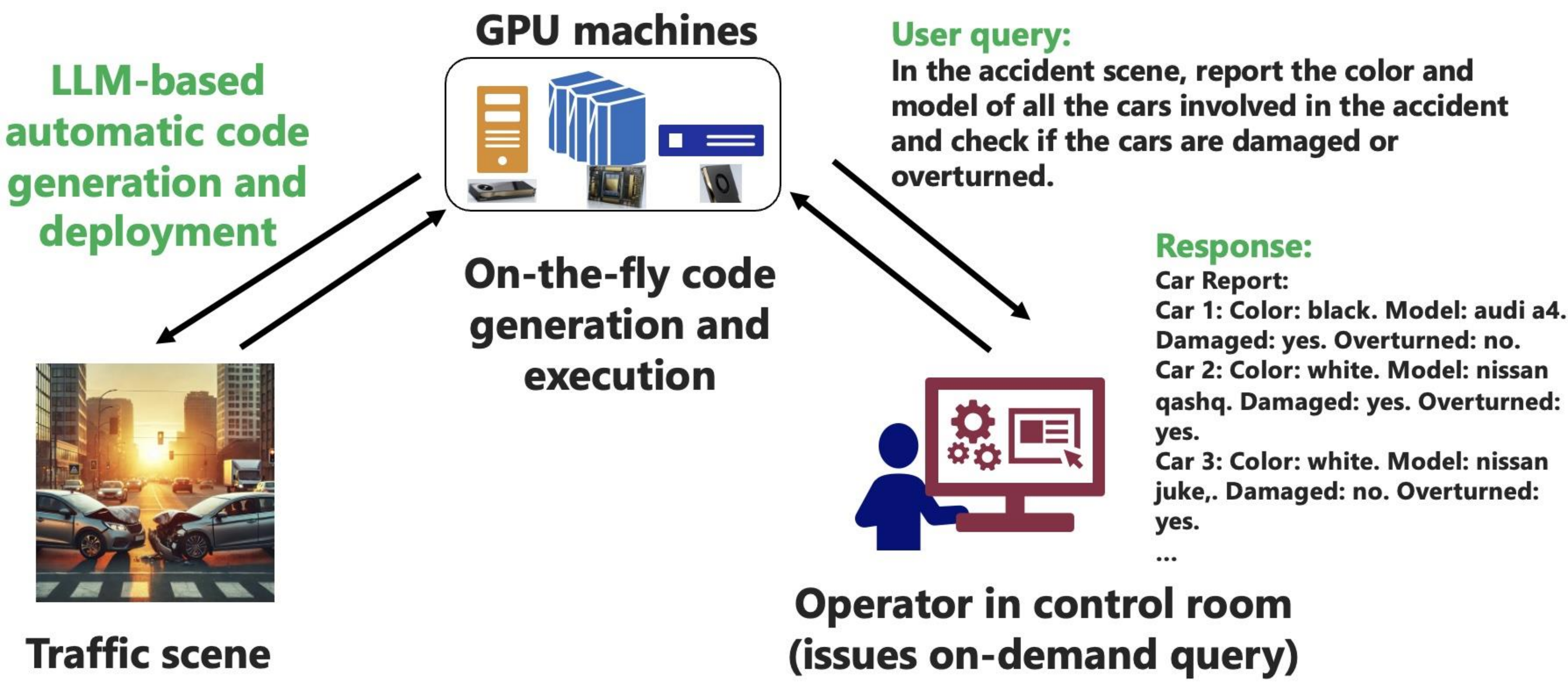
```
import asyncio
import hermod

async def image_patch_find(image_patch: ImagePatch,
    object_name: str) -> list[ImagePatch]:
    if object_name == ’people’:
        object_name = ’person’
    result = await hermod.call("yolov8",
        image=image_patch.image)
    patches = []
    for obj in result[’objects’]:
        p = image_patch.crop(obj[’x’], obj[’y’],
            obj[’x’] + obj[’width’], obj[’y’] +
            obj[’height’])
        p.object_name = obj[’class’]
        p.object_score = obj[’score’]
        patches.append(p)
    return patches

async def image_patch_simple_query(image_patch:
    ImagePatch, question: str) -> str:
    result = await hermod.call("blip2",
        image=image_patch.image, query=f"Answer
        briefly. {question}")
    return result[’output’]
```



```
async def execute_command(image):
    image_patch = ImagePatch(image)
    litter_patches = await
        image_patch_find(image_patch, "litter")

    if not litter_patches:
        return "No litter found in the image."

    # Find the closest piece of litter
    depths = [patch.compute_depth() for patch in
        litter_patches]
    closest_litter_patch =
        litter_patches[depths.index(min(depths))]
    closest_depth = min(depths)

    if closest_depth < 5:
        object_name_task = image_patch_simple_query(
            closest_litter_patch, "What is this?")
        material_task = image_patch_simple_query(
            closest_litter_patch, "What material is
                this made of?")
        object_name, material = await
            asyncio.gather(object_name_task,
            material_task)
        return f"Object: {object_name}, Material:
            {material}, Depth: {closest_depth}
            meters"
    else:
        return "No litter found within five meters
            of depth."
```

Example application for DiCE-C

Real-world insurance application (claims processing)

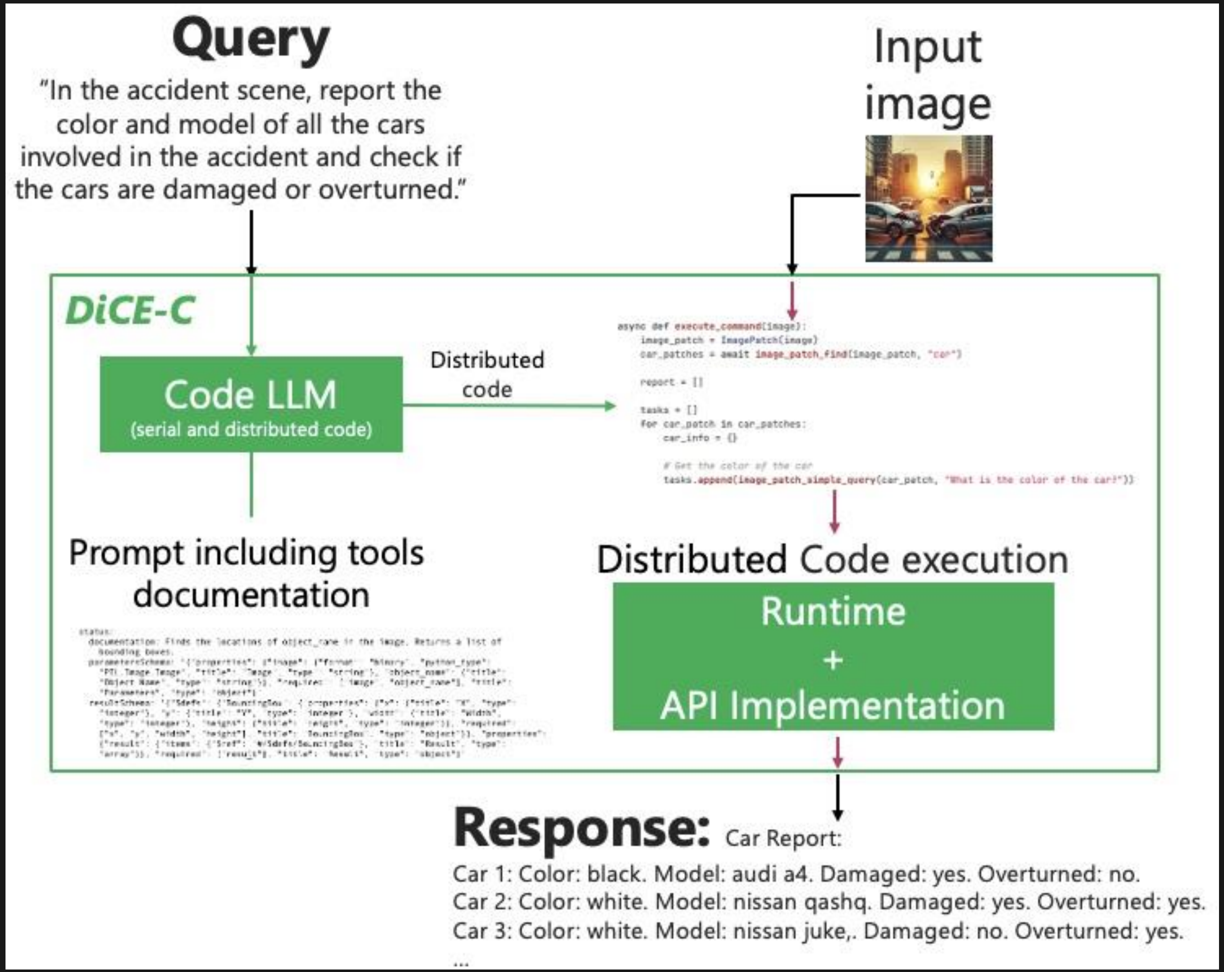


User query:
In the accident scene, report the color and model of all the cars involved in the accident and check if the cars are damaged or overturned.

Response:
Car Report:
Car 1: Color: black. Model: audi a4. Damaged: yes. Overturned: no.
Car 2: Color: white. Model: nissan qashq. Damaged: yes. Overturned: yes.
Car 3: Color: white. Model: nissan juke,. Damaged: no. Overturned: yes.
...

LLM can be leveraged to generate code on-the-fly to respond to user queries

DiCE-C System overview



"In the accident scene, report the color and model of all the cars involved in the accident and check if the cars are damaged or overturned."

```
async def execute_command(image):
    image_patch = ImagePatch(image)
    car_patches = await image_patch_find(image_patch, "car")

    report = []

    tasks = []
    for car_patch in car_patches:
        car_info = {}

        # Get the color of the car
        tasks.append(image_patch_simple_query(car_patch, "What is the color of the car?"))
```

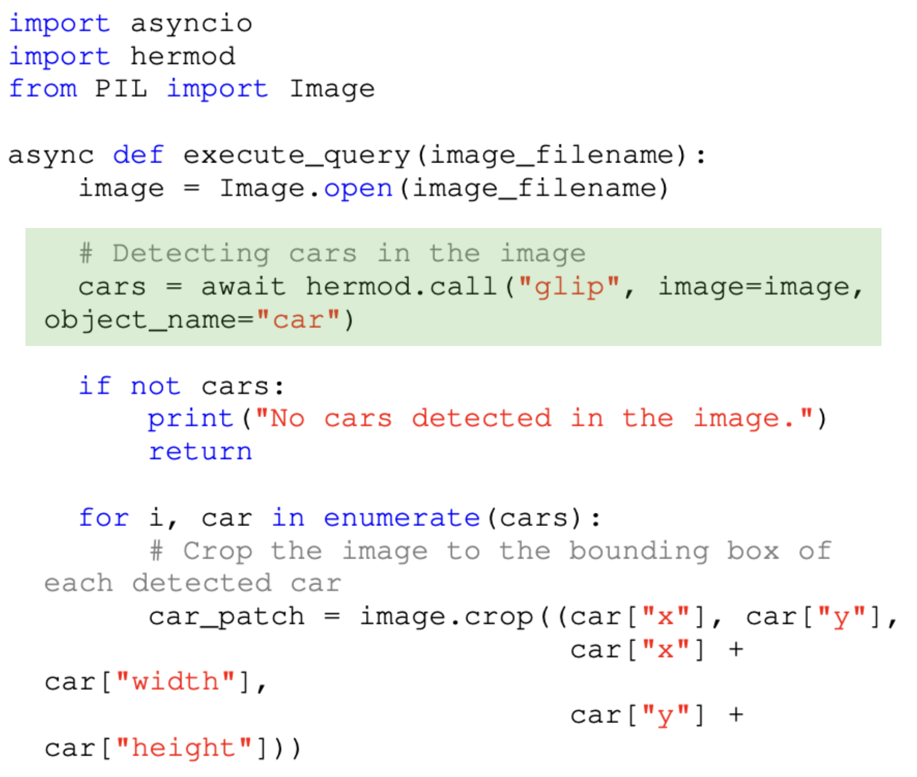
Car Report:
Car 1: Color: black. Model: audi a4. Damaged: yes. Overturned: no.
Car 2: Color: white. Model: nissan qashq. Damaged: yes. Overturned: yes.
Car 3: Color: white. Model: nissan juke,. Damaged: no. Overturned: yes.
...

DiCE-C is independent of ViperGPT and addresses cost implications

DiCE-C application code (1/3)

Query: In the accident scene, report the color and model of all the cars involved in the accident and check if the cars are damaged or overturned.

Initial Serial Code:



```
import asyncio
import hermod
from PIL import Image

async def execute_query(image_filename):
    image = Image.open(image_filename)

    # Detecting cars in the image
    cars = await hermod.call("glip", image=image,
  object_name="car")

    if not cars:
        print("No cars detected in the image.")
        return

    for i, car in enumerate(cars):
        # Crop the image to the bounding box of
  each detected car
        car_patch = image.crop((car["x"], car["y"],
                                car["x"] +
  car["width"],
                                car["y"] +
  car["height"]))
```

```
        # Query for the color of the car
        car_color = await hermod.call("blip",
  image=car_patch, question="What is the color of
  the car?")
        # Query for the model of the car
        car_model = await hermod.call("blip",
  image=car_patch, question="What is the model of
  the car?")
        # Check if the car is damaged
        car_damaged = await hermod.call("xvlm",
  image=car_patch, object_name="car",
  property="damaged")
        # Check if the car is overturned
        car_overturned = await hermod.call("xvlm",
  image=car_patch, object_name="car",
  property="overturned")

        # Compile the information
        car_info = f"Car {i+1}: Color -
  {car_color.get(’answer’, ’Unknown’)}, " \
                   f"Model -
  {car_model.get(’answer’, ’Unknown’)}, " \
                   f"Damaged -
  {car_damaged.get(’result’, False)}, " \
                   f"Overturned -
  {car_overturned.get(’result’, False)}"

        print(car_info)

image_filename = "accident_scene.jpg"
asyncio.run(execute_query(image_filename))
```

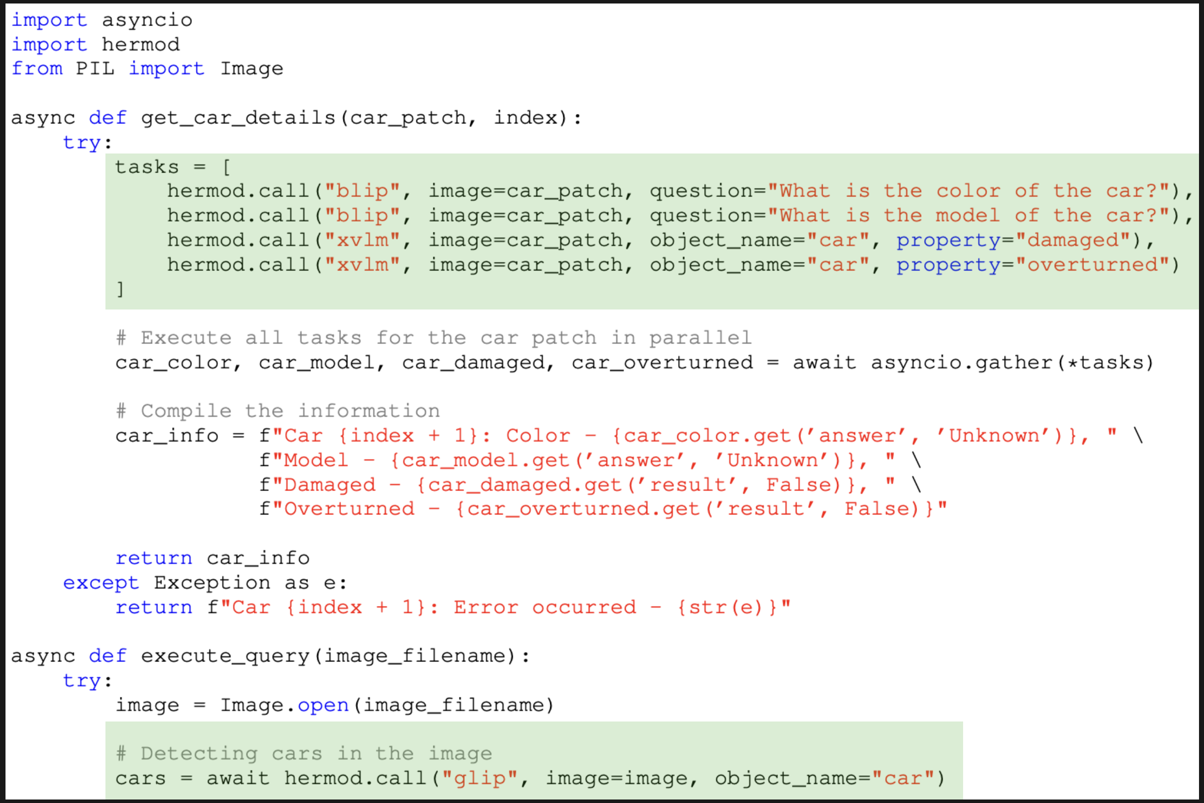
DiCE-C application code (2/3)

```
$ kubectl get functions glip -o yaml
apiVersion: hermod.nec-labs.com/v1
kind: Function
metadata:
    ...
status:
  documentation: Finds the locations of object_name in the image. Returns a list of
    bounding boxes.
  parametersSchema: '{"properties": {"image": {"format": "binary", "python_type":
    "PIL.Image.Image", "title": "Image", "type": "string"}, "object_name": {"title":
    "Object Name", "type": "string"}}, "required": ["image", "object_name"], "title":
    "Parameters", "type": "object"}'
  resultSchema: '{"$defs": {"BoundingBox": {"properties": {"x": {"title": "X", "type":
    "integer"}, "y": {"title": "Y", "type": "integer"}, "width": {"title": "Width",
    "type": "integer"}, "height": {"title": "Height", "type": "integer"}}, "required":
    ["x", "y", "width", "height"], "title": "BoundingBox", "type": "object"}}, "properties":
    {"result": {"items": {"$ref": "#/$defs/BoundingBox"}, "title": "Result", "type":
    "array"}}, "required": ["result"], "title": "Result", "type": "object"}'
```

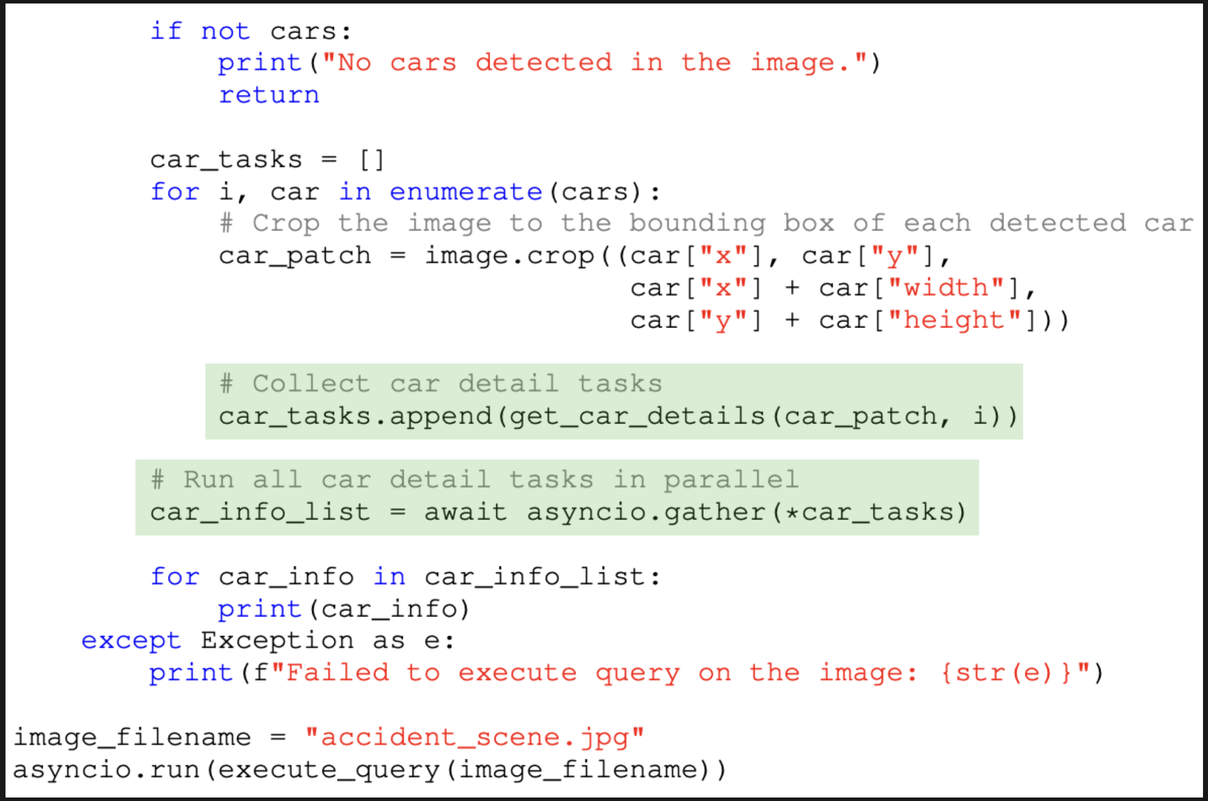
Sample function documentation and other details

DiCE-C application code (3/3)

Final Distributed Code

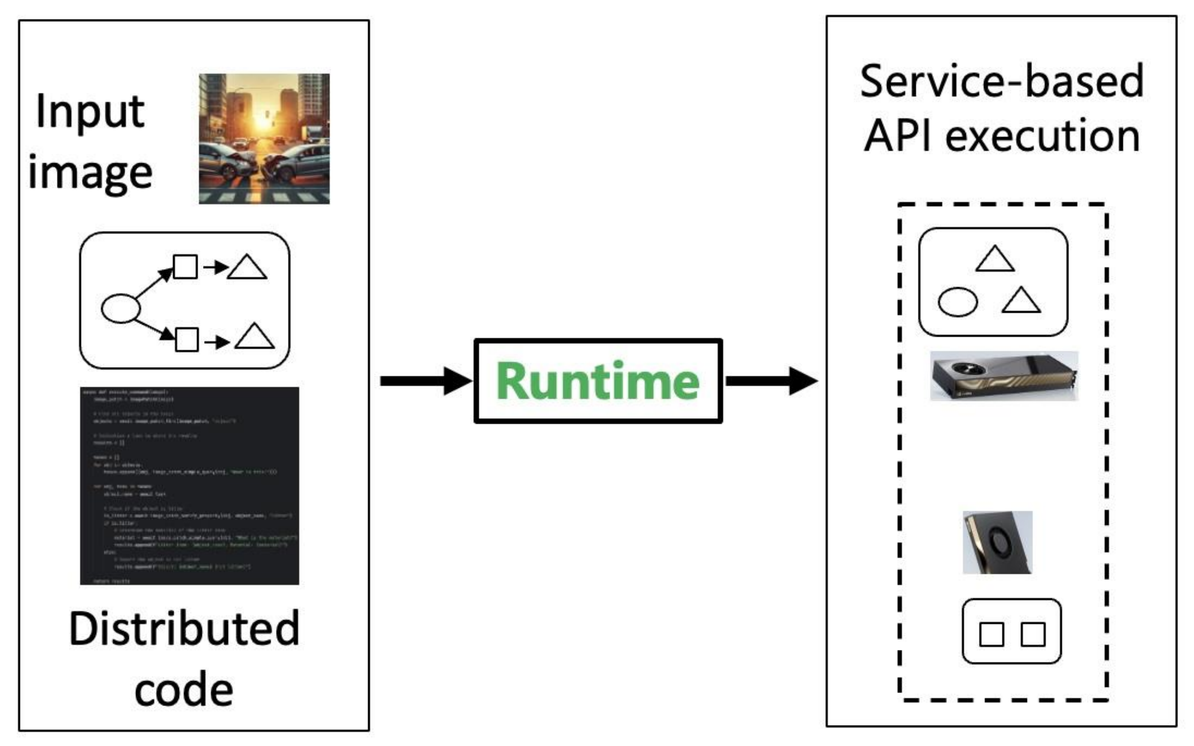


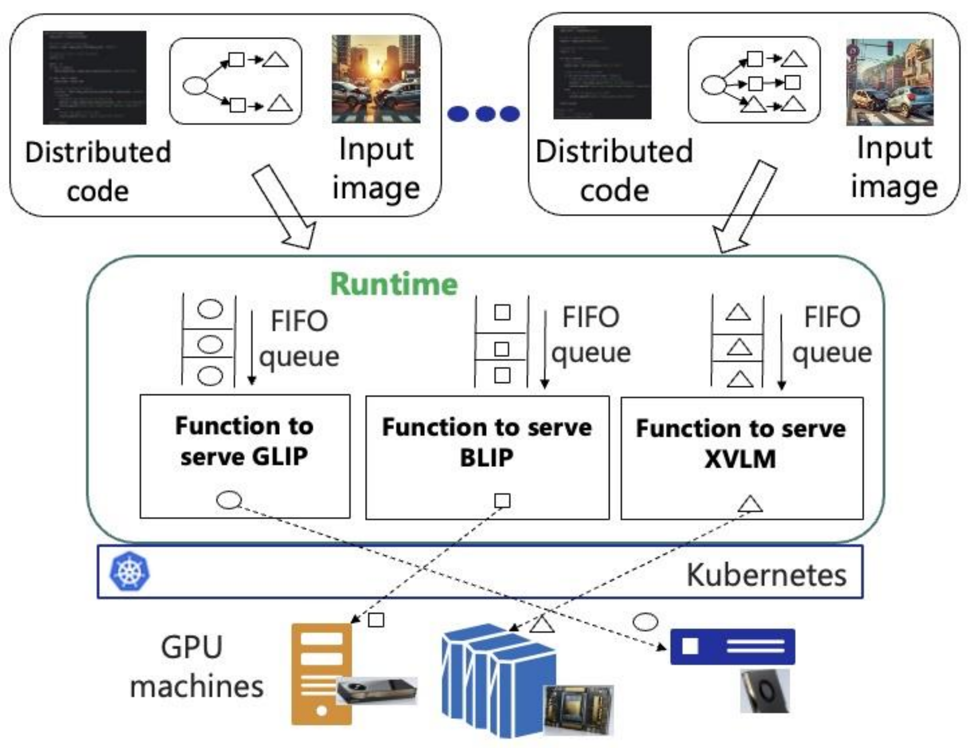
```python
import asyncio
import hermod
from PIL import Image

async def get_car_details(car_patch, index):
    try:
        tasks = [
            hermod.call("blip", image=car_patch, question="What is the color of the car?"),
            hermod.call("blip", image=car_patch, question="What is the model of the car?"),
            hermod.call("xvlm", image=car_patch, object_name="car", property="damaged"),
            hermod.call("xvlm", image=car_patch, object_name="car", property="overturned")
        ]

        # Execute all tasks for the car patch in parallel
        car_color, car_model, car_damaged, car_overturned = await asyncio.gather(*tasks)

        # Compile the information
        car_info = f"Car {index + 1}: Color - {car_color.get('answer', 'Unknown')}, " \
                   f"Model - {car_model.get('answer', 'Unknown')}, " \
                   f"Damaged - {car_damaged.get('result', False)}, " \
                   f"Overturned - {car_overturned.get('result', False)}"

        return car_info
    except Exception as e:
        return f"Car {index + 1}: Error occurred - {str(e)}"

async def execute_query(image_filename):
    try:
        image = Image.open(image_filename)

        # Detecting cars in the image
        cars = await hermod.call("glip", image=image, object_name="car")
```



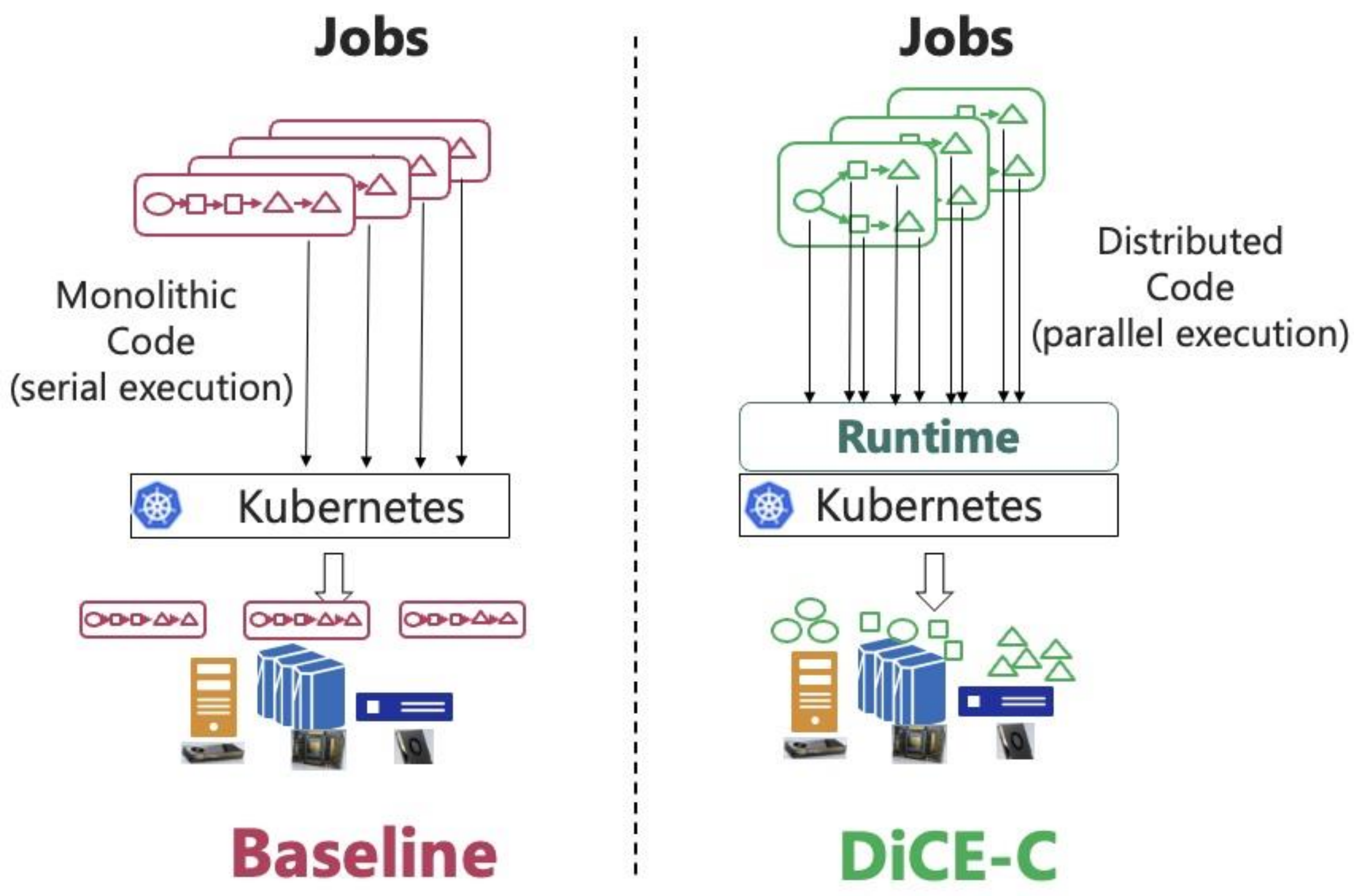
```python
        if not cars:
            print("No cars detected in the image.")
            return

        car_tasks = []
        for i, car in enumerate(cars):
            # Crop the image to the bounding box of each detected car
            car_patch = image.crop((car["x"], car["y"],
                                    car["x"] + car["width"],
                                    car["y"] + car["height"]))

            # Collect car detail tasks
            car_tasks.append(get_car_details(car_patch, i))

        # Run all car detail tasks in parallel
        car_info_list = await asyncio.gather(*car_tasks)

        for car_info in car_info_list:
            print(car_info)
    except Exception as e:
        print(f"Failed to execute query on the image: {str(e)}")

image_filename = "accident_scene.jpg"
asyncio.run(execute_query(image_filename))
```

Runtime for Distributed code execution





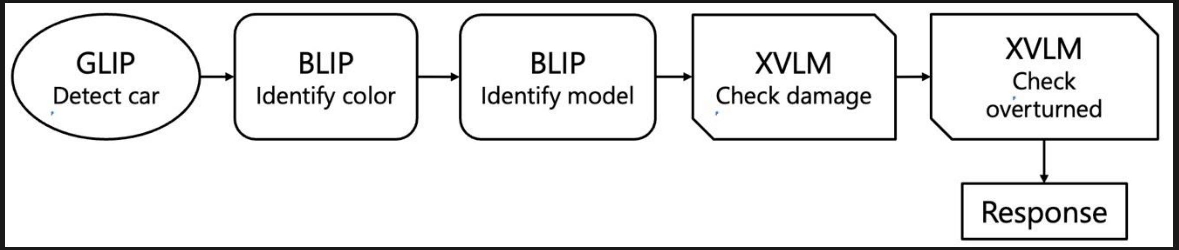
Baseline vs DiCE-C execution (1/2)



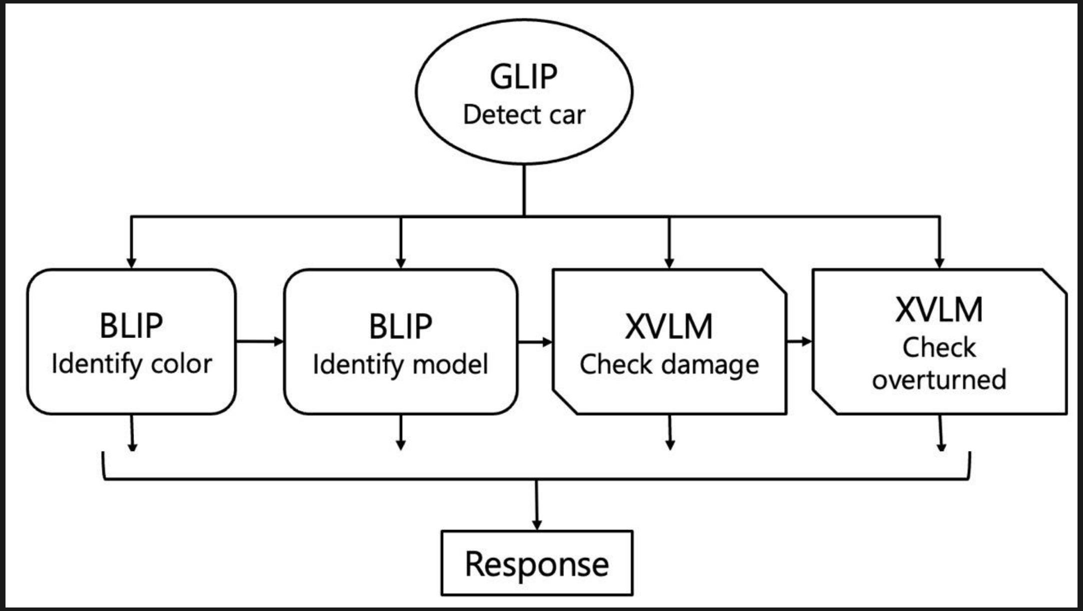
In baseline, entire pipeline runs on a single machine, while it is distributed in DiCE-C

Baseline vs DiCE-C execution (2/2)

Baseline

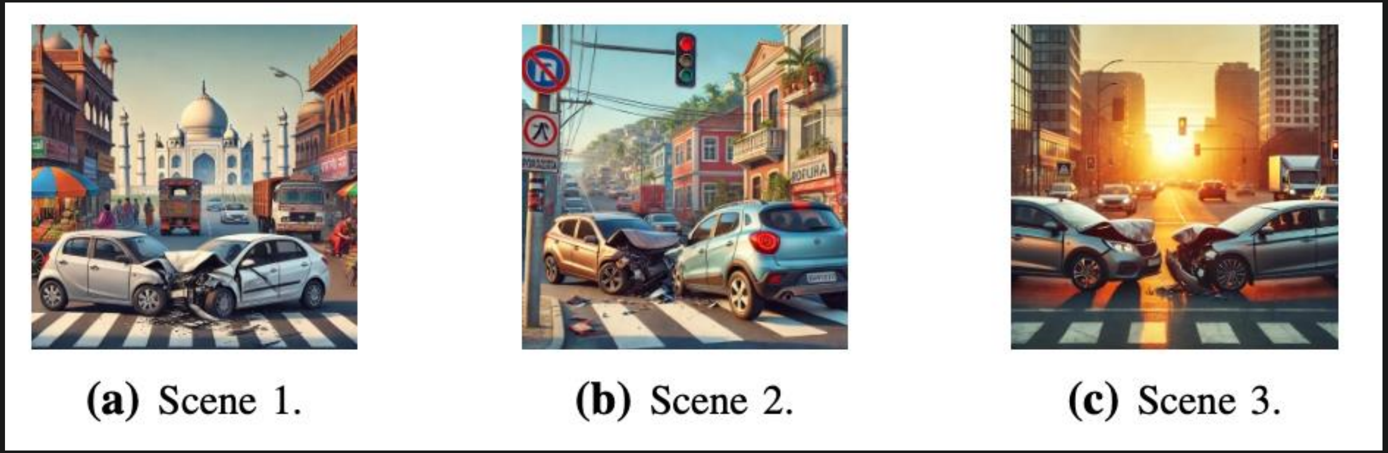


DiCE-C



Experimental Setup

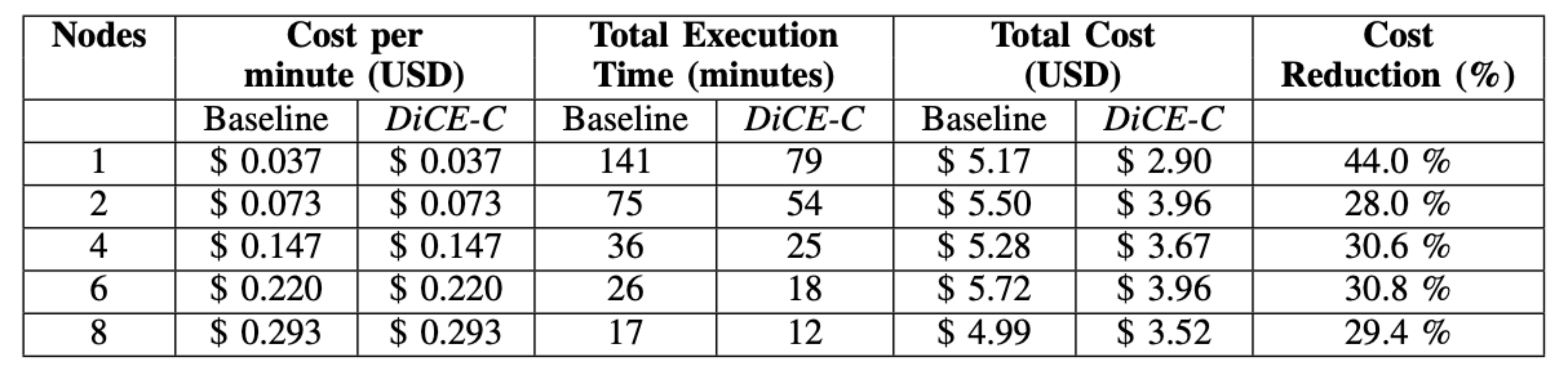
- Identical Hardware: Both the baseline and DiCE-C used A100 GPU nodes ($2.2/hour)
- Different Hardware: The baseline used A100 GPUs, while DiCE-C utilized a combination of A6000 and A4000 GPUs ($1.3/hour combined)
- We generated 30 accident scene images using GPT-4o
- Then, we replicated the final distributed code and created a batch of 1000 tasks, and randomly assigned the 30 images to these tasks


(a) Scene 1. (b) Scene 2. (c) Scene 3.

Experiments were run on Hyperstack cloud

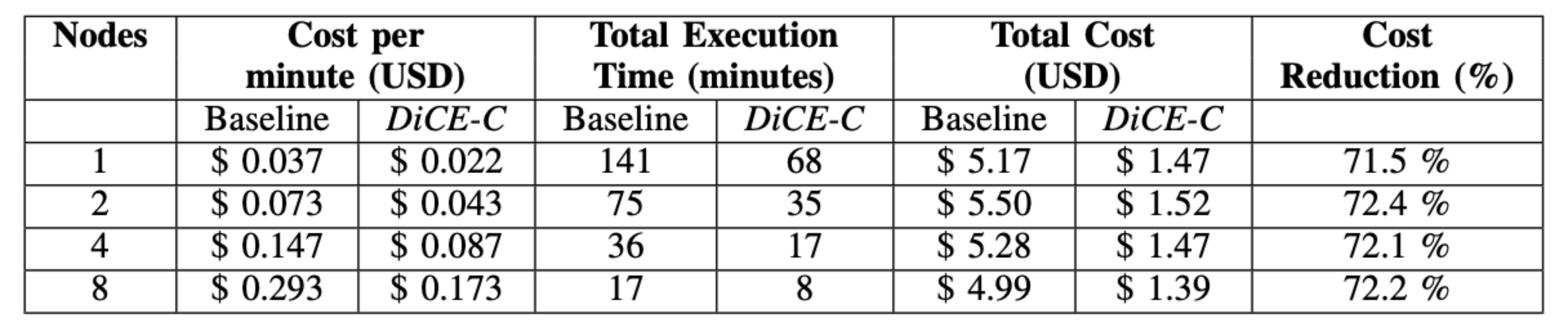
Results (1/2)

Identical Hardware



Nodes	Cost per minute (USD)		Total Execution Time (minutes)		Total Cost (USD)		Cost Reduction (%)
	Baseline	*DiCE-C*	Baseline	*DiCE-C*	Baseline	*DiCE-C*	
1	$ 0.037	$ 0.037	141	79	$ 5.17	$ 2.90	44.0 %
2	$ 0.073	$ 0.073	75	54	$ 5.50	$ 3.96	28.0 %
4	$ 0.147	$ 0.147	36	25	$ 5.28	$ 3.67	30.6 %
6	$ 0.220	$ 0.220	26	18	$ 5.72	$ 3.96	30.8 %
8	$ 0.293	$ 0.293	17	12	$ 4.99	$ 3.52	29.4 %

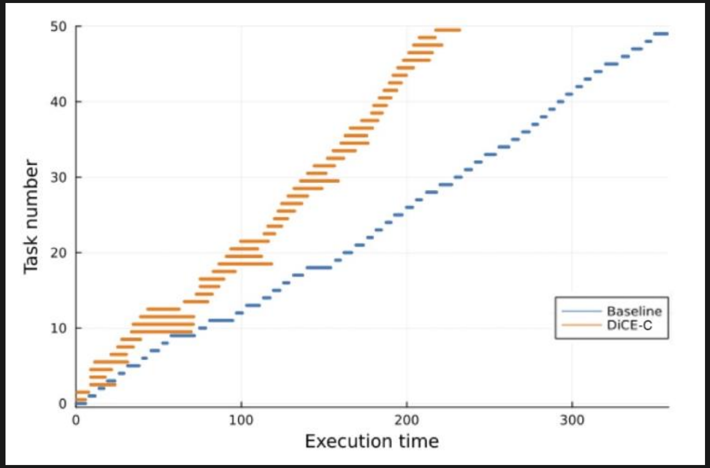
Different Hardware



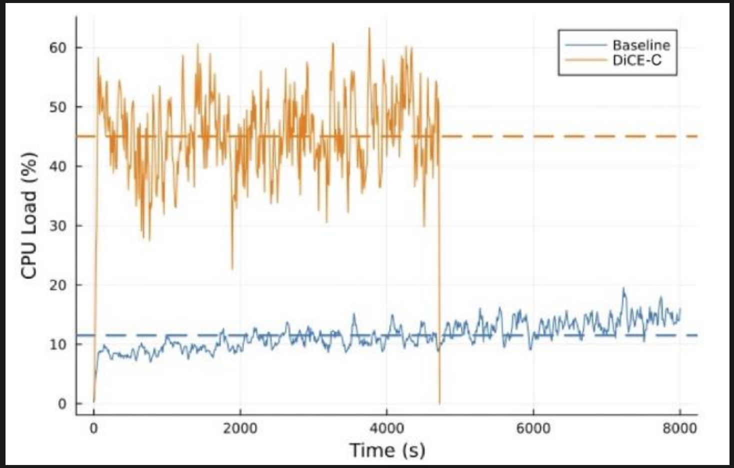
Nodes	Cost per minute (USD)		Total Execution Time (minutes)		Total Cost (USD)		Cost Reduction (%)
	Baseline	*DiCE-C*	Baseline	*DiCE-C*	Baseline	*DiCE-C*	
1	$ 0.037	$ 0.022	141	68	$ 5.17	$ 1.47	71.5 %
2	$ 0.073	$ 0.043	75	35	$ 5.50	$ 1.52	72.4 %
4	$ 0.147	$ 0.087	36	17	$ 5.28	$ 1.47	72.1 %
8	$ 0.293	$ 0.173	17	8	$ 4.99	$ 1.39	72.2 %

DiCE-C reduces costs by up to 72% for different hardware and by 32% for identical hardware

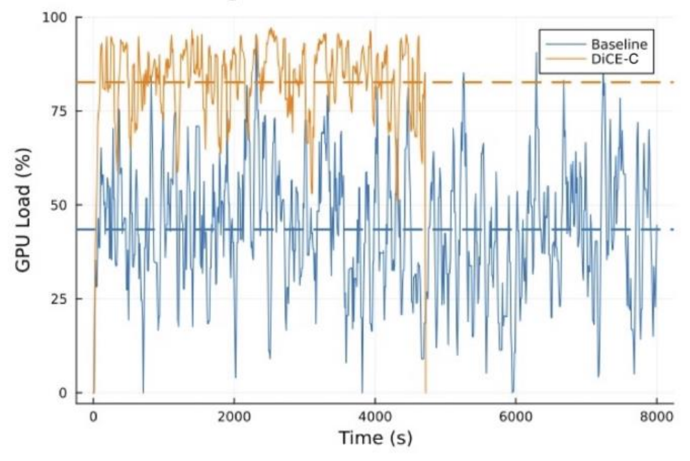
Results (2/2)



Execution pattern
(first 50 tasks)



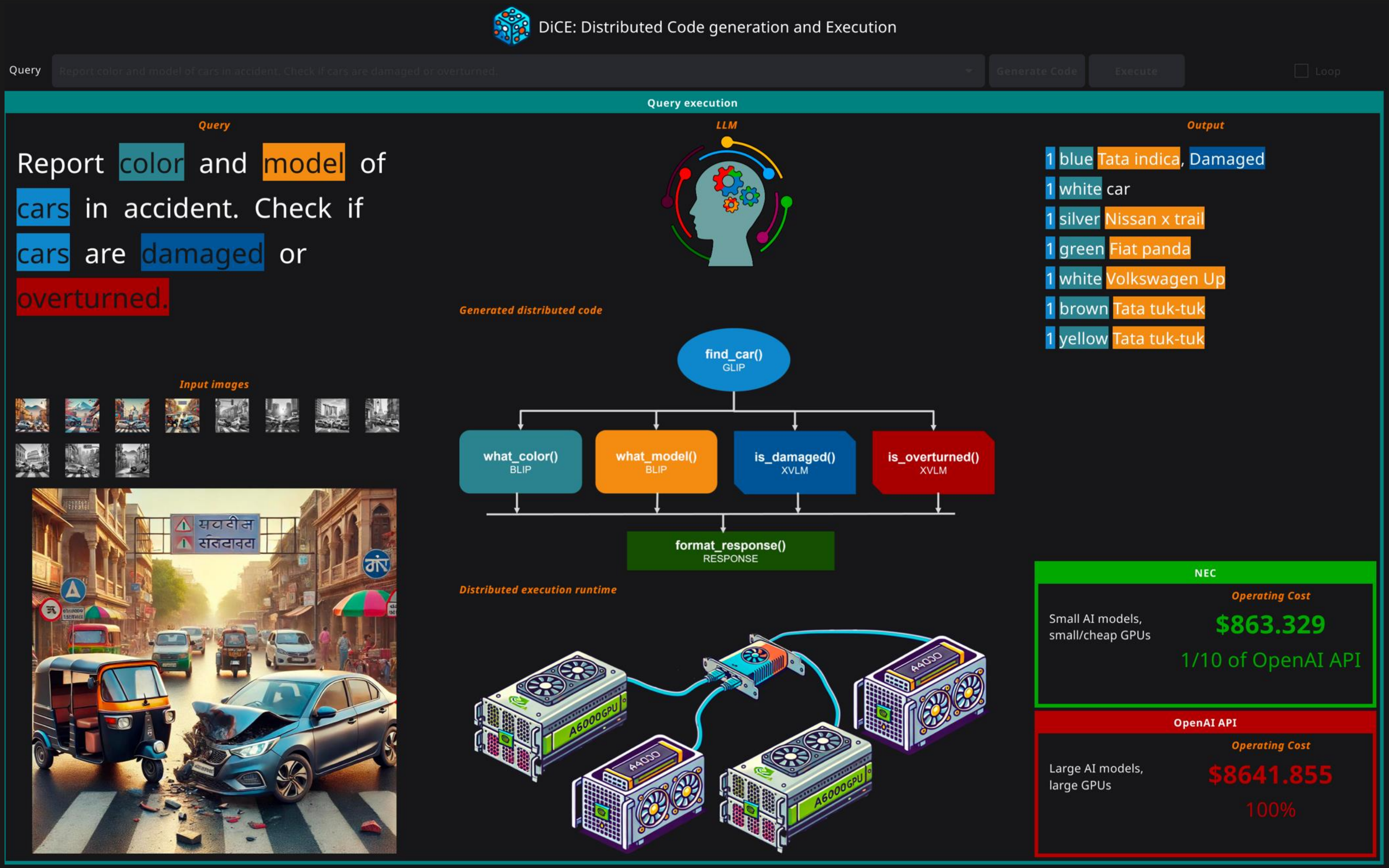
CPU Load



GPU Load

Execution occurs concurrently using DiCE-C and hardware utilization is high

Prototype system



DiCE: Distributed Code generation and Execution

Query Report color and model of cars in accident. Check if cars are damaged or overturned.

Query execution

Query

Report color and model of cars in accident. Check if cars are damaged or overturned.

Input images

LLM

Generated distributed code

Distributed execution runtime

Output

1 blue Tata indica, Damaged
1 white car
1 silver Nissan x trail
1 green Fiat panda
1 white Volkswagen Up
1 brown Tata tuk-tuk
1 yellow Tata tuk-tuk

NEC

	Operating Cost
Small AI models, small/cheap GPUs	$863.329 1/10 of OpenAI API

OpenAI API

	Operating Cost
Large AI models, large GPUs	$8641.855 100%

Summary

- Introduced DiCE-C, a cost-efficient system for deploying vision applications in cloud environments

- DiCE-C programmatically generates distributed code by leveraging runtime-exposed tool documentation

- DiCE-C reduces GPU idle time and supports the use of smaller, cost-efficient GPUs by dynamically managing API calls as independent services on Kubernetes

- Experimental evaluations on a real-world insurance application demonstrated that DiCE-C achieves an average cost reduction of 32% on identical GPU hardware and up to 72% when using smaller GPUs

Q&A

\Orchestrating a brighter world

NEC

NEC Laboratories **America**