

Resource Optimization in Large Language Model Deployment Using Reinforcement Learning and Adaptive Software Engineering

Dr. Parvaneh Asghari

Department of Computer Engineering, CT.C., Islamic Azad
University, Tehran, Iran
p_asghari@iauctb.ac.ir

Alireza Rahimpour Anaraki

Department of Computer Engineering, CT.C., Islamic Azad
University, Tehran, Iran
arahimpourdev@gmail.com

Abstract— Large Language Models (LLMs) are extremely resource-intensive to deploy, demanding high memory and compute. Static provisioning often leads to waste or unmet demand. We propose a conceptual framework that uses reinforcement learning (RL) and self-adaptive software engineering to optimize resource use in LLM deployments. An RL agent monitors system metrics (throughput, latency, GPU/CPU utilization) and takes actions such as scaling instances, adjusting model precision, or modifying batch sizes. The system employs a Monitor-Analyze-Plan-Execute (MAPE-K) loop where dynamic configuration parameters are tuned online to maximize throughput and minimize cost. We illustrate the approach with examples: RL-driven autoscaling (showing ~40–50% higher GPU utilization) and adaptive inference optimizations like key-value caching (up to 4× speedup). Real-world LLM deployments (cloud services and edge settings) exhibit highly variable workloads; our framework adapts to these changes. Experiments and industry reports show that RL-based adaptation can significantly improve resource efficiency and performance.

I. INTRODUCTION

Large Language Models (LLMs) such as GPT and LLaMA have revolutionized NLP but are huge: models can have tens of billions of parameters, requiring hundreds of GB of memory for inference. For example, loading a 70B-parameter model can demand 256 GB of RAM while even top-tier GPUs only have ~80 GB. Moreover, LLM inference is slow and latency-sensitive as tokens are generated one by one, and the input context grows over time. In production, traffic to LLM APIs is bursty and unpredictable, making static resource allocation inefficient or unreliable. Traditional DevOps

practices rely on fixed over-provisioning or simple autoscalers, which either waste resources or fail to meet service-level objectives (SLOs) during peaks.

Reinforcement learning (RL) offers a promising solution for dynamic, feedback-driven resource management. Recent work shows that RL can learn effective autoscaling and resource allocation policies for cloud applications. We adopt a self-adaptive software engineering perspective (MAPE-K loop) where an RL agent continuously adjusts system configuration based on observed performance. In our framework, the agent observes metrics (throughput, latency, utilization, SLO violations) and acts by tuning parameters such as the number of model replicas, tensor precision, or inference cache policies. A reward function balances service quality with cost.

This article presents a conceptual framework and implementation strategies for applying RL and adaptive software engineering to optimize LLM deployment. We review relevant literature on RL in cloud resource scheduling, adaptive system design, and LLM inference optimizations. We then describe our architecture: an RL controller integrated with the LLM inference platform (e.g. serving stack or Kubernetes cluster). Finally, we discuss real-world evaluation examples and practical deployment considerations. Our contributions include: (1) identifying key dynamic configuration knobs for LLMs, (2) framing their tuning as an RL problem within a MAPE-K loop, and (3) illustrating this with examples from cloud and edge deployments showing resource and performance gains.

II. STRUCTURE OF THE PAPER

The remainder of this paper is organized as follows. Section II provides a comprehensive review of the theoretical foundations and prior research related to self-adaptive software systems, particularly focusing on the MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge base) loop and its integration with reinforcement learning techniques for dynamic resource management in cloud environments. Section III introduces the proposed conceptual framework, detailing the architecture, components, and operation of a reinforcement learning-driven self-adaptive system for Large Language Model (LLM) deployment. Section IV presents a set of real-world evaluation scenarios and practical examples that illustrate the effectiveness of the framework in improving resource utilization and performance in both cloud and edge settings. Section V discusses the broader implications of the proposed approach, including implementation challenges, reward design considerations, monitoring requirements, and the integration with emerging LLMOps practices. Finally, Section VI concludes the paper by summarizing the key contributions and outlining potential directions for future work aimed at enhancing the scalability, efficiency, and adaptability of LLM deployment infrastructures.

III. BACKGROUND AND RELATED WORK

Self-Adaptive Systems and MAPE-K. Self-adaptive software monitors itself and its environment to meet goals under changing conditions. A canonical control model is the MAPE-K loop (Monitor-Analyze-Plan-Execute with shared Knowledge). In this paradigm, a Monitor gathers runtime data, Analyze determines if adaptation is needed, Plan devises changes, and Execute applies them. RL can be viewed as a learnable control policy within this loop. Instead of fixed rules, the agent acts as the analyzing/planning component that learns which actions maximize long-term reward (e.g., throughput minus cost).

RL for Cloud Resource Management. RL has been widely applied to dynamic resource allocation in clouds and distributed systems. For instance, Zhou et al. survey deep RL methods for cloud scheduling, noting RL's ability to handle complex dynamics where heuristics fail. Autonomic autoscalers using RL have been demonstrated to improve utilization and SLO compliance. Qiu et al. introduced AWARE, an extensible RL autoscaling framework that leverages meta-learning for rapid adaptation: in production tests, AWARE achieved 47.5% higher CPU and 39.2% higher memory utilization while drastically reducing SLO violations. In edge-cloud scenarios, Zhao et al. showed that co-optimizing DVFS (dynamic voltage/frequency scaling) and workload offloading with deep RL (DVFO system) cut inference energy by ~33% and latency by ~30–60%. These examples motivate RL for fine-grained, context-aware scaling of LLM services.

LLM Inference Optimization. Independent of RL, practitioners use numerous static optimizations for LLM

deployment. For example, key-value caching reuses attention computations across tokens. As illustrated in Figure 2, during generation the model stores past attention keys/values so only new ones are computed at each step. This cache-based approach avoids redundant work, yielding up to 4× speedups when combined with compiler optimizations. Similarly, techniques like quantization (reducing precision) and tensor parallelism spread computation across GPUs to fit large models on limited hardware. Dynamic batching and pipeline parallelism also trade latency for throughput. However, these optimizations are typically applied in a fixed manner, not adapting at runtime to workload changes.

Deployment and Security Challenges. Deploying LLMs at scale also raises operational issues. Mirsky et al. find that many public LLM services are misconfigured or insecure, often exposing endpoints without authentication. Moreover, managing a fleet of GPUs and models reliably requires automated orchestration. Our framework focuses on resource efficiency, but any dynamic adaptation must be integrated with best practices for robust deployment (e.g. secure defaults, monitoring) as highlighted by recent studies.

Persian-language Research. The Iranian research community has also explored RL for cloud resource allocation. For example, a Persian conference paper presents a reinforcement-learning approach to dynamic resource provisioning in cloud computing, aimed at minimizing energy cost and improving utilization. Another study proposed a MAPE-loop-based RL framework for auto-scaling cloud clusters, showing improved utilization under fluctuating demand. These works indicate local interest in self-adaptive cloud systems and motivate adapting such ideas to LLM services.

IV. PROPOSED CONCEPTUAL FRAMEWORK

We propose a self-adaptive architecture for LLM deployment that intertwines an RL controller with the inference engine (Figure 1). An RL agent continually observes system and workload metrics (e.g. request rate, queue length, GPU mem/utilization, token latency). At each decision step, the agent chooses actions that reconfigure the system. Actions may include: scaling the number of model instances or containers; adjusting the batch size; enabling/disabling features like KV caching; switching between precisions (FP16 vs FP32); tuning thread counts; or migrating models between nodes. The chosen action changes the managed system (the LLM service), which in turn alters performance and cost. A reward is computed (e.g. negative cost + throughput bonus) and fed back to the agent. Over time, the agent learns a policy that balances performance against resource usage.

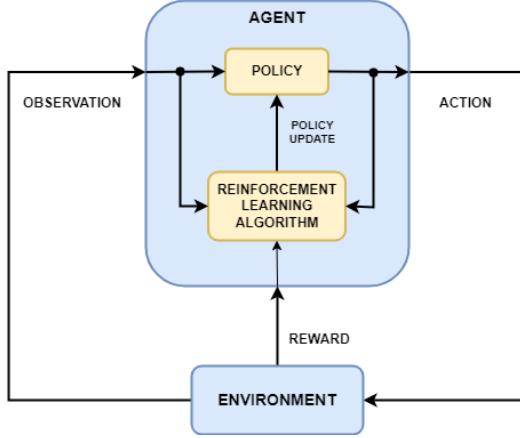


Figure 1. Reinforcement learning agent-environment loop for adaptive LLM deployment. The agent’s policy (yellow box) maps observed states (GPU/CPU loads, queue sizes, etc.) to actions (scaling, reconfiguration). After each action, the environment returns a reward based on performance (throughput/latency and cost). The RL algorithm (bottom) updates the policy to maximize cumulative reward.

Implementing this framework involves (1) Monitoring: collecting runtime data via telemetry or logs (the “M” step in MAPE-K); (2) Analysis/Planning: the RL agent evaluates policies and simulates possible reconfigurations; and (3) Execution: applying changes through orchestration APIs (e.g. Kubernetes autoscaler, gRPC commands to inference servers). We can use deep RL techniques (e.g. Deep Q-Networks, policy gradients) to handle large, continuous state spaces. In practice, one may bootstrap the agent from historical traces or use transfer learning between workloads to speed up learning, as suggested by AWARE.

. The state space should capture relevant dynamics: e.g. recent CPU/GPU utilization, queue backlog, average token latency, and workload trend indicators. Actions should be discrete for RL simplicity (e.g. “scale +1 instance”, “increase batch size by 16”). The reward function can penalize SLO violations and cost (idle hardware). The framework is extensible: multiple agents could manage different parts (model instances vs data pipelines), coordinating via shared knowledge if needed.

V. REAL-WORLD EVALUATION EXAMPLES

We illustrate the potential of RL-adaptive deployment with examples from actual LLM and related systems. Autoscaling with RL: In [1], Qiu et al. evaluated their AWARE framework on real cloud workloads. Compared to rule-based autoscalers, AWARE’s RL agent achieved 47.5% higher CPU utilization and 39.2% higher memory utilization while reducing service-level objective (SLO) violations by a factor of 16.9. This demonstrates how learned policies can provision resources more tightly than static heuristics. An analogous experiment

for LLM could involve an RL agent tuning the number of LLM replicas based on query load.

Efficient Inference Techniques: Consider key-value caching: Figure 2 illustrates KV caching during LLM generation. At each new token, the model reuses previously computed keys/values (purple) instead of recomputing them. By preallocating the cache (static allocation) and enabling just-in-time compilation, HuggingFace reports up to 4× speedups. In practice, our RL agent could decide dynamically whether to enable static vs dynamic caching based on input sequence lengths or GPU memory pressure. Other inference optimizations (mixed-precision, tensor parallelism, quantization) could similarly be turned on/off by the agent as conditions change.

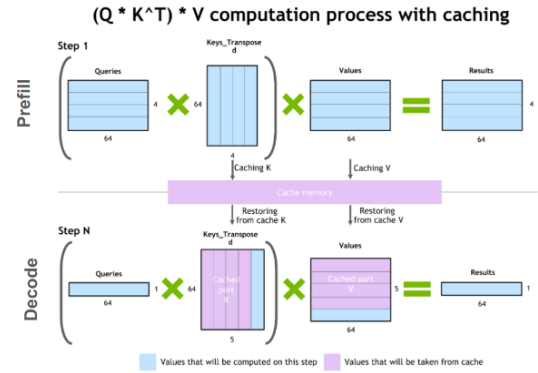


Figure 2. Key-Value caching in LLM inference (NVIDIA). The prefill phase computes all key/value pairs for input tokens in parallel (top). In the decode phase, only new token queries (Q) are multiplied by cached keys (K) and values (V) to generate the output, avoiding full recomputation.

Edge-Cloud Collaborative Inference: Edge deployments of LLMs (or smaller language models) must balance compute and power. Zhao et al.’s DVFO system co-optimizes CPU/GPU frequencies and offloading of work to the cloud using deep RL. They report 33% energy savings with modest latency tradeoffs. Similarly, an RL agent could adaptively throttle GPU clocks or offload large requests to remote servers for an LLM service running on limited hardware.

Deployment Strategy: In another study, Hou et al. analyzed over 300,000 real-world LLM services and found many insecure or misconfigured deployments. While not about optimization per se, this highlights the need for deployment-aware adaptation. Our framework assumes the infrastructure supports safe dynamic reconfiguration (e.g. zero-downtime scaling, secure communication). In practice, one would combine RL tuning with secure-by-default settings and human-in-the-loop validation for critical updates.

Overall, these examples suggest that RL-driven adaptation can yield substantial efficiency gains. For instance, AWARE’s

RL policy cut resource waste dramatically, and DVFO's RL reduced energy. In the context of LLMs, even modest improvements (e.g. 20–30% better GPU utilization, or 2× throughput for the same hardware) would be impactful. Our conceptual framework lays the groundwork to achieve such gains by making the LLM deployment self-tuning.

VI. DISCUSSION

The proposed RL-adaptive approach offers flexibility but also entails challenges. Training Overhead and Convergence: Training RL agents can be time-consuming and data-hungry. Techniques like transfer learning or meta-learning (as used in AWARE) can mitigate this by initializing agents with prior experience. One might first train policies offline on simulated loads before online deployment. Care must be taken to ensure safe exploration – e.g. using conservative “safe RL” to avoid disastrous actions in production.

Monitoring and Observability: Effective adaptation depends on accurate metrics. We assume fine-grained telemetry (CPU/GPU metrics, queue lengths, request latencies). Instrumentation overhead should be minimal. Adaptive systems also need robust logging and alerts; misadaptation must be detected. The knowledge base in MAPE-K must store historical data and models of the system.

Reward Design: Choosing the right reward is critical. A simple reward might combine negative cost (idle resources, energy) and positive terms for throughput or latency satisfaction. However, multi-objective rewards (e.g. cost vs latency) may need careful weighting. In practice, one can define acceptable SLO thresholds (hard constraints) and let the reward focus on cost minimization subject to meeting those SLOs.

Multi-agent and Hierarchical RL: In large deployments with many nodes or services, a single RL agent may not scale. A hierarchical or multi-agent approach could help: e.g. one agent per cluster region, coordinating via shared goals. Each agent can optimize local resources (microservices, GPUs) while a higher-level agent balances global traffic.

Dynamic Environment: Workloads and hardware can change over time. The RL agent should continuously learn (online learning) to adapt to workload drift. Approaches like continual learning or resettable environments can be used. Periodic retraining or confidence checks can ensure the policy remains valid.

Integration with LLMops: Just as MLOps has become crucial for ML training, LLMops (operational management of LLM services) is emerging. Our RL framework complements LLMops by automating infrastructure tuning. However, it should integrate with conventional DevOps pipelines (CI/CD for models, canary testing of new policies, etc.). Tools like Kubernetes custom controllers or serverless function orchestrators can implement the agent's actions.

Limitations: Our discussion is conceptual; implementing RL controllers in live systems is nontrivial. There may be latency in applying actions, network delays, and partial observability. Simulation environments (digital twins) can help validate policies before live deployment. We also rely on the assumption that any action (e.g. scaling or frequency change) has a predictable effect; inaccuracies can degrade learning.

Despite challenges, the advantages are compelling: Adaptivity and efficiency. Rather than static configurations, the system automatically adjusts to workload patterns. Over time, it learns specialized policies for different application types (chatbots, summarizers, embeddings). In contrast to fixed rule-based autoscalers, an RL-adaptive system can discover non-intuitive policies (e.g. slightly under-provision during an easy period to save cost) while still meeting SLOs.

VII. CONCLUSION

We have outlined a full framework for resource optimization in LLM deployments through reinforcement learning and adaptive software engineering. By framing LLM serving as a self-adaptive system, with an RL agent in the MAPE-K loop, the deployment becomes capable of autonomously scaling and reconfiguring in response to demand. We discussed key elements: defining states/actions for LLM inference, dynamic configuration knobs (scaling, caching, precision), and reward design. Examples like the AWARE autoscaler and DVFS controllers demonstrate that RL-driven adaptation can substantially improve utilization and latency. In the LLM context, similar gains could translate into faster responses and lower cloud costs.

Future work will involve prototyping such an RL controller in a real LLM service (e.g. using Hugging Face Inference Endpoints or a custom deployment). We plan to explore multi-agent schemes (per-model controllers), safe learning techniques, and integration with monitoring frameworks. Additionally, we aim to incorporate user-centric QoS metrics (e.g. response coherence vs speed) into the reward. As LLMs continue to grow, adaptive deployment strategies will be essential for sustainable, scalable AI services.

Acknowledgment— The authors thank their families for support and the anonymous reviewers for their constructive feedback, which greatly improved this work.

ACKNOWLEDGMENT

The author would like to thank Dr. Parvaneh Asghari for her valuable guidance and continued support throughout the course of this research. Her insights and constructive feedback significantly contributed to the quality of this work.



REFERENCES

- [1] H. Qiu et al., "AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems," in Proc. USENIX ATC '23, Jul. 2023, pp. 387–402.
- [2] G. Zhou, W. Tian, R. Buyya, R. Xue, and L. Song, "Deep reinforcement learning-based methods for resource scheduling in cloud computing: a review and future directions," Artif. Intell. Rev., vol. 57, no. 124, Apr. 2024.
- [3] S. Verma and N. Vaidya, Mastering LLM Techniques: Inference Optimization, NVIDIA Technical Blog, Nov. 17, 2023.
- [4] Hugging Face, "Optimizing inference," Transformers Documentation, 2024, accessed May 2025.
- [5] Y. Zhao, H. Li, and J. Liu, "DVFO: Learning-Based DVFS for Energy-Efficient Edge-Cloud Collaborative Inference," arXiv:2306.01811, 2023.
- [6] X. Hou, J. Han, Y. Zhao, and H. Wang, "Unveiling the Landscape of LLM Deployment in the Wild: An Empirical Study," preprint, 2025.
- [7] S. Abedi, M. G. Arani, and M. Mayin, "رویکردی مبتنی بر یادگیری تقویتی " برای مقیاس‌بندی خودکار منابع در محیط رایانش ابری (Reinforcement learning-based approach for automated resource scaling in cloud computing), in Proc. Int'l Conf. Web Research, 2017, pp. 53–61 (Persian).
- [8] "یک رویکرد برای تأمین پویای منابع در رایانش ابری بر اساس یادگیری تقویتی" (An RL-based approach for dynamic resource provisioning in cloud), Iranian National Conf. on Interdisciplinary Engg., 2016, pp. 112–119 (Persian).