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

Kunal Rao, Giuseppe Coviello, Gennaro Mellone, Ciro Giuseppe De Vita and Srimat Chakradhar
NEC Laboratories America, Inc., Princeton, NJ
email: {kunal, giuseppe.coviello, gmellone, cdevita, chak}@nec-labs.com

***Abstract*—The advancement of Generative Artificial Intelligence (AI), particularly Large Language Models (LLMs), is reshaping the software industry by automating code generation. Many LLM-driven distributed processing systems rely on serial code generation constrained by predefined libraries, limiting flexibility and adaptability. While some approaches enhance performance through parallel execution or optimize edge-cloud distributed processing for specific domains, they often overlook the cost implications of deployment, restricting scalability and economic feasibility across diverse cloud environments. This paper presents *DiCE-C*, a system that eliminates these constraints by starting directly from a natural language query. *DiCE-C* dynamically identifies available tools at runtime, programmatically refines LLM prompts, and employs a stepwise approach—first generating serial code and then transforming it into distributed code. This adaptive methodology enables efficient distributed execution without dependence on specific libraries. By leveraging high-level parallelism at the Application Programming Interface (API) level and managing API execution as services within a Kubernetes-based runtime, *DiCE-C* reduces idle GPU time and facilitates the use of smaller, cost-effective GPU instances. Experiments with a vision-based insurance application demonstrate that *DiCE-C* reduces cloud operational costs by up to 72% when using smaller GPUs (A6000 and A4000 GPU machines vs. A100 GPU machine) and by 32% when using identical GPUs (A100 GPU machines). This flexible and cost-efficient approach makes *DiCE-C* a scalable solution for deploying LLM-generated vision applications in cloud environments.**

***Keywords*-Cloud Computing; Large Language Models (LLMs); Distributed systems; Code generation; Cost reduction.**

I. Introduction

Advancements in Generative AI, particularly LLMs, are reshaping how vision applications are developed and deployed. Tools like ViperGPT [1] demonstrate how LLMs can generate application-specific vision code directly from natural language queries. For instance, users can issue queries, such as "detect traffic accidents" or "identify unattended objects", and ViperGPT automatically generates the required vision program. Figure 1 illustrates a scenario where an operator dynamically deploys such vision applications in real-time. While this represents a significant leap in automation and flexibility, deploying these applications in cloud environments often incurs substantial cost inefficiencies.

DiCE [2] and DiCE-M [3] are existing systems that take ViperGPT-generated serial code as their starting point and transform it into distributed code for parallel execution. While DiCE focuses on improving performance by exploiting API-level parallelism for faster execution of vision applications, DiCE-M targets marine applications using an edge and cloud approach to balance processing across distributed resources. However, neither system addresses the cost implications of deploying these applications, particularly in cloud settings. Additionally, both systems rely on the use of `image_patch` library, which exposes only a handful APIs. This limits the applicability to a very narrow set of applications. If additional functions are required for an application, then it cannot be built since `image_patch` does not have the necessary APIs.

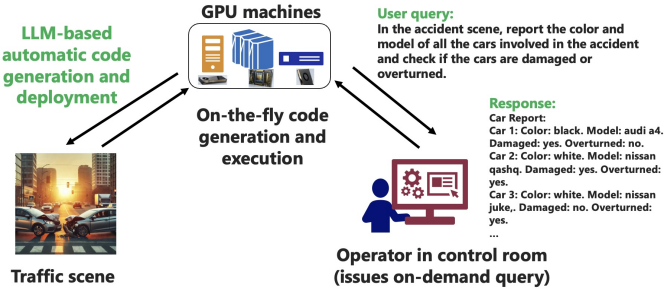

Figure 1: Use case scenario.

In this paper, we introduce *DiCE-C*, a system that removes the dependency on ViperGPT and the `image_patch` library. Unlike DiCE, *DiCE-C* starts with the original natural language query and dynamically discovers available tools through runtime APIs. It programmatically constructs prompts to guide an LLM to first generate serial code and then transform it into distributed code, leveraging a step-by-step approach that improves performance and accuracy. This flexibility enables *DiCE-C* to adapt to diverse workflows while focusing specifically on cost optimization in cloud environments.

After generating serial code, similar to DiCE, *DiCE-C* identifies high-level parallelism at the API level and transforms serial code into distributed code for efficient execution. By leveraging a Kubernetes-based runtime, *DiCE-C* dynamically manages API calls as services, allocating GPU resources only for the duration of individual service calls. This approach minimizes GPU idle time, allows the use of smaller, cost-efficient GPUs, and significantly lowers operational costs in cloud environments.

Our key contributions in this paper are:

- We identify the cost inefficiencies of deploying LLM-generated monolithic code in cloud environments, including GPU over-provisioning and under-utilization, and propose solutions to optimize resource allocation and execution to address these challenges.
- We introduce *DiCE-C*, which programmatically updates LLM prompts based on runtime API documentation to generate both serial and distributed code. This approach enhances flexibility, adaptability, and cost efficiency by leveraging dynamic tool discovery and resource optimiza-

tion.
- We demonstrate, using a real-world vision-based insurance application, that *DiCE-C* lowers cloud operational costs by up to 72% with smaller GPUs such as A6000 and A4000 GPU machines and by 32% with identical A100 GPU machines, both evaluated in the Hyperstack cloud.

The rest of the paper is organized as follows. Section 2 discusses related work. In Section 3, we examine the cost inefficiencies associated with deploying monolithic/serial code in cloud environments. Section 4 details the design and implementation of *DiCE-C*, focusing on how it generates serial code, transforms serial code into distributed code and then manages execution through a Kubernetes-based runtime. Section 5 reports experimental results highlighting the cost savings achieved by *DiCE-C* and showcases a prototype system. Finally, Section 6 concludes the paper.

II. Related Work

Optimizing cloud computing costs for AI-driven applications has garnered significant attention due to the increasing adoption of resource-intensive models in production environments. Systems like CloudScale [4], SpotDNN [5], SpotLake [6], Wang et. al [7], DEARS [8], ELASTIC [9], Saxena et. al [10], Ahmad et. al. [11], Alelyani et. al. [12] explore different strategies for efficient application execution in cloud computing environment. These works focus on leveraging spot instances, predictive scaling, and scheduling optimizations which can minimize operational expenses. However, they primarily target general-purpose workloads, whereas *DiCE-C* is specifically designed for LLM-generated vision applications, emphasizing API-level parallelism and distributed execution.

Kubernetes-based systems, such as Knative [13] and Kubeflow [14], provide platforms for scalable and efficient resource management. These frameworks offer primitives for deploying containerized workloads but lack the ability to dynamically adapt to the inherent parallelism of LLM-generated code. *DiCE-C* bridges this gap by integrating a runtime that transforms serial code into distributed code, dynamically managing API-level services, and optimizing GPU utilization, thereby reducing costs without requiring changes to the underlying Kubernetes infrastructure.

Recent advances in LLMs have contributed to code generation and parallelization. Tools like DSPy [15], AutoParLLM [16] and HPC-Coder [17] showcase the ability of LLMs to generate efficient pipelines and parallel programs. Systems like DiCE [2] leverage LLMs for transformation of serial code to distributed code for faster execution. DiCE-M [3] leverages LLMs to generate distributed code which can be executed in an edge + cloud infrastructure for marine applications. While these systems focus on improving performance or enabling edge + cloud execution, *DiCE-C* extends this paradigm to address cost optimization by integrating runtime-discovered tools and adapting workloads to varying cloud configurations.

Traditional compiler-based solutions like TVM [18] and Polyhedral [19] optimize program execution through low-level techniques such as memory layout transformations and loop optimizations. Although highly effective for individual tasks, these approaches are less applicable to the distributed, API-driven workloads targeted by *DiCE-C*. By focusing on high-level parallelism and runtime adaptability, *DiCE-C* complements such optimizations to address the unique challenges of cloud-based vision applications.

To the best of our knowledge, *DiCE-C* is the first system to integrate dynamic LLM-driven code generation with runtime cost optimization for vision applications, addressing the inefficiencies of monolithic code deployment and enabling scalable, cost-efficient execution in distributed cloud environments.

III. Motivation

Recent advancements in visual question answering benchmarks, such as RefCOCO, RefCOCO+ [20], GQA [21], OK-VQA [22], and NeXT-QA [23], have enabled tools like ViperGPT to synthesize visual programs directly from natural language queries using libraries such as `image_patch`. These tools showcase the capability of addressing real-world queries beyond benchmark datasets. Building on these advancements, *DiCE-C* eliminates the dependency on predefined libraries and dynamically generates serial code from user queries, enabling a wide range of applications, including the automation of complex, labor-intensive tasks like traffic accident reporting for insurance claim processing. For example, consider the query:

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.

Rather than relying on monolithic code generated by ViperGPT, *DiCE-C* dynamically constructs prompts based on documentation retrieved from the runtime API. These prompts guide an LLM to generate serial code, which is shown in Figure 2. The code utilizes AI models, such as `glip` [24], `blip` [25], and `xvlm` [26] to detect cars, extract their attributes, and evaluate their condition through API calls. Since these AI models require GPUs for execution, sequentially running the code on a single large GPU instance leads to inefficiencies, including extended idle times during CPU-bound operations and under-utilization of GPU resources for lightweight tasks.

This approach has substantial cost implications in cloud computing environments. Large GPU instances must often be provisioned to handle multiple AI models, even if a query only uses a subset of these models. Such instances are expensive to rent and may be unavailable during periods of high demand. Additionally, idle GPU time during CPU processing further inflates operational costs.

The code inherently offers opportunities for parallelism. After the initial `glip` API call to detect cars, subsequent API calls to query their properties (`blip` and `xvlm`) are independent and can be executed concurrently. This parallelism enables the use of smaller, more cost-effective GPU instances instead of relying on a single large GPU. Refactoring the code into a distributed format, where API calls are managed as independent services, allows for dynamic resource allocation,

```python
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))
```

Figure 2: Serial code generated by *DiCE-C* for an accident scene query.

reduces GPU idle time, and significantly lowers operational costs.

IV. Design and Implementation

Unlike existing systems such as DiCE [2] and DiCE-M [3], which rely on ViperGPT-generated monolithic code and are constrained by libraries like `image_patch`, *DiCE-C* removes these limitations. By leveraging runtime APIs to dynamically discover available tools and programmatically construct LLM prompts, *DiCE-C* enables flexible, cost-efficient deployments tailored to diverse workflows.

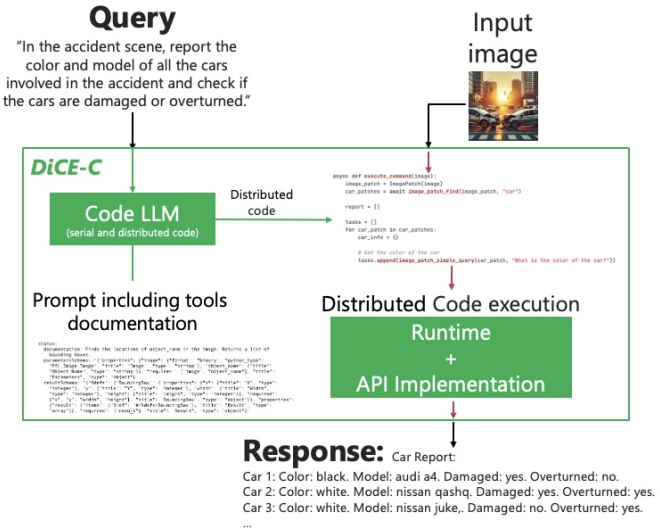

Figure 3: *DiCE-C* overview.

As shown in Figure 3, *DiCE-C* consists of two key components:

1) **Code Generation**: This includes the generation of serial code directly from natural language queries and its subsequent transformation into distributed code for parallel execution.
2) **Runtime**: A Kubernetes-based runtime that ensures cost-optimized deployment of distributed code by dynamically managing resources and minimizing GPU idle time.

A. Serial Code Generation

The first phase of *DiCE-C* involves generating functional serial code from a user-provided natural language query. This process begins by querying the runtime API to retrieve metadata about the available tools. Commands such as `kubectl get functions` and `kubectl get functions <function-name> -o yaml` provide detailed documentation, including the tool's functionality, input parameters, and output schemas. An example runtime API output for the `glip` function is shown in Figure 4.

This information is embedded into the LLM prompt alongside the natural language query. The prompt guides the LLM to generate serial code (in Python programming language) tailored to the tools available in the runtime. Figure 2 shows an example of serial code generated by *DiCE-C*. While functional, this sequential code may suffer from inefficiencies such as GPU idle time during CPU-bound operations and underutilization when lightweight tasks run on large GPUs, motivating the need for distributed code generation.

B. Distributed Code Generation

The second phase of *DiCE-C* transforms the serial code into distributed code by exploiting API-level parallelism. Figure 5 illustrates this process. Updated prompts are constructed using

```
$ 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"}'

```

Figure 4: GLIP function details obtained from runtime API.

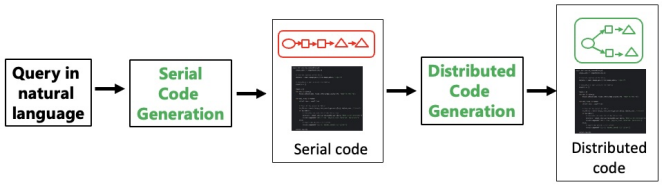

Figure 5: Distributed code generation overview.

information from the runtime API, which enables the LLM to identify independent API calls in the serial code and refactor them for concurrent execution. Additionally, *DiCE-C* incorporates elements from the prompt structure used in DiCE [2], adapting them to dynamically include runtime-discovered tools. This ensures that *DiCE-C* generates distributed code tailored to the specific environment.

Figure 6 shows the distributed code generated by *DiCE-C* using Python's *asyncio* library. By enabling concurrent execution of independent API calls, the distributed code utilizes multiple smaller GPU instances instead of relying on a single large GPU. This reduces idle GPU time, improves resource utilization, and lowers operational costs.

C. Runtime for Distributed Code Execution

The distributed code generated by *DiCE-C* is executed within a Kubernetes-based runtime (Figure 7). Each API call is treated as an independent service, allowing dynamic allocation of GPU resources. By reserving GPUs only for the duration of individual service calls, the runtime minimizes idle time and enables cost-efficient execution.

The runtime architecture, shown in Figure 8, processes requests through service queues, where each API call is transparently mapped to a GPU running the associated AI model. This design facilitates efficient resource sharing across multiple workloads, further reducing the cost of cloud-based deployments. By handling workloads dynamically, the runtime ensures scalable and adaptable execution of vision applications.

D. Cost-Optimized Execution in DiCE-C

Figure 9 illustrates the contrast between the baseline monolithic execution and *DiCE-C*'s distributed execution. In the baseline, all AI models are loaded onto a single large GPU, which remains reserved for the entire job duration. Conversely, *DiCE-C* transforms the code into a distributed version, enabling API calls to run as independent services on smaller GPUs. This minimizes idle GPU time and allows for dynamic scaling based on workload demands, resulting in substantial cost savings.

By combining serial and distributed code generation with a runtime, *DiCE-C* ensures scalable and cost-efficient deployment of vision applications in cloud environments.

V. Experimental Setup and Results

In this section, we evaluate the cost savings achieved by *DiCE-C* compared to the baseline execution of monolithic code in cloud computing environments. The experiments are based on the real-world insurance application use case described in Section III, where the task is to generate detailed accident reports from images. We validate the correctness of the distributed code generated by *DiCE-C* by manually comparing its output with that of the serial code generated by ViperGPT, and both outputs matched.

Figure 6 shows the distributed code generated by *DiCE-C* for the insurance application use case. The code introduces concurrency using Python's *asyncio* library, enabling independent API calls (e.g., querying color, model, damage status, and overturned status) to execute in parallel. This distributed execution reduces GPU idle time and allows dynamic resource allocation for servicing the API calls, thereby achieving substantial cost savings.

A. Experimental Setup

To evaluate cost efficiency, we created a batch of 1000 vision programs (tasks) by replicating the distributed code shown in Figure 6. Thirty different accident scene images were generated using OpenAI's "GPT-4o" model, and these images

```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")

        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))
```

Figure 6: Distributed code generated by *DiCE-C*.

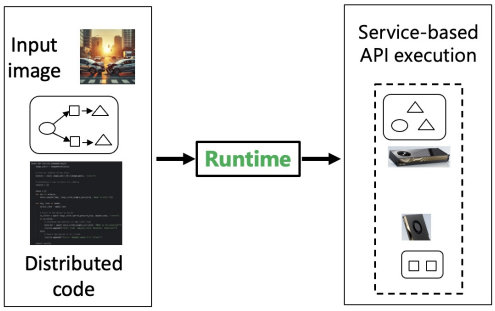

Figure 7: Runtime overview.

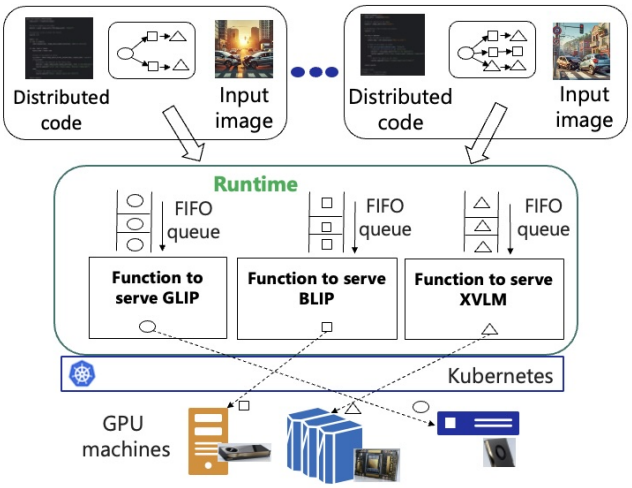

Figure 8: Runtime architecture.

were randomly assigned across the 1000 tasks. Figure 12 shows three sample images used in the experiments.

The experiments were conducted using machines in the Hyperstack [27] cloud under two configurations:

- **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).

Figures 10 and 11 illustrate the execution patterns of the

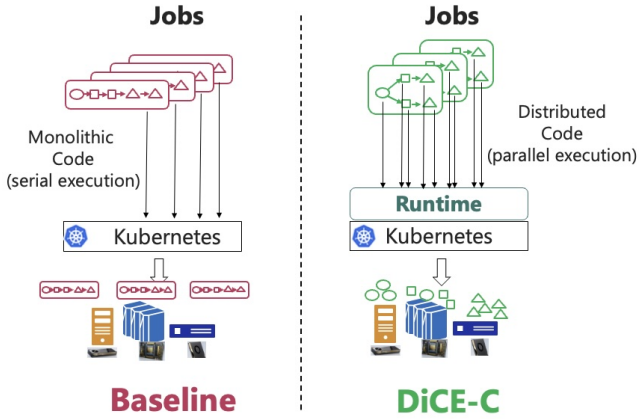

Figure 9: Execution in baseline vs *DiCE-C*.

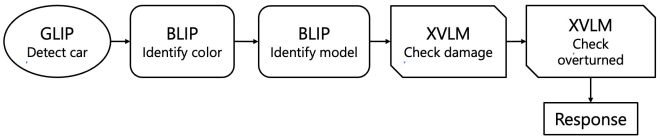

Figure 10: Execution of AI models in baseline.

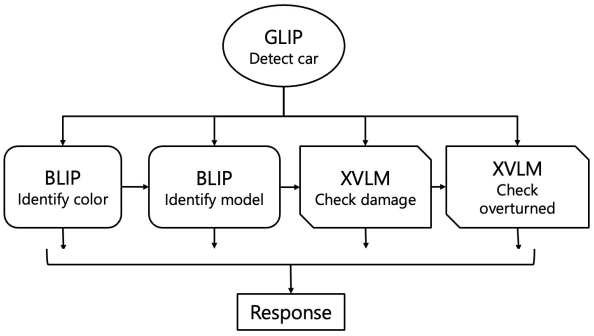

Figure 11: Parallel execution of AI models in *DiCE-C*.

baseline and *DiCE-C*, respectively. In the baseline, API calls are executed sequentially, whereas in *DiCE-C*, after the initial API call to detect cars (using `glip`), the remaining API calls (`blip`, `xvlm`) are executed concurrently. This parallel execution allows *DiCE-C* to minimize GPU idle time and optimize cloud resource usage.

B. Using Identical Hardware

We perform experiments using configurations of 1, 2, 4, 6, and 8 identical nodes. In the case of baseline, only 1 task runs on a machine at a time, whereas in the case of *DiCE-C*, we allow 16 tasks to run simultaneously. Table I shows the total execution time in each case and the corresponding

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

Figure 12: Sample accident scenes.

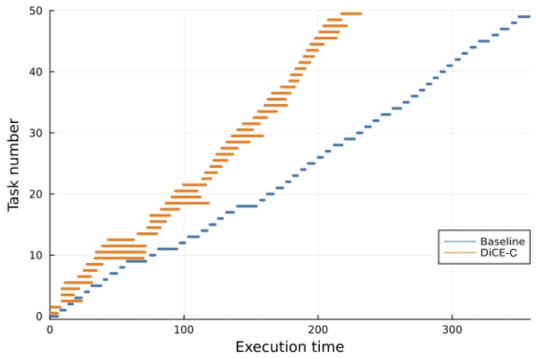

Figure 13: Execution pattern of first 50 tasks.

TABLE I: COST REDUCTION (USING 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 %

TABLE II: COST REDUCTION (USING 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 %

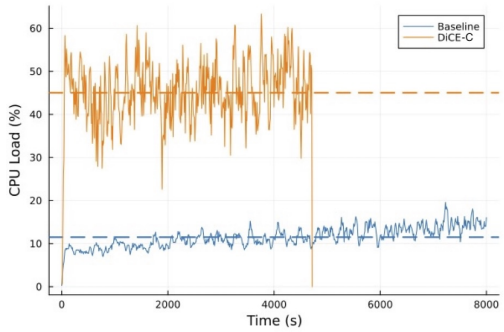

Figure 14: CPU Load.

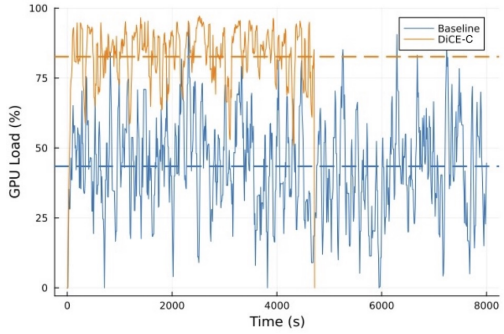

Figure 15: GPU Load.

cost incurred to complete the execution of a batch of 1000 tasks. We observe that on average *DiCE-C* saves up to 32 % operating cost.

We further dive deeper to understand how the execution goes in each case. Figure 13 shows the execution pattern for the first 50 tasks when running on a single node. We observe that in the case of baseline, the tasks start one after the other, as expected, since only 1 job runs at a time and the latency for execution (shown by horizontal line length) varies depending on the image that is used. In case of *DiCE-C*, we clearly see the overlap in execution across different tasks and note that there is a slight increase in latency for each task due to the contention of resources on the same hardware.

Overall, the execution for the batch of tasks goes faster using *DiCE-C* compared to the baseline (higher throughput),

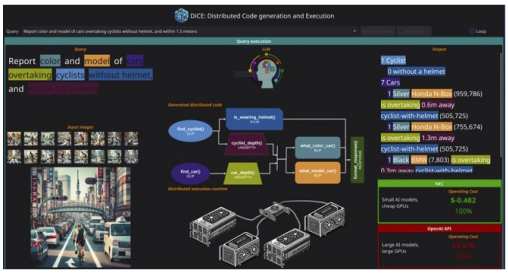

Figure 16: Prototype system for *DiCE-C*.

and this directly results in cost savings, since the machines in the cloud are used for less time. Figures 14 and 15 show the CPU and GPU load, respectively, when the batch of 1000 tasks is run on a single node. We observe that due to increased utilization, the load is higher in the case of *DiCE-C*, leading to faster completion of tasks.

C. Using Different Hardware

Table II shows the total execution time and cost reduction when using cheaper and smaller GPUs in *DiCE-C*. We observe that *DiCE-C* achieves an average 72 % reduction in operating cost by using smaller and cheaper GPUs. This highlights *DiCE-C*'s adaptability and efficiency in cloud computing environments.

D. Prototype system

Figure 16 shows a prototype system for *DiCE-C*. In this system, the user can write a query in natural language and behind the scenes, *DiCE-C* initially generates the serial code and then transforms this code into a distributed code version, which is then executed on a Kubernetes-based distributed cluster. In the user interface, we show the different AI models that are being used and the execution flow in the baseline vs *DiCE-C* case. In addition, we also show the cost savings achieved by using *DiCE-C*.

VI. Conclusion and Future Work

In this paper, we introduced *DiCE-C*, a cost-efficient system for deploying vision applications in cloud environments. Unlike prior approaches that rely on monolithic code generated

by tools like ViperGPT, *DiCE-C* programmatically generates distributed code by leveraging runtime-exposed tool documentation. By dynamically managing API calls as independent services on Kubernetes, *DiCE-C* reduces GPU idle time and supports the use of smaller, cost-efficient GPUs, significantly lowering operational expenses while preserving the correctness and functionality of the original application.

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. Although this work focuses on vision applications, our future work involves incorporating additional functions and tools support in the runtime, so that a wide range of applications and workloads can be supported and enabled by *DiCE-C*.

REFERENCES

[1] D. Surís, S. Menon, and C. Vondrick, "ViperGPT: Visual inference via python execution for reasoning," in *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023, pp. 11 854–11 864. DOI: 10.1109/ICCV51070.2023.01092.

[2] K. Rao, G. Coviello, and S. Chakradhar, "DiCE: Distributed Code Generation and Execution," in *2024 IEEE Conference on Pervasive and Intelligent Computing (PICom)*, 2024, pp. 8–15. DOI: 10.1109/PICom64201.2024.00008.

[3] G. Coviello, K. Rao, G. Mellone, C. G. De Vita, and S. Chakradhar, "DiCE-M: Distributed Code Generation and Execution for Marine Applications - An Edge-Cloud Approach," in *2024 IEEE/ACM Symposium on Edge Computing (SEC)*, 2024, pp. 468–475. DOI: 10.1109/SEC62691.2024.00054.

[4] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: Elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11, Cascais, Portugal: Association for Computing Machinery, 2011, ISBN: 9781450309769. DOI: 10.1145/2038916.2038921.

[5] R. Shang *et al.*, "SpotDNN: Provisioning Spot Instances for Predictable Distributed DNN Training in the Cloud," in *2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*, 2023, pp. 1–10. DOI: 10.1109/IWQoS57198.2023.10188717.

[6] S. Lee, J. Hwang, and K. Lee, *SpotLake: Diverse Spot Instance Dataset Archive Service*, 2022. arXiv: 2202.02973 [cs.DC].

[7] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang, "Exploiting Spot and Burstable Instances for Improving the Cost-efficacy of In-Memory Caches on the Public Cloud," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, Belgrade, Serbia: Association for Computing Machinery, 2017, pp. 620–634, ISBN: 9781450349383. DOI: 10.1145/3064176.3064220.

[8] M. Hassan, H. Chen, and Y. Liu, "DEARS: A deep learning based elastic and automatic resource scheduling framework for cloud applications," in *2018 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Ubiquitous Computing and Communications, Big Data and Cloud Computing, Social Computing and Networking, Sustainable Computing and Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, 2018, pp. 541–548. DOI: 10.1109/BDCloud.2018.00086.

[9] Y. Li *et al.*, "ELASTIC: Edge workload forecasting based on collaborative cloud-edge deep learning," in *Proceedings of the ACM Web Conference 2023*, ser. WWW '23, Austin, TX, USA: Association for Computing Machinery, 2023, pp. 3056–3066, ISBN: 9781450394161. DOI: 10.1145/3543507.3583436.

[10] D. Saxena and A. K. Singh, *Workload forecasting and resource management models based on machine learning for cloud computing environments*, 2021. arXiv: 2106.15112 [cs.DC].

[11] S. G. Ahmad, T. Iqbal, E. U. Munir, and N. Ramzan, "Cost optimization in cloud environment based on task deadline," *J. Cloud Comput.*, vol. 12, no. 1, Jan. 2023, ISSN: 2192-113X. DOI: 10.1186/s13677-022-00370-x.

[12] A. Alelyani, A. Datta, and G. M. Hassan, "Optimizing Cloud Performance: A Microservice Scheduling Strategy for Enhanced Fault-Tolerance, Reduced Network Traffic, and Lower Latency," *IEEE Access*, vol. 12, pp. 35 135–35 153, 2024. DOI: 10.1109/ACCESS.2024.3373316.

[13] Knative Community, *Knative: Kubernetes-based platform to deploy and manage modern serverless workloads*, https://knative.dev/, Accessed 2025-03-04, 2024.

[14] Kubeflow Community, *Kubeflow: The machine learning toolkit for kubernetes*, https://www.kubeflow.org/, Accessed 2025-03-04, 2025.

[15] O. Khattab *et al.*, "DSPy: Compiling declarative language model calls into self-improving pipelines," *arXiv preprint arXiv:2310.03714*, 2023.

[16] Q. I. Mahmud, A. TehraniJamsaz, H. D. Phan, N. K. Ahmed, and A. Jannesari, *Autoparllm: Gnn-guided automatic code parallelization using large language models*, 2023. arXiv: 2310.04047 [cs.LG].

[17] D. Nichols, A. Marathe, H. Menon, T. Gamblin, and A. Bhatele, "HPC-Coder: Modeling parallel programs using large language models," in *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, 2024, pp. 1–12. DOI: 10.23919/ISC.2024.10528929.

[18] T. Chen *et al.*, *Tvm: An automated end-to-end optimizing compiler for deep learning*, 2018. arXiv: 1802.04799 [cs.LG].

[19] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 101–113, ISBN: 9781595938602. DOI: 10.1145/1375581.1375595.

[20] L. Yu, P. Poirson, S. Yang, A. C. Berg, and T. L. Berg, *Modeling context in referring expressions*, 2016. arXiv: 1608.00272 [cs.CV].

[21] D. A. Hudson and C. D. Manning, *Gqa: A new dataset for real-world visual reasoning and compositional question answering*, 2019. arXiv: 1902.09506 [cs.CL].

[22] K. Marino, M. Rastegari, A. Farhadi, and R. Mottaghi, *Ok-vqa: A visual question answering benchmark requiring external knowledge*, 2019. arXiv: 1906.00067 [cs.CV].

[23] J. Xiao, X. Shang, A. Yao, and T.-S. Chua, *Next-qa:next phase of question-answering to explaining temporal actions*, 2021. arXiv: 2105.08276 [cs.CV].

[24] L. H. Li *et al.*, *Grounded language-image pre-training*, 2022. arXiv: 2112.03857 [cs.CV].

[25] J. Li, D. Li, S. Savarese, and S. Hoi, *Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models*, 2023. arXiv: 2301.12597 [cs.CV].

[26] Y. Zeng *et al.*, "$X^2$2-vlm: All-in-one pre-trained model for vision-language tasks," *IEEE Transactions on Pattern Analysis amp; Machine Intelligence*, vol. 46, no. 05, pp. 3156–3168, May 2024, ISSN: 1939-3539. DOI: 10.1109/TPAMI.2023.3339661.

[27] Hyperstack, *Hyperstack*, https://www.hyperstack.cloud/, Accessed 2025-03-04.