symmetry

MDPI

Article

Optimizing Microservice Deployment in Edge Computing with Large Language Models: Integrating Retrieval Augmented Generation and Chain of Thought Techniques

Kan Feng [1], Lijun Luo [1], Yongjun Xia [2], Bin Luo [2], Xingfeng He [1], Kaihong Li [3], Zhiyong Zha [4], Bo Xu [1,5] and Kai Peng [1,*]

[1] Hubei Key Laboratory of Smart Internet Technology, School of Electronic Information and Communications, Huazhong University of Science and Technology, Wuhan 430074, China; fengkan19850505@sina.com (K.F.); lijunluo@hust.edu.cn (L.L.); hexingfeng77@126.com (X.H.); xubosite@hust.edu.cn (B.X.)
[2] Hubei Huazhong Electric Power Technology Development Co., Ltd., Wuhan 430079, China; xiayj6@hb.sgcc.com.cn (Y.X.); luobin1425@163.com (B.L.)
[3] Electronic Information School, Wuhan University, Wuhan 430072, China; likaihong1245@163.com
[4] State Grid Information Telecommunication Co., Ltd., Wuhan 430048, China; hustboyzzy@163.com
[5] Hubei ChuTianYun Co., Ltd., Wuhan 430076, China
* Correspondence: pkhust@hust.edu.cn

Abstract: Large Language Models (LLMs) have demonstrated impressive capabilities in autogenerating code based on natural language instructions provided by humans. We observed that in the microservice models of edge computing, the problem of deployment latency optimization can be transformed into an NP-hard mathematical optimization problem. However, in the real world, deployment strategies at the edge often require immediate updates, while human-engineered code tends to be lagging. To bridge this gap, we innovatively integrated LLMs into the decision-making process for microservice deployment. Initially, we constructed a private Retrieval Augmented Generation (RAG) database containing prior knowledge. Subsequently, we employed meticulously designed step-by-step inductive instructions and used the chain of thought (CoT) technique to enable the LLM to learn, reason, reflect, and regenerate. We decomposed the microservice deployment latency optimization problem into a collection of granular sub-problems (described in natural language), progressively providing instructions to the fine-tuned LLM to generate corresponding code blocks. The generated code blocks underwent integration and consistency assessment. Additionally, we prompted the LLM to generate code without the use of the RAG database for comparative analysis. We executed the aforementioned code and comparison algorithm under identical operational environments and simulation parameters, conducting rigorous result analysis. Our fine-tuned model significantly reduced latencies by 22.8% in handling surges in request flows, 37.8% in managing complex microservice types, and 39.5% in processing increased network nodes compared to traditional algorithms. Moreover, our approach demonstrated marked improvements in latency performance over LLMs not utilizing RAG technology and reinforcement learning algorithms reported in other literature. The use of LLMs also highlights the concept of symmetry, as the symmetrical structure of input-output relationships in microservice deployment models aligns with the LLM's inherent ability to process and generate balanced and optimized code. Symmetry in this context allows for more efficient resource allocation and reduces redundant operations, further enhancing the model's effectiveness. We believe that LLMs hold substantial potential in optimizing microservice deployment models.

Keywords: large language models; retrieval augmented generation; microservice deployment; mobile edge computing

Citation: Feng, K.; Luo, L.; Xia, Y.; Luo, B.; He, X.; Li, K.; Zha, Z.; Xu, B.; Peng, K. Optimizing Microservice Deployment in Edge Computing with Large Language Models: Integrating Retrieval Augmented Generation and Chain of Thought Techniques. *Symmetry* **2024**, *16*, 1470. https://doi.org/10.3390/sym16111470

Academic Editor: Theodore E. Simos

Received: 14 September 2024
Revised: 26 October 2024
Accepted: 27 October 2024
Published: 5 November 2024

1. Introduction

Contemporary trends reveal an accelerating adoption of microservices, signifying a transformative approach in architecture and organization within Mobile Edge Computing (MEC). Essentially, microservices can be conceptualized as a collection of fine-grained, autonomous services that interact through well-defined interfaces [1]. These systems are characterized by their loose coupling and the capacity for independent deployment, progressively displacing traditional monolithic applications. The utilization of microservices architecture (MSA) is notably on the rise, particularly in large-scale distributed systems. Prominent corporations including (Netflix (https://about.netflix.com/ (accessed on 29 October 2024), Amazon (https://aws.amazon.com/) (accessed on 29 October 2024), and X (formerly Twitter (https://x.com/) (accessed on 29 October 2024))) have implemented this architectural model [2]. Consequently, numerous cloud service providers, such as Amazon Web Services (AWS) and Azure, have extensively integrated microservices to facilitate a variety of real-time cloud applications. Within these cloud data centers (http://dubbo.apache.org/ (accessed on 29 October 2024), http://springcloud.cc/ (accessed on 29 October 2024), latency-sensitive applications handle immense volumes of user requests per second, distributed across multiple queues and processed by various microservice instances. These requests are managed by an ensemble of microservices that engage in frequent recursive calls and communications, forming intricate microservice call graphs [3]. Furthermore, microservice containers, even when of the same type, are capable of serving distinct user requests, thereby enabling multi-instance and geographically dispersed deployments.

Contrary to conventional virtual network functions or cloud data center application scenarios, applications within MEC networks must navigate the limitations imposed by scarce computational resources. As a result, microservices within MEC architectures are structured to be autonomous and streamlined. Ordinarily, the handling of user requests is distributed among several microservices that are often recycled and engage in continuous interactions, thus managing a substantial influx of requests and creating an elaborate service invocation graph [4]. Considering the complex data interdependencies that characterize microservices, the overarching efficacy of expansive MEC applications is contingent upon meticulous service deployment and request routing strategies. This optimization entails the identification of optimal locations and capacities for deploying microservices. As microservice instances process user requests, these may traverse singular or multiple routing trajectories. The effectiveness of service deployment is influenced by the immediate scheduling and routing of requests across service instances, whereas the viability of request routing is predicated on the presence of an operational service instance. Although refining deployment strategies may alleviate the latency involved in processing user requests, an exclusive focus on routing might curtail the latency associated with request transmission [5–7]. However, targeted optimization tactics could elevate the risk of service failures, especially within MEC setups sensitive to latency [8]. In such scenarios, the simultaneous refinement of microservice deployment and request routing proves crucial in augmenting service performance.

Large Language Models (LLMs), particularly those based on the Transformer architecture like OpenAI's ChatGPT series (Generative Pre-trained Transformer) models, have been gaining significant popularity [9]. These models are characterized by their vast scale, encompassing billions to trillions of parameters [10], and are trained on extensive textual datasets using numerous GPUs. Research indicates that transformative capabilities only manifest in models with parameters exceeding 6.2 billion, and ChatGPT boasts approximately 175 billion parameters [11]. Beyond engaging in context-aware dialogues over multiple exchanges, LLMs excel in tasks such as information extraction, content generation, code creation, automatic summarization, and translation. These capabilities renew optimism in the feasibility of general artificial intelligence and catalyze breakthroughs in human-machine interaction and collaboration. Noteworthy implementations of LLMs in the code generation sector include Microsoft's CodeBERT [12], Facebook's CodeLlama [13], Deep-

Mind's AlphaCode [14], and OpenAI's GPT-4 (https://openai.com/index/openai-codex/ (accessed on 29 October 2024)), which also shows immense potential in code generation.

Our research centers on the GPT series models, which are at the forefront of current advancements. Utilizing a context learning mechanism and Reinforcement Learning from Human Feedback (RLHF), ChatGPT [15] exhibits adaptability across a diverse array of downstream tasks, demonstrating robust capabilities in natural language understanding and generation within low-resource and zero-shot scenarios. RLHF ensures that the outputs of the model align with human common sense, cognition, and values—critical factors in the domain of code generation where understanding and mastering computer language form the foundation for creating novel code. However, given the sparsity of reward signals and the necessity to explore a vast, structured space of potential programs, code generation presents unique challenges. Correct solutions can vary greatly, and assessing the usefulness of partial or incorrect solutions is particularly daunting—a mere single character change can radically alter a program's behavior.

This paper presents a novel approach to integrating large language models with edge computing. Initially, full-parameter fine-tuning of large-scale models, despite their enhanced reasoning capabilities, demands substantial computational resources, rendering them unsuitable for edge deployments. Alternatively, parameter compression may lead to smaller models that lack the full capabilities of their more extensive counterparts. Moreover, optimizing microservice deployment involves complex, multi-parameter dynamic optimization challenges, often necessitating lengthy periods to develop effective human-designed algorithms. Lastly, the reasoning and integration capabilities of large-scale language models align seamlessly with the 'Model as a Service (MAAS)' concept within microservice architecture. Our overall process framework is depicted in Figure 1. In summary, this document highlights our principal contributions:

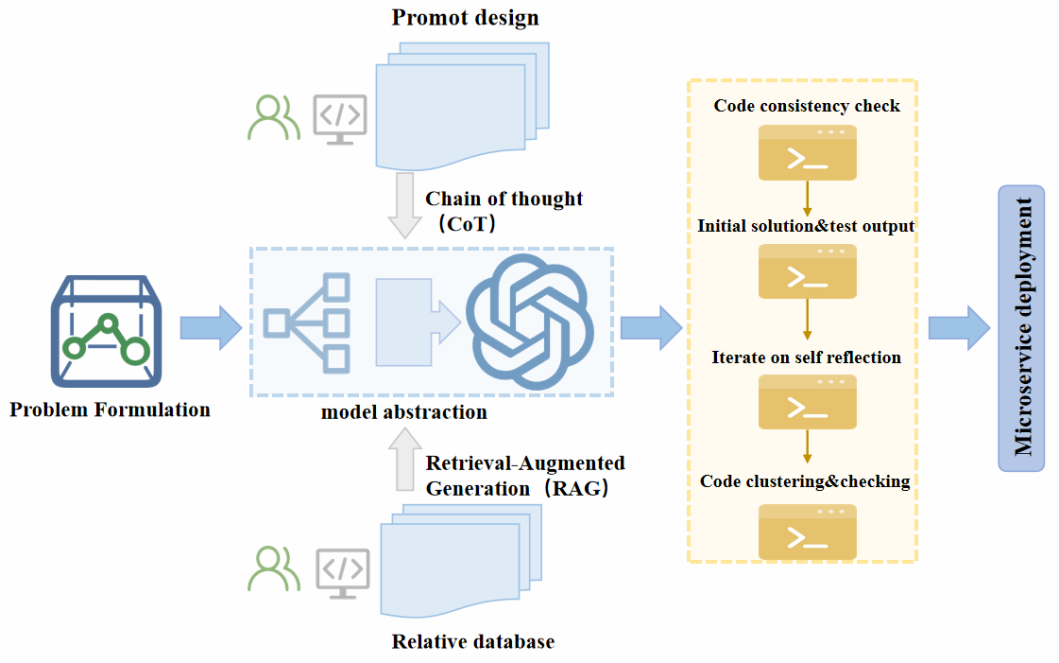

Figure 1. Our Large language Model Assisted Microservice Deployment framework workflow.

- We established a proprietary database populated with prior knowledge, which was systematically formatted into standard question-answer pairs to facilitate effective learning by large models. This enhancement substantially improved the retrieval-generation process, bolstering the model's capability to reason through specific problems.
- We leveraged a finely-tuned large model to generate code within a deep reinforcement learning framework, aimed at addressing a practical deployment issue. This issue

was abstracted into a Mixed-Integer Linear Programming (MILP) problem, where we achieved a locally optimal solution.
- We conducted manual evaluations of the generated code and engaged in comparative analyses against algorithms documented in the related literature. By standardizing environment variables and simulation parameters, We executed rigorous experimental comparisons and provided detailed analytical insights.

The general framework of the article is as follows. Section 2 discusses the related work, Section 3 introduces the mathematical models and techniques applied, Section 4 describes the experimental setup and procedures in detail, Section 5 evaluates the experimental results, and Section 6 concludes the study.

2. Background and Related Works

2.1. Large Language Models

Large Language Models (LLMs) have demonstrated extraordinary capabilities in Natural Language Processing (NLP). Such emergent abilities include in-context learning,instruction following,step-by-step reasoning and etc. Currently, LLMs are having a profound impact on the AI community. The emergence of models like ChatGPT and GPT-4 has prompted a reevaluation of the possibilities surrounding artificial general intelligence (AGI). OpenAI has published a technical article titled "Planning for AGI and Beyond", (https://openai.com/index/planning-for-agi-and-beyond/ (accessed on 29 October 2024)) which outlines both short-term and long-term strategies for approaching AGI. More recently, Literature [16] has suggested that GPT-4 might be considered an early version of an AGI system.

In the realm of NLP, LLMs function as versatile solvers for various language tasks, marking a shift in research paradigms towards their utilization. In the field of Information Retrieval (IR), traditional search engines are being challenged by AI chatbots, such as ChatGPT, which offer a new approach to information seeking [17]. New Bing (https://www.bing.com/new (accessed on 29 October 2024)) represents an initial attempt to enhance search results using LLMs. In Computer Vision (CV), researchers are striving to develop ChatGPT-like vision-language models that facilitate more effective multi-modal dialogues, with GPT-4 already supporting multi-modal input by integrating visual information [18].

This technological advancement holds the potential to create a flourishing ecosystem of real-world applications based on LLMs. Despite their progress and impact, the fundamental principles behind LLMs remain largely unexplored. The reasons why emergent abilities appear in LLMs, but not in smaller pre-trained language models (PLMs), are still a mystery. More generally, there is a lack of in-depth investigation into the key factors contributing to the superior capabilities of LLMs. Understanding when and how LLMs acquire such abilities is crucial for further advancements in the field.

2.2. Fine Tuning of LLMs

Instruction Dataset: Instruction tuning delineates the procedure of additional training of LLMs using a corpus composed of <INSTRUCTION, OUTPUT> pairings in a regimented manner. Each entry within an instruction corpus encapsulates three components: an instruction, which is a sequence of natural language text delineating the task; an optional input offering ancillary context; and a projected output contingent upon the instruction. Predominantly, two methodologies exist for the compilation of instruction datasets. Resources such as Dolly [19] and Public Pool of Prompts [20] are formulated from data harvested from extant annotated linguistic datasets. An alternative modality involves the creation of outputs through the utilization of LLMs, exemplified by InstructWild (https://github.com/XueFuzhao/InstructionWild (accessed on 29 October 2024)) and Self-Instruct [21]. For dialogic open sources within IT datasets that entail multi-turn conversations, large language models may undertake various personas to fabricate dialogues in a conversational schema.

Adapters-based Tuning: This approach introduces supplementary trainable neural modules or parameters, absent in the foundational architecture of original LLMs or their pre-training phase. As a pioneering endeavor in delta tuning, adapter-based techniques interpose diminutive neural units into the Transformer layers. Subsequently, only these newly added adapters are fine-tuned to adapt the model. This straightforward instantiation mechanism [22] not only significantly curtails the necessity to modify the model's existing parameters but also extensively leverages the inductive bias imparted by the adapter layer. Compacter [23] advocates the utilization of a synthesis of hypercomplex multiplication and parameter sharing. From the perspective of tuning computational efficiency, the computational burden may be dynamically diminished by excising adapters from the inferior Transformer strata [24]. Studies further indicate that adapter-based fine-tuning surpasses traditional methods in robustness, particularly excelling over vanilla fine-tuning in few-shot and cross-lingual contexts [25]. In summary, adapters serve as minimalistic, additional neural units that are trainable in a task-specific manner, effectively encapsulating task-relevant information.

Prompt-based Tuning: Rather than embedding neural modules within the Transformer architecture, prompt-based techniques encapsulate the original input with augmented contextual information. Serving as a tactic to activate pre-training language models by emulating pre-trained objectives for downstream tasks, prompt-based instruction has garnered commendable outcomes across a spectrum of NLP tasks [26–28]. A pivotal foundational study in this research domain is prefix-tuning [29], which appends trainable continuous tokens (prefixes) to the input and intermediary states across the Transformer layers. Comprehensive discussions on the methodology and applications of prompt-based instruction are well-documented in existing literature [30]. Prompt tuning integrates novel prompts not as integral elements of the pre-trained model's parameters, but as a supplementary parameter matrix. Throughout the training phase, these soft prompt parameters are refined via gradient descent. Studies [31] have illustrated that as the model's size expands, the efficacy disparity between prompt tuning and comprehensive parameter tuning progressively diminishes. Furthermore, prompt tuning has showcased its capability for cross-task generalization, underscoring its aptitude in effectively adapting expansive pre-trained models. Nonetheless, it is critical to recognize that instruction tuning poses significant challenges in optimization in real-time settings, and soft prompts do not invariably guarantee model convergence across all scenarios.

3. Problem Formulation

In this section, we present a structured collection of models for microservice deployment and request routing, accompanied by an in-depth description of their parameters. Additionally, we have compiled an overview of the key symbols utilized in this document, with their associated physical interpretations displayed in Table 1.

Table 1. Notation and Definition.

Notation	Definition
V, v	the group of devices
C_v	count of CPU cores within an edge node v
μ	the capacity of a single core to handle user requests
D	matrix of network propagation delays
$d(v_1, v_2)$	delay in signal propagation across edge nodes v_1 and v_2
M, m	the collections of sequence microsevice in different types
F, f	the collections of sequence in different user requests
M_f	the request in a collection of microservices contained in f
λ_f	the arrival rate of user request f
$D_f^{\max}$	the maximum tolerable delay for the user request f
N_v^m	the allocation choice of microservice m to edge node v
$P(m \mid m_i, v \mid v_i)$	likelihood that the request is directed to microservice m_i on edge node v_i

Table 1. *Cont.*

Notation	Definition
λ_v^m	the rate of request arrivals for microservice m in edge node
$\overline{W_v^m}$	the response time for processing user requests via microservice m in edge node
L_f, f^i	request the set of routing paths and the routing trajectory
D_f	lag in addressing user queries
η_F	request success rate

3.1. Microservice and User's Request Models

We construct a model for a Mobile Edge Computing (MEC) network that illustrates the network's architecture and the communication pathways among edge nodes, as depicted in Figure 2. Denote $I = (V, D)$ as an MEC network spread across diverse geographic regions. In this model, $V = \left\{v_1, v_2, \cdots, v_{|V|}\right\}$ represents the collection of edge nodes. Each node $v \in V$ includes a base station that manages data transmission and interfaces with adjacent edge nodes, along with an edge server. It is posited that each edge server is equipped to furnish the necessary service resources for hosting microservice instances and processing requests from users. The computational capacity of each node is expressed by the number of CPU cores, C_v, which sets the upper limit of its service capabilities. Allocating microservice instances on individual cores guarantees that the operations of one instance do not interfere with others [5]. Hence, it is established in this study that each CPU core is allocated to a single microservice instance, and users' requests might traverse various edge nodes due to changes in geographical location. These nodes cater to distinct microservice deployment needs, serving areas like commercial districts, residential zones, and intelligent manufacturing facilities. This study specifies the operational range of edge nodes as 200 to 400 m [32], adhering to the policy that requests target the nearest edge nodes.

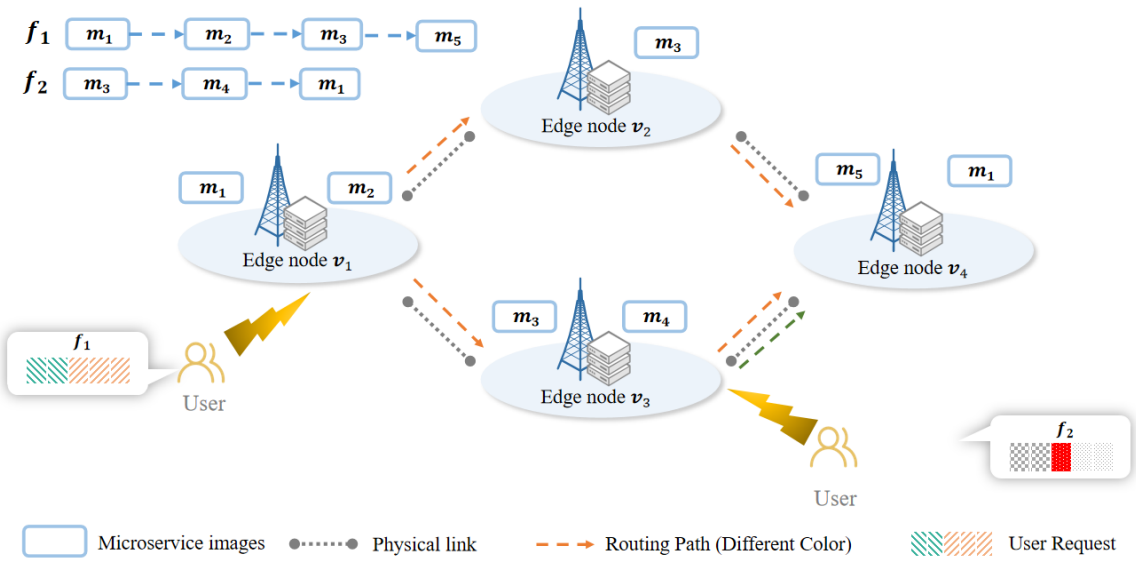

Figure 2. An example of the mobile edge computing network. Microservice deployment and request routing policies are constrained by the computing resources of the edge node servers.Dotted arrows of different colors represent different types of routing requests.

The latency matrix for network communication among edge nodes is designated as D. This matrix is mainly influenced by the geographical positioning of the edge nodes and specifies the transmission delay between any two nodes. For instance, if $d(v_1, v_2)$ equals 3, this indicates a transmission delay of 3 milliseconds (ms) from node v_1 to node

v_2. Furthermore, the delay associated with request transmission within the same node is considered trivial, recorded as $d(v_1, v_1)$ equals 0. Additionally, we characterize the processing throughput of a single core for handling user requests, denoted by μ.

In light of the operational context, microservices constitute granular service fragments tailored to individual user needs. Employing the designation $M = \left\{m_1, m_2, \cdots, m_{|M|}\right\}$, we delineate the assortment of delay-prone microservice types, while the aggregate of user's inquiries within MEC networks is expressed as $F = \left\{f_1, f_2, \cdots, f_{|F|}\right\}$. These user queries are conceptualized as interconnected structures; for example, an online payment request sequence is processed through a quartet of microservices (user identification, balance inquiry, payment execution, and balance modification). Each user query triggers a sequential cascade of microservices $M_f = \left\{m_1, m_2, \cdots, m_{|M_f|}\right\}$ in a predetermined logical progression. The parameter λ_f denotes the influx rate of user requests, which is invariably non-negative and adheres to a Poisson distribution, whereas the notation $D_f^{\max}$ specifies the maximal permissible delay for a user's request.

The strategical allocation of microservices is captured by the integer variable $N_v^m \in \{0, \cdots, C_v\}$, signifying the quantity of microservices m either deployed (greater than or equal to 1) or not deployed (equal to 0). Given the finite resources available at edge nodes, we face the ensuing computational capacity constraints:

$$\sum_{m \in M} N_v^m \leqslant C_v, \forall v \in V \tag{1}$$

The routing variable is represented as $P(m|m_i, v|v_i) \in [0, 1]$, signifying the likelihood of directing a user's traffic to microservice m at edge node v subsequent to the completion of microservice m_i at node v_i. It is presupposed that edge nodes can exclusively forward user requests to those nodes where the respective microservice is operative. Concurrently, it is imperative that the aggregate requests for each microservice in every user's query are comprehensively accommodated. Furthermore, we incorporate Burke's theorem and the concept of Poisson flows to articulate the dynamics of user's request traffic, enabling the amalgamation of these flows within the queuing network. Thus, we posit [5,33]:

$$\sum_{v \in V} P(m|m_i, v|v_i) = 1, \forall m_i, \forall m, \forall v_i \tag{2}$$

$$P(m|m_i, v|v_i) \leqslant \frac{N_v^m}{C_v}, \forall v, \forall m, \forall f \tag{3}$$

Given that instances of microservices may concurrently serve multiple user requests at the identical edge node, we introduce the variable λ_v^m to signify the cumulative rate of user requests arriving. Moreover, we stipulate that the volume of user requests directed should not surpass the operational capacity of any microservice stationed on edge node v. The formulation is expressed as follows [5]:

$$\lambda_v^m = \sum_{f \in F, m_i \in M \setminus m} \sum_{v_i \in V} \frac{P(m|m_i, v|v_i)}{\sum_{f \in F} P(m|m_i, v|v_i)} \lambda_{v_i}^{m_i}, \forall m \in M, \forall v \in V \tag{4}$$

$$\lambda_v^m \leqslant N_v^m \mu, \forall m \in M, \forall v \in V \tag{5}$$

3.2. Optimization Objectives

As illustrated in Figure 2, let's consider two user's requests: f_1 and f_2, marked as $F = \{f_1, f_2\}$, each user request follows a processing order:

$$f_1 : m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_5 \tag{6}$$

$$f_2 : m_3 \rightarrow m_4 \rightarrow m_1 \tag{7}$$

To orchestrate the aforementioned user requests, we postulate that the MEC environment encompasses four edge nodes $V = \{v_1, v_2, v_3, v_4\}$, with a quintet of required microservices enumerated as $M = \{m_1, m_2, m_3, m_4, m_5\}$. Figure 2 delineates the architecture of the multi-edge network, complete with specified latency interconnections between nodes, encapsulated within the propagation delay matrix D. Microservices are systematically instituted across each node, ensuring that a minimum of one instance per microservice is operational within the network. Additionally, Figure 2 elucidates the navigational trajectory for user request f_1, delineating its allocation to specific microservice instances. For example, as the request sequence f_1 migrates from microservice m_2 at node v_1 to m_3, the possible pathways include routes $v_1 \rightarrow v_2$ and $v_1 \rightarrow v_3$.

Within this study, our paramount concern is the minimization of delay, pivotal in determining the Quality of Service (QoS) during the processing of user requests. This delay comprises two principal segments: processing and propagation delays. The temporal expenditure incurred by a user request whilst traversing a microservice is captured by the auxiliary variable $\overline{W_v^m}$. Grounded in the principles of First Come First Serve (FCFS) queuing theory and Processor Sharing (PS), we establish [33]:

$$\overline{W_v^m} = \frac{1}{N_v^m \mu - \lambda_v^m}, \forall m \in M, \forall v \in V \tag{8}$$

In the analysis of the response latency for user inquiries, consideration must be extended beyond the dwell time of such requests. Inter-node user request routing introduces the necessity to factor in the network propagation latency between edge nodes. We use $L(f) = \left\{f^1, f^2, \cdots, f^{|L(f)|}\right\}$ to denote the set of user's request routing paths that follows the order of the microservice contained in M_f. The path $f^i = \left\{v_{m_1}^{f^i}, v_{m_2}^{f^i}, \cdots, v^{f^i}_{m_{|M_f|}}\right\}$ means the ith request route contains the edge nodes passed under this path. Hence the propagation writes [33]:

$$\sum_{j=1}^{|f^i|-1} d\left(v_j^{f^i}, v_{j+1}^{f^i}\right) \tag{9}$$

We define D_f as the user's request response delay:

$$D_f = \sum_{k=1}^{|L(f)|} \left(\left(\prod_{i=1}^{|M_F|-1} p(m_{i+1}|m_i, v_{i+1}|v_i) \right) \left(\sum_{n=1}^{|f^i|} \overline{W_{f_n^i}^{m_n}} + \sum_{j=1}^{|f^i|-1} d\left(v_j^{f^i}, v_{j+1}^{f^i}\right) \right) \right) \tag{10}$$

In summary, the minimization of response delays is achieved by strategically selecting the quantity and positions of microservice instances alongside optimizing the pathways for request routing. Consequently, this entire issue is reformulated as a Response Delay Minimization Problem (RDMP), classified as a MILP problem [34] and recognized as NP-hard. Addressing this problem allows us to determine the response delays for each user request. Uniform benchmarks must appraise the outcomes. Various user request types exhibit distinct thresholds for maximum allowable delays. Thus, our analysis is centered on the correlation between the observed delays for user requests D_f and their respective maximal permissible delays $D_f^{\max}$. The ensuing constraint formulations are presented below:

$$\text{s.t.} \begin{cases} \min \quad D_f \\ (1) - (5) \\ N_v^m \in \{0, \cdots, C_v\} \\ P(m|m_i, v|v_i) \in [0, 1] \end{cases} \tag{11}$$

3.3. Prompting Strategies

Consider a Large Language Model (LLM) during inference $\mathcal{LM}_\theta : \mathcal{X} \to \mathcal{Y}$, characterized by parameters θ. Additionally, for each choice o_j^i, there corresponds a reasoning distribution $\mathbb{R}^i(\bullet \mid q_i; \boldsymbol{\theta})$ aligned with the set of queries q_i. Notably, it is posited that each selection proffered by the LLM represents an optimal resolution. Furthermore, let us denote the reasoning linked to a chosen alternative o_j^i as $r_j \sim \mathbb{R}^i\left(\boldsymbol{o}_j^i \mid q_i; \boldsymbol{\theta}\right)$. Let the prompt $\mathcal{P} = \left\langle q_i, \mathcal{O}_i, \mathcal{R}^i \right\rangle$ be an input to make LLM generate corresponding response $\mathcal{J}^i$, in which $o_j^i \in \mathcal{O}_i$ and $\mathcal{J}^i \in \mathcal{Y}$. Therefore, we can see that in order to make the inference process faster and the result more in line with the distribution of the optimal solution, it is lower cost and more reasonable to use a better prompt strategy on the premise of not fine-tuning the number of parameters.

3.3.1. Prompt Components

As a programming language in the era of general artificial intelligence, Prompt [31] is not just a simple input or query, it is the key to our interaction with large language models (LLM), from simple questions and answers, text generation to complex logical reasoning mathematical operations, and even writing drawings that are considered to be the display of human creativity, by rational and subtle use of prompt, We can interact with the AI in real time without waiting for a lengthy compilation process, and a complete prompt should contain clear prompts, relevant context, a few examples to help the model understand, clear inputs, and a description of the desired output and output format. To effectively use LLM for addressing questions related to microservices deployment,specifically focusing on request delay optimization, we constructed our prompts with the following four components:

Role Prompt: As illustrated by the MetaGPT research [35], the role an agent plays significantly enhances the performance of Artificial General Intelligence(AGI) models. In our implementation, we defined the agent's role as 'An experienced algorithm developer in the Microservices domain'. This designation ensures that the model tailors its responses to this specific role, drawing upon relevant prior knowledge and domain logic, thereby enriching its insights into microservice deployment challenges. We opted for a more general role rather than specialized titles like 'engineer' or 'programmer' to fully leverage the model's capabilities. Additionally, this approach signals to the LLM that our objective involves dissecting an abstract mathematical problem into manageable sub-problems. Accordingly, we instructed the model to conserve tokens and provide immediate feedback—responding only with 'got it'—before proceeding to generate the actual answer.

Progressive Context Prompt: We equip the Large Language Model (LLM) with a detailed context to assess its ability to accurately answer questions based on the provided information. This context may encompass specific prior knowledge, an abstractly described mathematical model, a designated theoretical method for solving the problem, and analogous code examples presented in natural language. Acknowledging that the LLM might not generate viable solutions immediately, we introduce hints progressively, from simple to complex, thereby aiding the model in gradual knowledge accumulation. The concept of symmetry plays a role here, as we balance the complexity of input information with the model's output capabilities, ensuring a structured and aligned progression of learning. Throughout this process, we may choose to include or omit certain pieces of information based on the feedback from the LLM. This symmetry between input and output is crucial, as it helps the model generate coherent and optimized results while minimizing errors. For instance, at each stage, we incorporate prompts that encourage the model to self-reflect. Should the LLM generate inaccuracies or erroneous outputs, we judiciously reduce the amount of information provided. This strategy enables the LLM to progressively acquire knowledge and insights.

Task Prompt: Given the role prompt and context prompt, we employ a more targeted approach to prompt engineering. For problem segments requiring an understanding of

specific definitions, we take a sequential method. Initially, we prompt the LLMs to define the concept, and if the response is accurate, we restate the original problem to the LLMs in a more comprehensible manner. The final answer is then formulated as the task prompt. Our approach emphasizes expressing abstract mathematical problems in natural language rather than technical jargon. We believe that this method enables LLMs to generate clearer and more precise answers.

3.3.2. Chain of Thought

The functionality of large-scale language models, grounded on Transformer architecture, is often perceived as a proficiency in grasping empirical phenomena and orchestrating logical connections among disparate concepts within the realm of natural language. The Chain of Thought (CoT) [20] methodology has been deployed to surmount the challenges inherent in enabling large language models to autonomously resolve intricate mathematical quandaries. By necessitating the model to articulate sequential inferential steps prior to delivering the ultimate resolution, CoT substantially augments the model's inferential output. Unlike conventional prompt-based direct mappings from <input-output>, CoT facilitates a <input-reasoning-output> transformation. This capability allows large language models to fragment complex, multi-tiered problems into discernible intermediate sequences, thereby permitting additional computational resources to be directed towards segments necessitating extensive deductive processes. Additionally, CoT offers a lens through which the model's operational logic can be interpreted, illuminating potential pathways by which specific conclusions were derived and presenting avenues to diagnose erroneous reasoning trajectories. In scenarios where a mathematical issue is thoroughly defined yet erroneously resolved by the LLM, CoT enables precise identification of the juncture at which the model's reasoning deviated.

The easiest way to provide CoT is by using prompt words like "let us think it step by step " or "After Tom gives 3 dollars to his dad he has 7 dollars.... So we can infer Tom has 3 + 7 = 10 dollars at the begining", but sometimes it's not that easy, espcially in the face of zero-shot or few-shot problems. Our proposed approach is to augment each example in few-shot prompting with a chain of thought for an associated answer,as illustrated in Figure 1. We manually composed a set of several few-shot exemplars with chains of thought for prompting (not undergo precise prompt engineering). Given that we are dealing with np-hard problems, the exemplars are not all optimal, but try to teach the model a way to approximate the optimal solution.

3.4. Deep Reinforce Learning

Reinforcement Learning (RL), alternatively termed as appraisal learning or evaluative learning, embodies a machine learning paradigm and methodology. It delineates a framework employed to elucidate and tackle the objective of optimizing returns or attaining specified ambitions through strategic interaction between an agent and its environment. In essence, Reinforcement Learning adopts a learning mechanism that deciphers mappings from states to actions to optimize accrued rewards. This approach encompasses four principal components: Strategy, Reward, Value, and Environment or Model, with iterative learning and deferred rewards as its pivotal features. The entire methodology is typically encapsulated by the Markov Decision Process (MDP), characterized as a discrete stochastic process possessing Markov properties.

Deep Reinforcement Learning (DRL) amalgamates the perceptual strength of deep learning with the strategic acumen of reinforcement learning, creating an integrated framework. This amalgamation enables direct manipulation from unprocessed inputs to outputs through a process of end-to-end learning. The DRL methodology unfolds across three sequential phases: (1) At each timestep, the agent engages with the environment to glean a high-dimensional observation, subsequently processed via deep learning techniques to delineate discrete or nebulous state attributes. (2) The agent appraises the value function for potential actions, predicated on expected returns, and elects an action rooted in a

pre-established policy. (3) The environment reacts to the chosen action, furnishing a new observation and sustaining a cycle that continuously hones the policy towards peak efficacy. Prominent models of DRL include the Deep Q-Network (DQN), Deep Deterministic Policy Gradient (DDPG), Trust Region Policy Optimization (TRPO), Proximal Policy Optimization (PPO), and Asynchronous Advantage Actor-Critic (A3C), among others [36].

The Deep Deterministic Policy Gradient (DDPG) is a seminal reinforcement learning algorithm tailored for continuous control domains. This off-policy method synergizes Policy Gradient (PG) techniques with the framework of Deep Q-Network (DQN), representing an advancement over DQN. DDPG enhances the traditional Q-network by incorporating an Actor network, which predicts an action value aimed at maximizing the Q value, thereby adeptly handling continuous action spaces. It also integrates the target network and experience replay mechanisms found in DQN. Addressing the model discussed in Section 3.1 entails devising optimal microservice deployment and routing strategies. However, formulating the most effective strategy within complex marginal environments poses considerable challenges. This requires a holistic assimilation of data, including geographic locales, bandwidth capacities, resource availability at edge nodes, terminal device geographies, and network bandwidth. Moreover, a profound understanding of the application request architecture and the resource demands for executing microservices is crucial. Theoretically, DDPG's aptitude for managing continuous deployment actions capitalizes on the comprehensive data available in edge environments, making it ideally suited for the optimization objectives outlined.

Consequently, in the process of driving large models to generate code for deployment strategies applicable to our proposed microservice model, the significance of DDPG is underscored. We employ a gradual instruction policy, demanding that the large model generate strategies based on the DDPG framework within the realm of deep reinforcement learning. This approach ensures that the algorithm's potential is fully leveraged to meet the specific requirements of our microservice architecture, optimizing both deployment and operational efficiency. The flowchart of the DDPG algorithm is depicted in Figure 3.

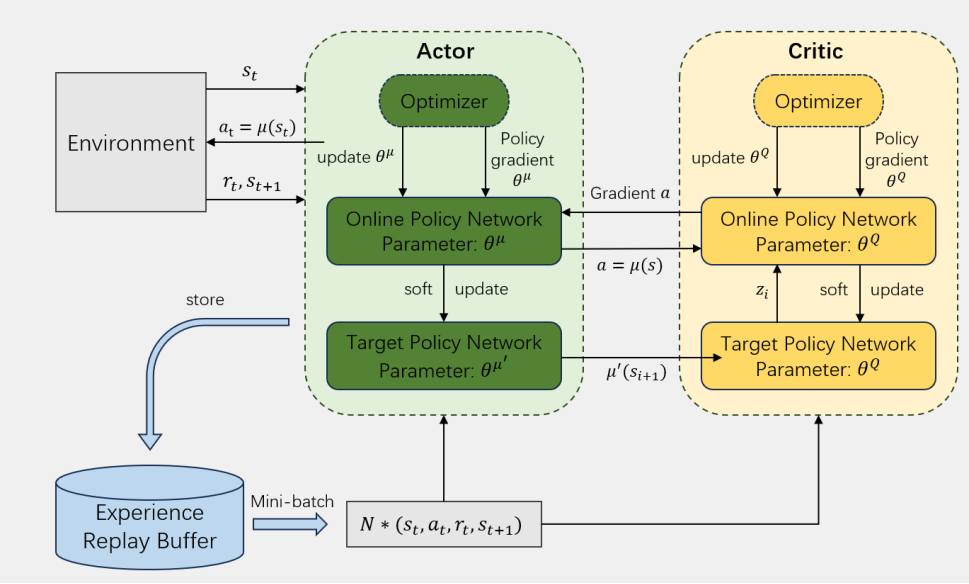

Figure 3. This is the flowchart of the DDPG algorithm, which uses the Actor-Critic framework as its base. The algorithm employs dual neural networks for both the policy and value functions, comprising an Online network and a Target network. The Critic Target network approximates the Q-value function for the next state-action pair and evaluates the current policy, ensuring slow parameter updates via a soft update mechanism. The Actor Target network provides the policy for the next state. Experience samples (current state s_t, action a_t, reward r_t, next state s_{t+1}) generated by Actor-environment interactions are stored in a replay buffer. Batch sampling from this buffer removes sample correlation and dependency, facilitating easier algorithm convergence.

3.5. Retrieval Augmented Generation

Retrieval-Augmented Generation (RAG) technology enhances large language models (LLMs) by enabling them to retrieve information from dedicated external data sources. This capability allows the model to supplement its responses to queries with data from an external database, thereby ensuring more accurate and relevant outputs. Variations in the training corpora and datasets of pre-trained LLMs launched by different entities can lead to inaccuracies when these models are posed domain-specific questions beyond their initial training data. To address this, we construct a specialized private dataset incorporating theoretical knowledge from fields such as microservice deployment, queuing theory, and reinforcement learning. The pre-trained LLM then retrieves information from both external open sources and our tailored knowledge base. This retrieved information serves as context for the LLM to generate appropriate responses, significantly enhancing the factual accuracy and relevance of its outputs.

Numerous variants of RAG exist. Herein, We delineate a prototypical RAG workflow stage. This typically bifurcates into two principal processes: an Index process, executed singularly at the commencement of the application, and a Query process, recurrently triggered by incoming inquiries. The Index process unfolds as delineated: the input document is segmented into distinct fragments $K = \left\{k_1, k_2, \cdots, k_{|n|}\right\}$. Employing an encoder model, these fragments are transformed into embedding vectors $\vec{d}_i = \text{encoder}(k_i)$, subsequently archived within a vector database. This repository facilitates the retrieval of pertinent segments for specified queries. In the ensuing Query process, responding to user queries Q, the encoder model crafts a vector representation of the query $\vec{v} = \text{encoder}(Q)$. The database is scrutinized to extract the uppermost t fragment embeddings $\left\{\vec{d}_1, \vec{d}_2, \ldots, \vec{d}_t\right\}$ akin to the query embeddings $\vec{v} = \text{encoder}(Q)$. Cosine similarity metrics are utilized to ascertain congruence between chunk embeddings and query embeddings, dictating the selection and retrieval of chunks. The foremost t chunks, alongside the query, are integrated into the prompt template. The completed prompt then serves as input to an LLM model, which renders an output predicated on the provided data. This output is subsequently delivered to the user. The pseudocode executed through this process is expounded in Algorithm 1, whilst the workflow for the comprehensive fine-tuning of the large language model is illustrated in Figure 4; further exposition will follow in Section 4.

Algorithm 1 Process of a typical RAG system

Index Process:
Load embeddings: $embeddings \leftarrow$ load("fine-tuning_model")
Load document: $doc \leftarrow$ load("file_name")
Chunk document: $c \leftarrow$ chunk.file(doc)
Embed chunks: $Ce \leftarrow embeddings.chunk(c)$
Create database index: $db \leftarrow$ index(Ce)
Query Process:
Initialize system prompt: $sys\_prompt \leftarrow$ "you are an AI..."
Load model: $model \leftarrow$ load("fine-tuning_model")
repeat
    Fetch user query: $q \leftarrow$ get.user_query()
    Embed query: $Qe \leftarrow embeddings.chunk(q)$
    Search database: $chunks \leftarrow db.search(Qe)$
    Merge search results: $context \leftarrow$ merge(chunks)
    Create complete prompt: $prompt \leftarrow$ create_prompt(sys_prompt, q, context)
    Generate answer: $answer \leftarrow model.generate(prompt)$
until false ▷ Infinite loop

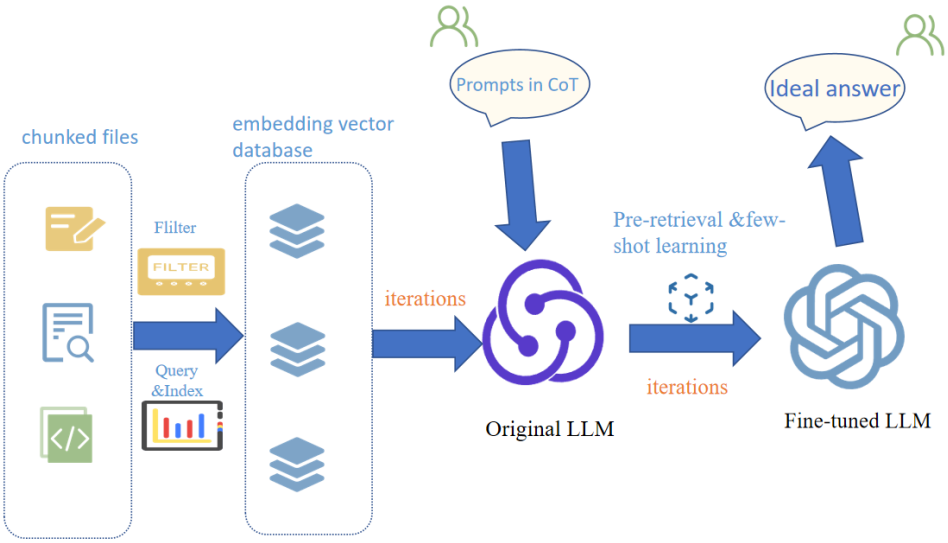

Figure 4. Our Retrieval Augement Generation flowcharts.

4. Experimental Design

In our experiments, we utilized the ChatGPT gpt-3.5-turbo-16k model with a maximum context length of 16,000 tokens. The experimental environment consisted of a server with 2.0 TiB RAM, 2 Intel Core i7-13700K 16-core processors(Intel Corporation, Santa Clara, CA, USA), and 4 NVIDIA RTX 3090 GPUs (24 GiB VRAM each)NVIDIA Corporation, Santa Clara, CA, USA). To fine-tune the selected large language model, we built a Retrieval-Augmented Generation (RAG) dataset by processing relevant documents using the LangChain library. We extracted data from sources such as books, journals, and research reports with search terms including "Microservice Deployment", "Deep Deterministic Policy Gradient", and "Queuing Theory". This process resulted in a dataset comprising 3957 question-answer pairs, which was used to enhance the model's performance in solving specific tasks. Text was segmented using a chunk size of 100 with an overlap of 50, and Maximum Marginal Relevance (MMR) [37] was applied for document retrieval.

For the simulation parameters and environment necessary to validate the code generated by the fine-tuned model, we utilized the cluster-trace-microservices-v2021 and cluster-trace-microarchitecture-v2022 datasets, publicly available from Alibaba [5,38]. These datasets provided real-world data, enabling us to simulate realistic scenarios. To ensure the reliability and stability of the results, all outcomes presented in this paper are the average of 500 independent experiments. Our experimental parameter settings are based on real-world data traces from MEC (Mobile Edge Computing) network scenarios. Code generation followed a Chain-of-Thought (CoT) [39,40] approach, with DDPG-based optimization to solve the Response Delay Minimization Problem (RDMP) [41–44]. All generated code was based on Python 3.10 and PyTorch 1.12.0.

4.1. Experimental Setup

For our experimental execution,We used ChatGPT gpt-3.5-turbo-16k (https://openai.com/index/gpt-3-5-turbo-fine-tuning-and-api-updates/) (accessed on 29 October 2024) as the source model throughout all of our main experiments, which is a variant of OpenAI's Generative Pre-trained Transformer (GPT) model [9]. This particular version has been optimized for efficiency and performance,from casual conversations to more specific tasks requiring nuanced understanding,making it suitable for various applications. Compared to other LLMs, the most standout feature of the "gpt-3.5-turbo-16k" which attracts us is its ability to consider up to 16,000 context tokens in a single interaction. (normal LLaMA2 (https://llama.meta.com/llama2/ (accessed on 29 October 2024)) only provides 4096 tokens in a single interaction). This allows the model to handle longer conversations or

detailed content more effectively, capturing intricate details and providing more coherent responses over extended interactions.

The general experimental procedure involves the following steps. First, we express the microservice deployment model in natural language. In essence, abstract mathematical problems are described in segmented natural language, while providing clear contextual connections. Secondly, we build an instruction dataset to create our enhanced search database, also known as RAG. We then pass progressive and context-aware prompts to our GPT model, guiding it to generate further answers by searching both private knowledge databases and open-source databases. During this process, we meticulously examine the model's generative answers step by step to ensure it has acquired the necessary learning, reasoning, reflecting, and generative capabilities. Finally, we let the large language model, fine-tuned with CoT and RAG, solve the problem outlined in step one and generate the required code. It is important to note that we did not train this model ourselves; instead, we used the model accessible through OpenAI's API without further modification. Re-training an LLM requires expensive computational resources and a large dataset, and we believe in the inherent power of our base model.

4.2. Experimental Implementation

4.2.1. Instruction Dataset Preparation

We use 'Microservice Deployment ', 'Deep Deterministic Policy Gradient', and 'Queuing theory' as search terms to search and screen related books, journals, papers, and research reports in open source databases (https://huggingface.co/ (accessed on 29 October 2024)). The first step was to parse the original PDF,Word and Markdown document file into text format for further processing, we use the *LangChain* library (https://python.langchain.com/ (accessed on 29 October 2024)) to process files to the same format by using filtering, compression and formatting. In addition to text, the file also contained tables and images illustrating all entities. The tables and images were manually converted to text by a human expert who wrote sentences describing the information. Mathematical formulas and derivations are treated in the same manner.

The ensuing phase entailed segmenting the document into discrete portions. As explicated in Section 3.5, this segmentation was orchestrated according to the section headings, with each distinct heading initiating a new chunk. Subsequently, we created <question, answer> pairs for each segment over several iterations. Initially, the gpt-3.5-turbo-16k model was employed to fabricate questions and answers for the designated segments. The outcomes, encompassing questions, answers, and pertinent segments, were cataloged. In a subsequent iteration, We engaged the model with a sample dialog and solicited it to forge a discourse between a user and an AI assistant for each segment. During the third iteration, the model was again charged with the task of generating questions and answers, this time utilizing examples of questions devised by a human expert from the document. The collection of questions and answers generated across these iterations was amalgamated to assemble our dataset. Quality assurance was executed through meticulous scrutiny of the produced questions and answers, including the excision of any redundant questions. The ultimate compilation of our dataset comprises 3957 question and answer pairs.

4.2.2. The Work Flow and Implementation Configurations

The implementation of RAG is to generate answers using blocks from the original document using the gpt-3.5-turbo-16k model. Since the open source model used for experimentation in this paper has a pre training data cut off time on July 2021. We store the vector database based on Weaviate-Client, and we are responsible for the arrangement of the whole process in Langchain without additional pre-training. We use the TextLoader provided by Langchain to load the collected data sets. In order to ensure the continuity between texts, we use CharacterTextSplitter, chunk size is set to 100, chunk overlap is set to 50, we used Maximum Marginal Relevance (MMR) [37] for document selection during

retrieval; this algorithm selects documents based on a combination of similarity to the input query while also optimizing for diversity of retrieved documents. For the embedding model we used "Instructor", a text embedding model that can produce embeddings for a variety of domains. During inference, We use greedy decoding (temperature is 0) with a repetition penalty of 1.05 to generate responses.

Next, we utilize the enhanced large language model generated after retrieval to address our Response Delay Minimization Problem (RDMP). We begin by decomposing the problem into several sub-problems and then input progressive Chain-of-Thought (CoT) instructions into the model to produce corresponding inference steps and code snippets for each sub-problem. We apply a DDPG-based approach within the RAG framework to derive the final algorithm. The model outputs code in python format, which minimizes uncertainty and opaque knowledge inherent in "prompt engineering" processes, and simplifies the presentation of complex tasks into class-based code. We explicitly instruct the model to break down the generated code into small subfunctions, use meaningful variable names, and ensure clear functionality within each function body. This approach addresses our observation that large language models often perform poorly when tasked with generating lengthy single functions. By reducing the complexity of problems [45], the readability of the generated code improves, and the model is more successful at autonomously detecting and fixing bugs during iteration. Throughout this process, we instruct the model to reproduce the same output for identical instructions, then have it correct its own output and send the final result to the user. We believe this dual-verification approach effectively mitigates errors and hallucinations.

All experiments were conducted on a server with 2.0 TiB RAM, 2 Intel Core i7-13700K 16-Core Processors, and 4 NVIDIA RTX3090 Gpus, each having 24 GiB of VRAM. and all the generated deep reinforcement learning code was based on the python 3.10, pytorch 1.12.0 version of the framework.

5. Results

5.1. Code Consistency Check

Evaluating the outputs of LLMs [46] is commonly done using either automated or human evaluation,Although automated evaluation is easier to scale and more accessible, this approach relies on a larger and more powerful language model, such as GPT-4 o. However, feedback from real users may be more cost-effective and more readily accepted by human systems. Therefore, we employ manual evaluation to assess the microservice deployment strategy code generated by our system. We created two algorithmic frameworks using a model fine-tuned solely for prompt fine-tuning and another model enhanced with both prompt fine-tuning and RAG. To facilitate comparison with existing algorithms in the field, we selected three established approaches, including heuristic and reinforcement learning algorithms: the Greedy Nearest Service Deployment Algorithm (First Fit Decreasing, FFD) [33], the Arrival Rate Descending Priority Algorithm (GMDA) [47], and the Deep Q-Network based Reinforcement Learning Algorithm (RSDQL) [5].

20 doctoral students with extensive coding experience in the field and 30 developers from other domains were invited to evaluate the code generated by the large language model and the reproduced algorithms. The evaluation focused on the code's executability, readability, and both time and space complexity. To streamline the process, we employed three assessment criteria: Agree, Neutral Scale, and Disagree.We requested the invited participants to conduct a code consistency check from five aspects: code correctness, logical soundness, completeness, time complexity, and cyclomatic complexity. Each of these five modules was scored out of 20 points. The scores were then normalized based on the weighting they provided, resulting in the heatmap shown below. We categorized a total score below 60 as disagreement, between 60 and 80 as neutral, and above 80 as agreement. Given that the participants were unaware whether the code they evaluated was generated by a large model or written by a human, we believe the heatmap effectively reflects the

readability and correctness of the evaluated code. The evaluation results are illustrated in the Figure 5 below.

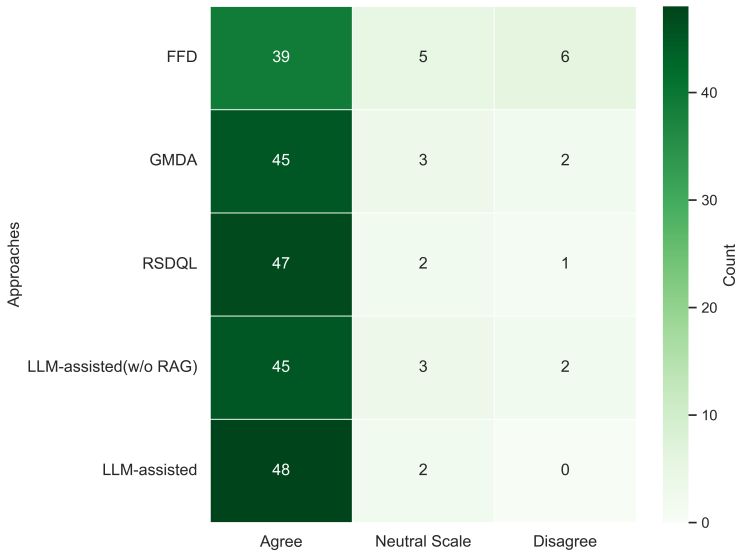

Figure 5. Code evaluation acceptance score table.

5.2. Baseline Algorithms and Parameters

We used the GPT model, enhanced with instruction tuning and retrieval-augmented generation (RAG) [48], to generate deep reinforcement learning code for the minimum delay optimization problem. This process is not straightforward, as the large model can produce incorrect code or hallucinations. Through multiple iterations of optimization and generation, we refined and evaluated the code snippets for consistency, ensuring they could run on our platform. We refer to this as the LLM-assisted algorithm. For comparison, we also generated an algorithm using the initial GPT model with only instruction tuning and no retrieval enhancement, keeping all other instructions the same. We call this the LLM-assisted (w/o RAG).

We then compared the generated code with three existing algorithms. The First Fit Decreasing (FFD) algorithm is a greedy service deployment algorithm that prioritizes nearest deployment. If edge node resources are insufficient, it deploys to the next nearest node, and so on, until all microservices are deployed. The Greedy Microservice Deployment Algorithm (GMDA) [49] also uses a greedy approach, prioritizing deployment based on descending request arrival rates. RSDQL is a reinforcement learning algorithm based on deep Q-networks, sharing rewards at each step to ensure learning towards the optimal policy, thereby achieving optimal deployment cost.

It should be noted that the models used in the original studies of these three comparison algorithms differ from the model used in this study. We replicated their core concepts to ensure accuracy. For edge nodes and terminal devices, we simulated a realistic edge environment by setting the number of edge nodes between 4 and 10 and the number of microservice types between 3 and 20. Considering the heterogeneous resource requirements for executing microservices on edge nodes, we set the bandwidth between edge nodes to 1–5 Mbps, CPU resources to 15 ($\times$1000)–30 ($\times$1000) millicores, and memory resources to 200–300 MB. Each microservice instance's CPU resource consumption ranges from 1 ($\times$1000) to 2 ($\times$1000) millicores, and memory consumption ranges from 10 to 15 MB. Additionally, the dependency data between different microservices varies, ranging from 0 to 3 MB. The hyperparameter settings for the baseline algorithms are shown in the Table 2 below:

Table 2. Hyperparameter Settings in Baseline Algorithms.

Parameter	Value	Description
lr_actor	0.005	Learning rate of the Actor network
lr_critic	0.005	Learning rate of the Critic network
lr_dql	0.005	Learning rate of RSDQL
μ	0.01	Soft update factor
γ	0.95	Reward discount factor
η	0.8	Resource weight factor
VAR	2	Exploration factor
memory buffer	10,000	Experience pool capacity
batch size	32	Data sampling size

5.3. Performance Evalution

We changed three parameters that may affect delay performance in the moving edge scenario: the number of users (corresponding to the number of requests), the number of edge nodes and the number of microservice types, and compared them with the three baseline algorithms in the same setting environment. Our optimization goal is to minimize delay.

5.3.1. Delay Performance with Increasing Request Flows

As shown in Figure 6, increasing the number of request flows leads to a rise in total request access delay. This trend continues until the system reaches resource saturation, at which point the average request access delay stabilizes within a certain range. The proposed LLM-assisted algorithm consistently outperformed the baseline algorithms, showcasing superior delay performance. Specifically, the LLM-assisted algorithm improved delay performance by approximately 5.6% compared to LLM-assisted (w/o RAG), 7.2% compared to RSDQL, 22.8% compared to FFD, and 16.3% compared to GMDA. The results clearly demonstrate that as the number of request flows increases, total access delay also rises due to growing resource demands. However, once the system reaches resource saturation, the delay stabilizes, reflecting the system's inability to further optimize performance without additional resources.

The LLM-assisted algorithm's performance stands out due to the integration of RAG, which allows the model to retrieve relevant knowledge dynamically. The 5.6% improvement over the version without RAG underscores how this additional knowledge enhances decision-making, enabling the system to handle complex requests more efficiently. The 7.2% improvement over RSDQL shows that the combination of RAG and the CoT reasoning allows the LLM to outperform traditional reinforcement learning models, which are typically less flexible in adapting to dynamic request flows.

The algorithm's 22.8% improvement over FFD further highlights the advantages of LLMs in adapting to fluctuating real-world conditions. FFD, while efficient in static environments, lacks the adaptive reasoning power of an LLM. Similarly, the 16.3% advantage over GMDA suggests that the LLM-assisted algorithm, through its ability to retrieve and reason over data, can manage resource allocation more effectively in diverse scenarios.

These results validate the effectiveness of incorporating RAG into the LLM framework, particularly in edge computing environments where real-time optimization is crucial. The model's ability to retrieve relevant information on-demand and process it with CoT instructions leads to more robust and responsive optimization of microservice deployment.

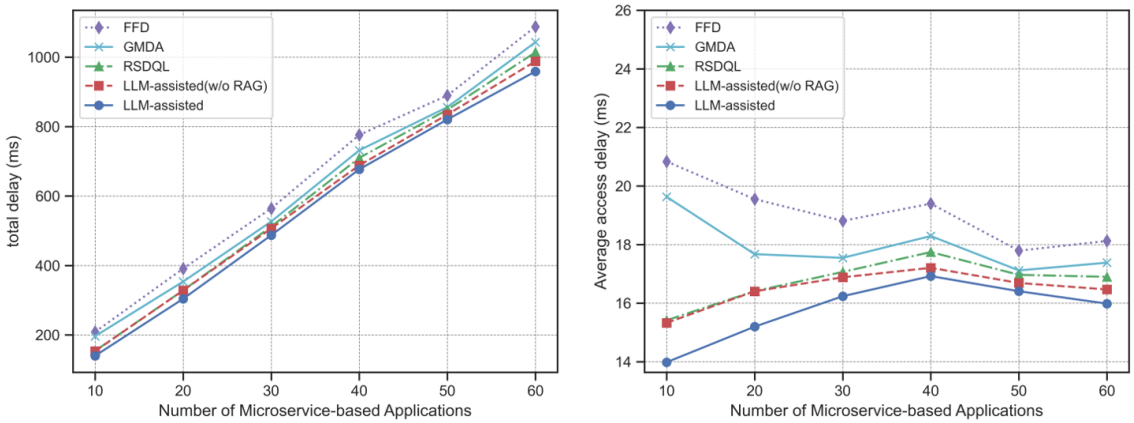

Figure 6. Total delay and Average delay Performance with Increasing Request Flows.

5.3.2. Impact of Microservice Classes

As illustrated in Figure 7, an increase in the number of microservice Classes necessitates the system to deploy more instance resources to ensure all types of microservices are processed correctly. This approach, aligned with queuing theory, results in a slight increase in deployment cost but reduces the queuing delay for the same class of microservice, thereby decreasing the overall request access delay. The LLM-assisted algorithm again demonstrated its superiority, with performance improvements of approximately 9.8% compared to LLM-assisted (w/o RAG), 11.6% compared to RSDQL, 37.8% compared to FFD, and 32.3% compared to GMDA. The results shown in Figure 7 demonstrate that as the number of microservice classes increases, the system requires more resources to ensure all microservices are processed correctly. This increase in resource allocation, which is consistent with queuing theory, leads to a slight rise in deployment costs. However, this additional investment significantly reduces queuing delays, especially within the same class of microservices, ultimately lowering the overall request access delay.

The 9.8% improvement of the LLM-assisted algorithm compared to the version without RAG shows how the retrieval of prior knowledge enables more informed and optimized resource deployment decisions. This is critical in complex environments where various microservice classes must be managed efficiently. The 11.6% improvement over RSDQL further reinforces the advantages of using an LLM in combination with RAG, as it provides a more adaptable and robust solution than traditional reinforcement learning approaches, which may struggle with handling a broad spectrum of microservice classes.

The most significant improvements were observed when comparing the LLM-assisted algorithm to heuristic algorithms like FFD (37.8% improvement) and GMDA (32.3% improvement). These results underscore the limitations of traditional heuristic methods, which are less suited to dynamic environments where microservices vary in complexity and demand. The LLM-assisted algorithm's ability to dynamically allocate resources and optimize queuing delays makes it more efficient in managing diverse microservice types.

Overall, these findings highlight the LLM-assisted algorithm's strength in handling a wide range of microservice classes by optimizing resource allocation and minimizing queuing delays, further demonstrating the model's potential in enhancing microservice deployment performance in edge computing scenarios.

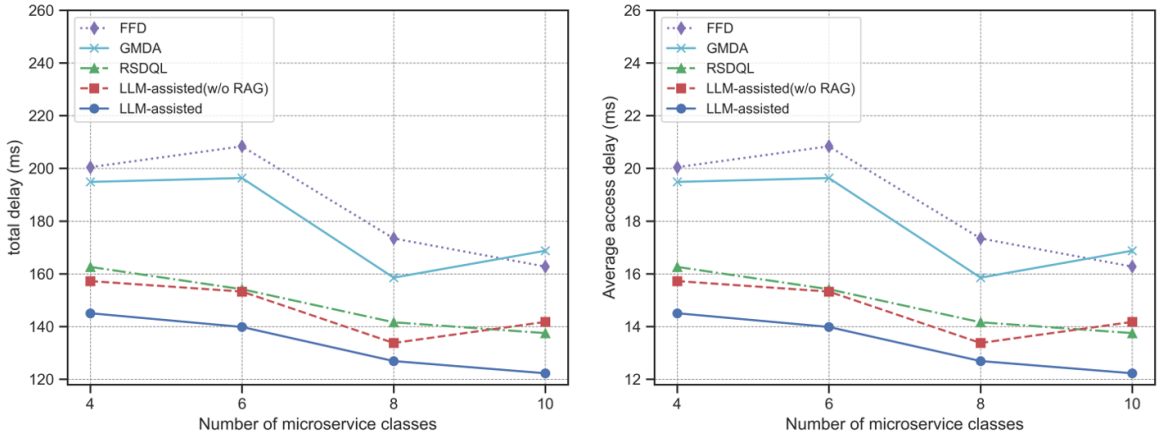

Figure 7. Total delay and Average delay Performance with different Microservice Classes.

5.3.3. Network Complexity and Edge Node Capacity

As depicted in Figure 8, as edge nodes reach full capacity, the network's complexity increases, leading to higher system delays due to the complexity of routing. The experimental results confirm that user request access delay generally increases with the number of edge nodes. However, the LLM-assisted algorithm consistently maintained superior delay performance, with improvements of approximately 8.3% over RSDQL, 12.2% over LLM-assisted (w/o RAG), 39.5% over FFD, and 28.5% over GMDA.

The 8.3% improvement over RSDQL indicates that the LLM-assisted algorithm, supported by Retrieval-Augmented Generation (RAG) and Chain of Thought (CoT) reasoning, handles network complexity more effectively than traditional reinforcement learning methods. This advantage is particularly evident in managing the routing decisions required when edge nodes are fully utilized.

The 12.2% improvement compared to the LLM-assisted algorithm without RAG further emphasizes the importance of incorporating prior knowledge into the decision-making process. With RAG, the model retrieves relevant information that helps optimize routing and resource management even under high network load, contributing to more efficient operations.

The LLM-assisted algorithm's 39.5% improvement over FFD highlights the limitations of heuristic approaches in complex network environments. Heuristic methods like FFD, which are not adaptive, struggle to handle the increasing complexity as edge nodes reach capacity. Similarly, the 28.5% improvement over GMDA indicates that even sophisticated optimization algorithms are less effective than the LLM-assisted approach when dealing with the dynamic nature of edge network routing.

Overall, the LLM-assisted algorithm's ability to maintain efficient routing and manage network complexity, even as edge node capacity is fully utilized, demonstrates its robustness and scalability in edge computing environments. This makes it a highly effective solution for optimizing resource allocation and reducing delays in real-world applications where network complexity is a significant challenge.

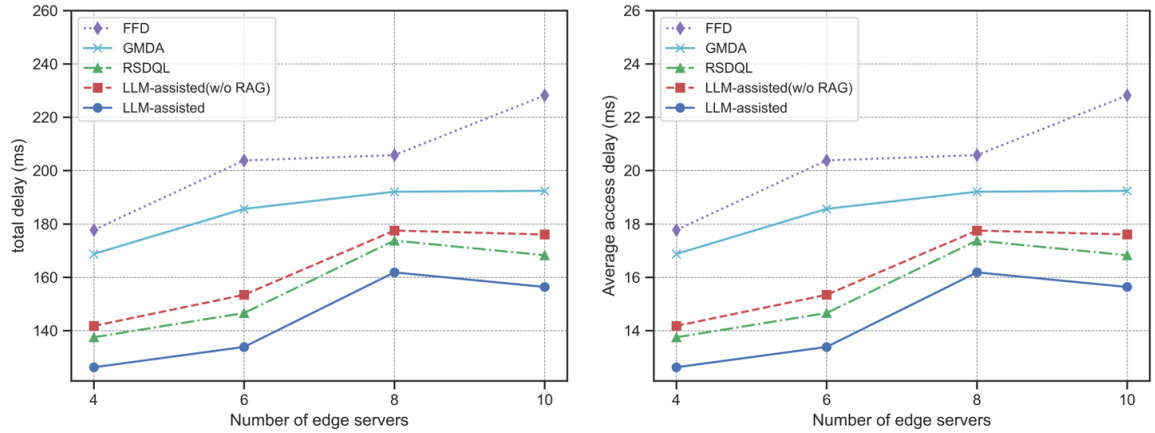

Figure 8. Total delay and Average delay Performance with different number of edge nodes.

6. Conclusions

Our study presents a novel approach to leveraging Large Language Models (LLMs) for optimizing microservice deployment delay in edge computing scenarios. Due to the impracticality of fine-tuning large models directly on edge devices, we integrated the inherent reasoning abilities of LLMs with retrieval-augmented generation (RAG) databases to create a domain-specific training set. This combined approach enabled the generation of optimized solutions by progressively guiding the model through carefully designed instructions.

We simplified the NP-hard microservice deployment delay optimization problem into smaller sub-problems and used the Chain of Thought (CoT) prompt technique to ensure accurate learning and reasoning. By comparing the generated code against baseline algorithms under identical conditions, we demonstrated that our fine-tuned LLM could handle key parameters such as the number of users, edge nodes, and microservice types with performance on par with human-designed algorithms.

Furthermore, the study shows that incorporating prior knowledge into a structured question-answer format significantly enhances the LLM's problem-solving capabilities. The generated deep reinforcement learning framework achieved locally optimal solutions for practical deployment problems. Manual evaluation and rigorous comparisons with existing algorithms confirmed the effectiveness of our approach, highlighting the potential of LLMs in edge computing optimization tasks.

Looking ahead, there are several promising directions for future research. One critical area of focus is improving the security and privacy of the code generated by LLMs, especially in sensitive deployment environments. Techniques such as federated learning, homomorphic encryption, and differential privacy could be explored to ensure that data and models are protected without compromising performance. Additionally, enhancing the robustness of LLMs to mitigate issues like hallucinations and erroneous outputs remains a key challenge. Investigating the use of more advanced validation techniques or combining LLMs with rule-based systems may further improve code accuracy and reliability. Moreover, expanding the scope of LLMs to address real-time optimization and resource management in dynamic edge environments presents another exciting avenue for future exploration.

By addressing these challenges, we believe that LLMs can play an even more impactful role in edge computing, offering scalable, secure, and efficient solutions.

Author Contributions: Conceptualization, K.F. and Y.X.; methodology, L.L.; software, L.L.; validation, B.L., K.L. and K.P.; formal analysis, X.H.; investigation, Z.Z.; writing—original draft preparation, L.L.; writing—review and editing, K.F. and B.X.; supervision, K.P.; funding acquisition, Y.X. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded in part by the Key Research and Development Program of Hubei Province under grant 2022BAA038, in part by the Key Research and Development Program of Hubei Province under grant 2023BAB074, in part by the Key Research and Development Program of Hubei Province under grant 2024BAB031, in part by the special fund for Wuhan Artificial Intelligence Innovation under grant 2022010702040061.

Data Availability Statement: The datasets presented in this article are not readily available because the data are part of an ongoing study. Requests to access the datasets should be directed to contact with us.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Mao, Y.; You, C.; Zhang, J.; Huang, K.; Letaief, K.B. A survey on mobile edge computing: The communication perspective. *IEEE Commun. Surv. Tutor.* **2017**, *19*, 2322–2358. [CrossRef]
2. Soldani, J.; Tamburri, D.A.; Van Den Heuvel, W.J. The pains and gains of microservices: A systematic grey literature review. *J. Syst. Softw.* **2018**, *146*, 215–232. [CrossRef]
3. Khan, M.G.; Taheri, J.; Al-Dulaimy, A.; Kassler, A. Perfsim: A performance simulator for cloud native microservice chains. *IEEE Trans. Cloud Comput.* **2021**, *11*, 1395–1413. [CrossRef]
4. Luo, S.; Xu, H.; Lu, C.; Ye, K.; Xu, G.; Zhang, L.; Ding, Y.; He, J.; Xu, C. Characterizing microservice dependency and performance: Alibaba trace analysis. In Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, 1–4 November 2021; pp. 412–426.
5. Peng, K.; Wang, L.; He, J.; Cai, C.; Hu, M. Joint optimization of service deployment and request routing for microservices in mobile edge computing. *IEEE Trans. Serv. Comput.* **2024**, *17*, 1016–1028. [CrossRef]
6. Li, B.; He, Q.; Cui, G.; Xia, X.; Chen, F.; Jin, H.; Yang, Y. READ: Robustness-oriented edge application deployment in edge computing environment. *IEEE Trans. Serv. Comput.* **2020**, *15*, 1746–1759. [CrossRef]
7. Xiao, Y.; Zhang, Q.; Liu, F.; Wang, J.; Zhao, M.; Zhang, Z.; Zhang, J. NFVdeep: Adaptive online service function chain deployment with deep reinforcement learning. In Proceedings of the International Symposium on Quality of Service, Phoenix, AZ, USA, 24–25 June 2019; pp. 1–10.
8. Xu, B.; Hu, Y.; Hu, M.; Liu, F.; Peng, K.; Liu, L. Iterative dynamic critical path scheduling: An efficient technique for offloading task graphs in mobile edge computing. *Appl. Sci.* **2022**, *12*, 3189. [CrossRef]
9. Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F.L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. Gpt-4 technical report. *arXiv* **2023**, arXiv:2303.08774.
10. Alahmadi, A.; Alnowiser, A.; Zhu, M.M.; Che, D.; Ghodous, P. Enhanced first-fit decreasing algorithm for energy-aware job scheduling in cloud. In Proceedings of the 2014 International Conference on Computational Science and Computational Intelligence, Las Vegas, NV, USA, 10–13 March 2014; IEEE: Piscataway, NJ, USA , 2014; Volume 2, pp. 69–74.
11. Wei, J.; Tay, Y.; Bommasani, R.; Raffel, C.; Zoph, B.; Borgeaud, S.; Yogatama, D.; Bosma, M.; Zhou, D.; Metzler, D.; et al. Emergent abilities of large language models. *arXiv* **2022**, arXiv:2206.07682.
12. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
13. Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; et al. Code llama: Open foundation models for code. *arXiv* **2023**, arXiv:2308.12950.
14. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. Competition-level code generation with alphacode. *Science* **2022**, *378*, 1092–1097. [CrossRef] [PubMed]
15. Floridi, L.; Chiriatti, M. GPT-3: Its nature, scope, limits, and consequences. *Minds Mach.* **2020**, *30*, 681–694. [CrossRef]
16. Schuett, J.; Dreksler, N.; Anderljung, M.; McCaffary, D.; Heim, L.; Bluemke, E.; Garfinkel, B. Towards best practices in AGI safety and governance. *arXiv* **2023**, arXiv:2305.07153.
17. Cao, Y.; Li, S.; Liu, Y.; Yan, Z.; Dai, Y.; Yu, P.S.; Sun, L. A comprehensive survey of ai-generated content (aigc): A history of generative ai from gan to chatgpt. *arXiv* **2023**, arXiv:2303.04226.
18. Wu, C.; Yin, S.; Qi, W.; Wang, X.; Tang, Z.; Duan, N. Visual chatgpt: Talking, drawing and editing with visual foundation models. *arXiv* **2023**, arXiv:2303.04671.
19. Conover, M.; Hayes, M.; Mathur, A.; Xie, J.; Wan, J.; Shah, S.; Ghodsi, A.; Wendell, P.; Zaharia, M.; Xin, R. Free dolly: Introducing the world's first truly open instruction-tuned llm. *Co. Blog Databricks* **2023**.
20. Sanh, V.; Webson, A.; Raffel, C.; Bach, S.H.; Sutawika, L.; Alyafeai, Z.; Chaffin, A.; Stiegler, A.; Scao, T.L.; Raja, A.; et al. Multitask prompted training enables zero-shot task generalization. *arXiv* **2021**, arXiv:2110.08207.
21. Wang, Y.; Mishra, S.; Alipoormolabashi, P.; Kordi, Y.; Mirzaei, A.; Arunkumar, A.; Ashok, A.; Dhanasekaran, A.S.; Naik, A.; Stap, D.; et al. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. *arXiv* **2022**, arXiv:2204.07705.
22. Huang, L.; Bras, R.L.; Bhagavatula, C.; Choi, Y. Cosmos QA: Machine reading comprehension with contextual commonsense reasoning. *arXiv* **2019**, arXiv:1909.00277.

23. Rücklé, A.; Geigle, G.; Glockner, M.; Beck, T.; Pfeiffer, J.; Reimers, N.; Gurevych, I. Adapterdrop: On the efficiency of adapters in transformers. *arXiv* **2020**, arXiv:2010.11918.
24. Mahabadi, R.K.; Ruder, S.; Dehghani, M.; Henderson, J. Parameter-efficient multi-task fine-tuning for transformers via shared hypernetworks. *arXiv* **2021**, arXiv:2106.04489.
25. Sakaguchi, K.; Bras, R.L.; Bhagavatula, C.; Choi, Y. Winogrande: An adversarial winograd schema challenge at scale. *Commun. ACM* **2021**, *64*, 99–106. [CrossRef]
26. Gao, T.; Fisch, A.; Chen, D. Making pre-trained language models better few-shot learners. *arXiv* **2020**, arXiv:2012.15723.
27. Gu, Y.; Han, X.; Liu, Z.; Huang, M. Ppt: Pre-trained prompt tuning for few-shot learning. *arXiv* **2021**, arXiv:2109.04332.
28. Tan, Z.; Zhang, X.; Wang, S.; Liu, Y. MSP: Multi-stage prompting for making pre-trained language models better translators. *arXiv* **2021**, arXiv:2110.06609.
29. Li, X.L.; Liang, P. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv* **2021**, arXiv:2101.00190.
30. Liu, P.; Yuan, W.; Fu, J.; Jiang, Z.; Hayashi, H.; Neubig, G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.* **2023**, *55*, 1–35. [CrossRef]
31. Liu, X.; Ji, K.; Fu, Y.; Tam, W.L.; Du, Z.; Yang, Z.; Tang, J. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *arXiv* **2021**, arXiv:2110.07602.
32. Wang, Y.; Shi, W.; Hu, M. Virtual servers co-migration for mobile accesses: Online versus off-line. *IEEE Trans. Mob. Comput.* **2015**, *14*, 2576–2589. [CrossRef]
33. Hu, M.; Guo, Z.; Wen, H.; Wang, Z.; Xu, B.; Xu, J.; Peng, K. Collaborative Deployment and Routing of Industrial Microservices in Smart Factories. *IEEE Trans. Ind. Inform.* **2024**. [CrossRef]
34. Hu, M.; Luo, J.; Wang, Y.; Veeravalli, B. Adaptive scheduling of task graphs with dynamic resilience. *IEEE Trans. Comput.* **2016**, *66*, 17–23. [CrossRef]
35. Hong, S.; Zheng, X.; Chen, J.; Cheng, Y.; Wang, J.; Zhang, C.; Wang, Z.; Yau, S.K.S.; Lin, Z.; Zhou, L.; et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv* **2023**, arXiv:2308.00352.
36. Arulkumaran, K.; Deisenroth, M.P.; Brundage, M.; Bharath, A.A. A brief survey of deep reinforcement learning. *arXiv* **2017**, arXiv:1708.05866. [CrossRef]
37. Xia, L.; Xu, J.; Lan, Y.; Guo, J.; Cheng, X. Learning maximal marginal relevance model via directly optimizing diversity evaluation measures. In Proceedings of the 38th international ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, 9–13 August 2015; pp. 113–122.
38. Peng, K.; He, J.; Guo, J.; Liu, Y.; He, J.; Liu, W.; Hu, M. Delay-Aware Optimization of Fine-Grained Microservice Deployment and Routing in Edge via Reinforcement Learning. *IEEE Trans. Netw. Sci. Eng.* **2024**. [CrossRef]
39. Hoffmann, J.; Borgeaud, S.; Mensch, A.; Buchatskaya, E.; Cai, T.; Rutherford, E.; Casas, D.d.L.; Hendricks, L.A.; Welbl, J.; Clark, A.; et al. Training compute-optimal large language models. *arXiv* **2022**, arXiv:2203.15556.
40. Ding, N.; Qin, Y.; Yang, G.; Wei, F.; Yang, Z.; Su, Y.; Hu, S.; Chen, Y.; Chan, C.M.; Chen, W.; et al. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *arXiv* **2022**, arXiv:2203.06904.
41. Han, X.; Meng, X.; Yu, Z.; Kang, Q.; Zhao, Y. A service function chain deployment method based on network flow theory for load balance in operator networks. *IEEE Access* **2020**, *8*, 93187–93199. [CrossRef]
42. Fu, T.Z.; Ding, J.; Ma, R.T.; Winslett, M.; Yang, Y.; Zhang, Z. DRS: Auto-scaling for real-time stream analytics. *IEEE/ACM Trans. Netw.* **2017**, *25*, 3338–3352. [CrossRef]
43. Xu, B.; Guo, J.; Ma, F.; Hu, M.; Liu, W.; Peng, K. On the Joint Design of Microservice Deployment and Routing in Cloud Data Centers. *J. Grid Comput.* **2024**, *22*, 42. [CrossRef]
44. Wang, S.; Guo, Y.; Zhang, N.; Yang, P.; Zhou, A.; Shen, X. Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach. *IEEE Trans. Mob. Comput.* **2019**, *20*, 939–951. [CrossRef]
45. Hu, M.; Luo, J.; Wang, Y.; Veeravalli, B. Practical resource provisioning and caching with dynamic resilience for cloud-based content distribution networks. *IEEE Trans. Parallel Distrib. Syst.* **2013**, *25*, 2169–2179. [CrossRef]
46. Zhao, W.X.; Zhou, K.; Li, J.; Tang, T.; Wang, X.; Hou, Y.; Min, Y.; Zhang, B.; Zhang, J.; Dong, Z.; et al. A survey of large language models. *arXiv* **2023**, arXiv:2303.18223.
47. Lv, W.; Wang, Q.; Yang, P.; Ding, Y.; Yi, B.; Wang, Z.; Lin, C. Microservice deployment in edge computing based on deep Q learning. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 2968–2978. [CrossRef]
48. Driess, D.; Xia, F.; Sajjadi, M.S.; Lynch, C.; Chowdhery, A.; Ichter, B.; Wahid, A.; Tompson, J.; Vuong, Q.; Yu, T.; et al. Palm-e: An embodied multimodal language model. *arXiv* **2023**, arXiv:2303.03378.
49. Hu, Y.; Wang, H.; Wang, L.; Hu, M.; Peng, K.; Veeravalli, B. Joint deployment and request routing for microservice call graphs in data centers. *IEEE Trans. Parallel Distrib. Syst.* **2023**, *34*, 2994–3011. [CrossRef]