Automating Model Deployment on Mobile Devices

Leming Shen[1,3], Qiang Yang[2], Xinyu Huang[1], Zijing Ma[1],
Yuanqing Zheng[1], Chris Xiaoxuan Lu[3]
[1]The Hong Kong Polytechnic University, [2]University of Cambridge, [3] University College London

Abstract

We present AgentDeploy that aims to deploy AI models on heterogeneous mobile devices by fully automating model optimization and adaptation (O&A) with Large Language Model (LLM) agents. AgentDeploy not only significantly reduces human effort via automation but also enhances inference efficacy by integrating LLMs with device-specific knowledge. To deliver a robust automation framework, AgentDeploy develops novel solutions to address three key technical challenges: 1) To help users elicit clear requirements, we propose a *user requirement standardization* module; 2) To mitigate the environment mismatch between models and heterogeneous mobile devices, we design an *operator-aware model partition* paradigm that enables different model parts to be efficiently executed on distinct inference engines. 3) To avoid re-searching optimization hyperparameters for new tasks, we propose a *configuration reuse* module that heuristically retrieves previous configurations for efficient reuse. Experiments demonstrate that by fully automating the entire O&A process, AgentDeploy reduces entire model adaptation time by up to 80% compared with manual O&A while achieving higher task accuracy (more than 13%) than the baselines.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**; • **Computer systems organization** → **Embedded and cyber-physical systems**.

Keywords

LLM Agent, Model Deployment

ACM Reference Format:
Leming Shen[1,3], Qiang Yang[2], Xinyu Huang[1], Zijing Ma[1],, Yuanqing Zheng[1], Chris Xiaoxuan Lu[3] . 2026. Automating Model Deployment on Mobile Devices. In *The 24th Annual International Conference on Mobile Systems, Applications and Services (MobiSys Workshop '26), June 21–25, 2026, Cambridge, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3812836.3814754

This work is licensed under a Creative Commons Attribution 4.0 International License.
MobiSys Workshop '26, Cambridge, United Kingdom
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2712-2/26/06
https://doi.org/10.1145/3812836.3814754

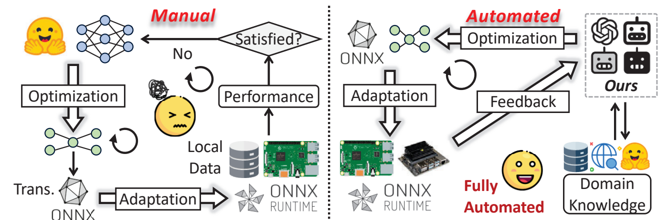


Figure 1: Deploying pre-trained open-source models on mobile devices (manual vs. automated).

1 Introduction

Edge AI has emerged as a transformative paradigm, enabling the deployment of AI models directly on mobile devices. Edge AI facilitates real-time onboard data processing of various sensors and enables low-latency decision-making [16]. In addition, edge AI ensures privacy and security by keeping sensitive data on local devices [14].

To develop an edge AI application, developers can download a model from open-source platforms and deploy the model on target mobile devices. In practice, however, most models cannot be directly deployed on mobile devices for two main reasons. 1) These models often contain a huge amount of parameters, making them hard to fit into resource-constrained mobile devices [8]. 2) These models are primarily available in PyTorch [13] format, which may not be supported by mobile devices with heterogeneous execution environments. To tackle these practical issues, developers need to perform multiple iterations of model optimization and adaptation (O&A), as shown in Fig. 1 (left). To tailor models for resource-constrained devices, developers adopt various optimization techniques with tunable hyperparameters (*e.g.*, pruning rate) to compress the models. To facilitate the portability of the compressed models across heterogeneous devices, they need to translate the compressed models into platform-independent *portable models*, which can be deployed and executed on mobile devices via cross-platform inference engines (*e.g.*, ONNX Runtime [1]). Finally, developers can iteratively adapt the portable models to meet the runtime performance requirements of the mobile devices.

Nevertheless, such an iterative O&A process is cumbersome, error-prone, and inefficient. 1) *Large Search Space*: Manually defining and adjusting the search space of tunable hyperparameters is time-consuming even when assisted by

automated search frameworks [5]. Worse still, highly specialized expertise is required to identify an appropriate search space for each hyperparameter. 2) *Heterogeneous Target Devices*: Heterogeneous mobile devices have diverse execution environments, making the selection and installation of an appropriate inference engine with various dependencies challenging. Mastering and manually configuring these device-specific frameworks demands substantial time and domain expertise. 3) *Laborious Iterative Refinement*: Optimizing an AI model for high performance given the resource constraints of a target device typically necessitates multiple rounds of trial-and-error iterations, which is time-consuming even for experienced experts, let alone novice developers. This motivates the following research question: *Can we automate the O&A process and democratize the AI models that are readily open-sourced yet inaccessible for edge AI applications?*

To fill this gap, our insight is to employ LLM agents to fully automate the entire O&A process. LLMs have demonstrated exceptional language processing and task automation capabilities. Embedded with extensive world knowledge, LLMs can effectively accomplish complex tasks (*e.g.*, program synthesis [15, 17], IoT data interpretation [20]) in response to user requirements. Furthermore, by integrating LLMs with various third-party tools (*e.g.*, web search engines), we can empower LLM agents to facilitate task automation via enriched domain knowledge. To this end, we propose `AgentDeploy`, a multi-agent-assisted model O&A system for heterogeneous edge AI applications, as illustrated in Fig. 1 (right).

Three technical challenges need to be addressed: 1) **Vague User Requirements.** Due to numerous evaluation metrics, developers often struggle to clearly and comprehensively articulate their requirements in a single attempt. 2) **Environment Mismatch.** Models may not be fully compatible with a single inference engine due to mismatched execution runtimes. For example, some models may use the `Mod` operator during inference, which is not supported by TensorRT [2]. Developers have to either wait for engine updates or fallback to inefficient CPU runtimes. 3) **Exhausted Hyperparameter Search.** If `AgentDeploy` performed model O&A from scratch for new tasks, it would become extremely time-consuming due to the large hyperparameter search spaces.

We address these challenges with three novel technical modules (§ 3) and evaluate `AgentDeploy` on 3 mobile devices across various edge AI applications. We compare `AgentDeploy` with two types of baselines: a SOTA LLM agent tailored for coding tasks and human developers. We invite 15 users as human baselines and evenly divide them into three groups based on their development experience and expertise. Experiments demonstrate that `AgentDeploy` can reduce the time required for the O&A process by up to 61% compared to manual execution by experts, significantly improving the model O&A efficiency and reducing the human effort. Moreover, the portable models optimized by `AgentDeploy` outperform baselines in terms of both task accuracy (> 13%) and resource consumption (< 14%).

2 Background & Motivation

In this section, we analyze why executing the O&A process for edge AI is cumbersome, inefficient, and error-prone.

Model Optimization. To reduce resource consumption, developers apply various optimization methods, each with distinct hyperparameters. However, optimal hyperparameter selection is time-consuming even for experienced developers due to two reasons: 1) Determining appropriate hyperparameters from vast search spaces requires highly specialized expertise, as each hyperparameter influences the optimization result differently; 2) Multiple hyperparameter combinations may all satisfy the resource constraints of the target device, making the selection process non-trivial. For example, if the target device operates on battery power, hyperparameters that can minimize energy consumption may be prioritized.

Model Adaptation. To facilitate portability, developers translate optimized models into portable models, which can be deployed on mobile devices via cross-platform inference engines. Nonetheless, manually configuring model execution environments poses three key challenges: 1) Empirically following traditional practices (*e.g.*, directly deploying a PyTorch model on a server installed with commonly used environments) often fails due to hardware incompatibilities, dependency conflicts, and obsolete packages for mobile devices. For instance, Jetson Nano may only support tailored versions of Ubuntu and Python. 2) Mobile devices often exhibit varying affinities for different engines. For example, NVIDIA-developed TensorRT is optimized for NVIDIA Jetson Nano, while ONNX Runtime may be more suitable for Raspberry Pis. Mastering such highly specialized, device-specific knowledge is time-consuming even for experienced developers. 3) Different models contain a diverse set of operators to perform various computation tasks. Similarly, different engines support distinct operators based on their diverse architectures. This indicates that it is impractical to rely on a single engine for various AI models on mobile devices.

Multiple Iterations. The O&A process typically undergoes multiple trial-and-error iterations by experts to optimize the model performance. *Such a labor-intensive process underscores the need for an automated system to enhance overall efficiency.*

3 `AgentDeploy` Design

Fig. 2 shows the overall workflow of `AgentDeploy`. The user only needs to provide the model name, the target mobile device name, and real-time performance requirements. Upon receiving this input, `AgentDeploy` coordinates four agents

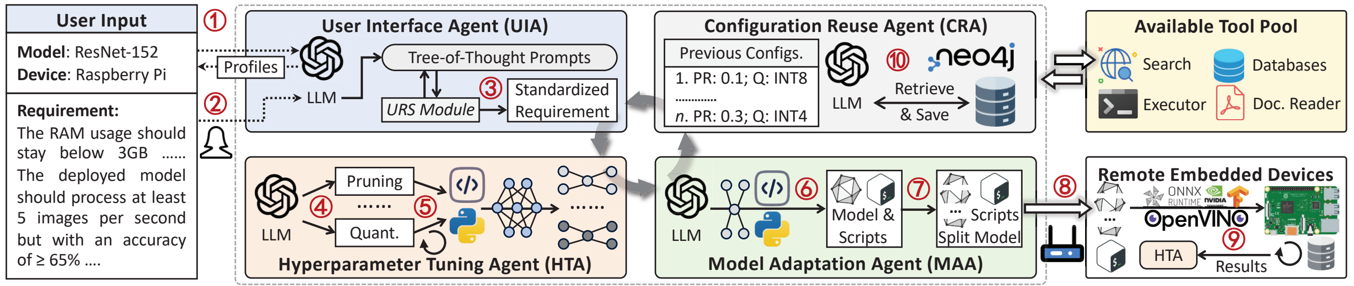


Figure 2: The system overview and workflow of AgentDeploy with four specialized agents.

to automatically execute the O&A process and present the final deployment results. Specifically, *User Interface Agent* **(UIA)** first constructs detailed profiles of the model and the target device (①), serving as references for the user to elicit her deployment requirements (②). UIA then utilizes the *user requirement standardization* module to iteratively refine the requirement into a standardized format with precise semantics (③). *Hyperparameter Tuning Agent* **(HTA)** then generates a list of optimizable hyperparameters (④) and synthesizes a script for automated hyperparameter tuning (⑤). After iterative optimization, multiple optimized models that meet resource constraints are generated. *Model Adaptation Agent* **(MAA)** translates them into portable models and generates a script to split the models into multiple parts via an *operator-aware model partition* module (⑥). MAA then synthesizes a script that can be executed on the target device to perform inference using a *cross-engine inference pipeline* (⑦). The partitioned model and the script are then transmitted to the target device (⑧), which monitors the real-time performance on its local data and sends the results to HTA as feedback to start a new optimization round (⑨). *Configuration Reuse Agent* **(CRA)** further optimizes HTA by reusing previous hyperparameters (⑩) during model optimization, thereby enhancing overall efficiency rather than from scratch.

3.1 User Interface Agent (UIA)

Challenge. The heterogeneous AI models and mobile devices make it difficult even for experienced developers to clearly articulate their requirements. This poses significant challenges for agents in accurately interpreting the requirements and verifying if they are ultimately fulfilled. To tackle this, we employ UIA to help users iteratively elicit clear and structured requirements. UIA contains two main modules: *system profiling* and *user requirement standardization* (URS).

System Profiling presents the resource consumption information of the model and the hardware capabilities of the target device. Specifically, UIA first invokes the web search tool to download the model and then deploys it on a powerful server to perform inference tasks. Runtime resource consumption information (*e.g.*, model size, GPU memory usage) is recorded into a *model profile*. Next, UIA retrieves detailed specs of the target mobile device from its official website. Then, we utilize few-shot prompts [6] to instruct UIA to process and format the retrieved contents into a structured *device profile*, including key specifications such as CPU, GPU, RAM, disk storage, and I/O ports.

User Requirement Standardization (URS). We adopt a Tree-of-Thought (ToT) [21] method to elicit clear and consistent requirements from the user. This approach helps users sort out ideas and facilitates systematic requirement refinement via multi-path reasoning, effectively resolving ambiguities and inconsistencies. **ToT 1**: UIA presents the system profiles for the user as references. **ToT 2**: UIA analyzes the system profiles and generates a list of possible metrics (*e.g.*, classification accuracy). The user can then selectively specify her requirement for those metrics. **ToT 3**: UIA is prompted to detect ambiguities, missing details, and inconsistencies in the requirements. Based on the detection results, UIA is further instructed to provide examples of well-defined requirements and pose follow-up questions to elicit further clarification from the user. Through multiple iterations of the "requirement → detection → question → clarification" cycle, user requirements are progressively refined into a clear format.

3.2 Hyperparameter Tuning Agent (HTA)

Challenge. HTA tailors models for resource-constrained devices by tuning various optimization hyperparameters. However, selecting optimal hyperparameters requires highly specialized domain expertise. Thus, we meticulously design several ToT prompts and instruct HTA to comprehensively analyze the system profiles and resource requirements.

Hyperparameter Determination (ToT 1). HTA is first prompted to generate a comprehensive tunable hyperparameter set that can reduce resource consumption of the model.

Search Space Generation (ToT 2). For each hyperparameter, we use Chain-of-Thought [19] prompts for HTA to: 1) Analyze the resource profiles of the original model and the resource requirements; 2) Generate an appropriate search range for the hyperparameter. For instance, if the parameter size of the original model is 100 MB while the user specifies

that the parameter size of the optimized model should be less than 80 MB, the pruning rate can be within $[0.2, 0.5]$.
Optimal Hyperparameter Searching (ToT 3). We prompt HTA to create a comprehensive multi-objective optimization function that takes tunable hyperparameters as inputs with the aim of minimizing the resource consumption of the optimized model. Accordingly, HTA generates a script that performs iterative hyperparameter tuning to identify multiple hyperparameter sets that satisfy all resource requirements. Finally, HTA synthesizes a script that translates the original model into multiple optimized portable models by applying the identified hyperparameter sets. The portable models are then sent to MAA for model adaptation.

3.3 Model Adaptation Agent (MAA)

Challenge. In practice, portable models' operators may not be fully supported by a single inference engine. Existing solutions either rely on fallback runtimes (*e.g.*, CPU) or continuously develop new cross-platform frameworks to support additional operators, both of which are inefficient and cumbersome. To tackle this, we propose an *operator-aware model partition* module in MAA that divides the portable model into multiple partitions, each compatible with a distinct engine. Accordingly, we design a *cross-engine inference pipeline* to arrange the model partitions and perform inference tasks on the target device by switching between engines.
Operator-Aware Model Partition (OAMP). We perform *operator availability check* to analyze the operators in the portable model and those supported by engines. We first instruct MAA to analyze the portable model and produce a list of its operators. Next, MAA is prompted to generate a list of engines compatible with the target device. For each engine, MAA retrieves a detailed list of supported operators. Based on this information, we propose a *greedy partition algorithm* to minimize the total inference latency while preserving operator-engine compatibility. Specifically, for each portable model operator, three cases are considered: 1) If it is supported by multiple engines, it is assigned to the one with the lowest latency. 2) If the operator is supported by only one engine, it is directly assigned to that engine. 3) If the operator is not supported by any available engine, it is assigned to CPU. After such a greedy approach, MAA generates a list of model partitions with their corresponding engines.
Cross-Engine Inference Pipeline (CEIP). To ensure efficient data flow between model partitions during inference, we propose the CEIP module to implement a pipelined inference paradigm across multiple engines. Specifically, MAA first loads each model partition onto its assigned engine, ensuring that memory and data alignment are consistent across different execution environments. Inference is then performed sequentially or concurrently, depending on the dependency structure of the portable model. This design effectively exploits the strengths of different engines to maximize inference efficiency and reduce execution latency.
Evaluation. With the model partitions and the inference pipeline, the target mobile device performs inference on its local data and monitors the real-time performance, which is then sent to MAA. *For each portable model's monitoring results, MAA judges if all performance requirements are satisfied. If yes, the hyperparameter configuration is delivered to CRA and the monitoring results are presented to the user.*

3.4 Configuration Reuse Agent (CRA)

Challenge. During model optimization, we find that hyperparameter tuning often becomes time-consuming. This is because `AgentDeploy` currently generates a completely new search space for each user request, resulting in heavy computational overhead. *Can we progressively accelerate hyperparameter tuning to further enhance the efficiency of* `AgentDeploy`*?* Our insight is that, as system utilization increases, `AgentDeploy` accumulates numerous historical hyperparameters, which can potentially be reused for new but similar models. To this end, we develop CRA that maintains two databases storing *system profiles* and *hyperparameters*.
Profile Reuse. Model profiles record the resource consumption of the original model, while device profiles include specs of the target device. For each unique model and mobile device, CRA stores the constructed profiles in a database. When a user specifies the same model or device, CRA retrieves the previously stored profiles from the database and delivers them to UIA and HTA for further processing.
Hyperparameter Reuse. We model each hyperparameter tuning process as a finite state machine (FSM) and store the constructed FSM in a graph database. When optimizing a new large model, CRA first retrieves several similar FSMs and then accordingly determines narrowed search spaces for efficient hyperparameter tuning. Specifically, an FSM can be regarded as a unique feature for each model after hyperparameter tuning, which is represented as:

$$H = (S, s_0, \delta, F) \quad s_i \xrightarrow{\delta} s_j, \quad F = \{f_1, f_2, \cdots, f_q\} \subset S \tag{1}$$

where S is a set of states (optimized models), s_0 represents the initial state (the original model), δ is the state transition function that transitions from s_i to s_j by modifying tunable hyperparameters, and F is a set of final states (models that satisfy resource requirements). With the automatically constructed FSM by HTA after hyperparameter tuning, CRA stores it into a dynamically maintained graph database $\mathcal{D}$. Next, given the profile of a new model for optimization, CRA searches the FSM database and retrieves the top-k FSMs whose initial resource profiles are similar to the current one. Specifically, considering each resource metric (*e.g.*, model size, RAM, GPU memory) in the profile, we define resource

profile similarity (RPS) between the new model and each FSM in the database as:

$$RPS_i = \frac{|r_i^1 - r^1|}{r^1} + \frac{|r_i^2 - r^2|}{r^2} + \cdots + \frac{|r_i^N - r^N|}{r^N} \; \forall r_i^j \in \mathcal{D} \quad (2)$$

where r^N is the consumption of the N-th resource metric of the new model, r_i^N is the consumption of the N-th resource metric of the i-th FSM. A smaller RPS value indicates a higher similarity between the new model and the FSM. The selected top-k similar FSMs are $\mathcal{T} = \{H_1, H_2, \cdots, H_k\}$. CRA then finds an appropriate hyperparameter set for reuse. It first performs a breadth-first search (BFS) from each FSM's initial state and obtains a state list with their hyperparameters:

$$\begin{aligned} &\mathcal{T} = \{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_k\}, \; \forall \mathcal{S}_i \subset S_i \\ &\mathcal{S}_i = \{s = \{\gamma_1^i, \gamma_2^i, \cdots, \gamma_m^i\} \mid \forall s \text{ adjacent to } s_0^i, \; s_0^i \in S_i\} \\ &\Gamma_m = \{\gamma_m^1, \gamma_m^2, \cdots, \gamma_m^i\}, \; \forall \gamma_m^i \in \mathcal{S}_i \end{aligned} \quad (3)$$

where $\mathcal{S}_i$ is a set of states adjacent to the initial state s_0^i from the i-th FSM, γ_m^i is the value of the m-th hyperparameter of an adjacent state, and Γ_m is a list of the m-th hyperparameter from all adjacent states. The lower bound γ_m^l of the m-th search space is defined as the minimum value of Γ_m. The rationale is that, during tuning, hyperparameters are often initialized with small values to minimize performance degradation. Consequently, the states obtained after BFS contain smaller values among all the hyperparameters. Next, CRA traverses the entire FSM via a depth-first search and record the maximum value of the m-th hyperparameter, serving as its searching upper bound: $\gamma_m^u = \max\{\gamma_m^1, \gamma_m^2, \cdots, \gamma_m^j, \cdots\}$, where γ_m^j denotes the j-th value of the m-th hyperparameter value in the traversed path. As such, the search space for the m-th hyperparameter is narrowed and confined to $[\gamma_m^l, \gamma_m^u]$.

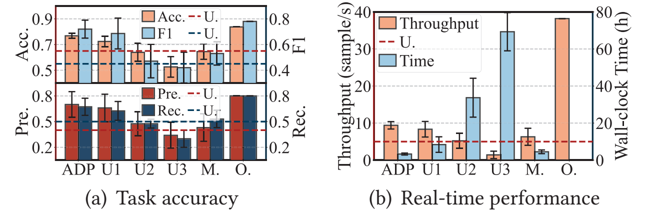
4 Experiments & Evaluations

4.1 Implementation

We deploy `AgentDeploy` on an edge server and deploy various models on three mobile devices: a Raspberry Pi, a Jetson Nano board, and a mobile mini PC. The edge server and the mobile devices are connected via a router within a local area network. We use LangChain [3] as the agent framework and implement our *cross-engine inference pipeline* considering the following inference engines: ONNX Runtime [1], TensorRT [2], OpenVINO [4], TVM [7], and MLIR [12]. GPT-4o [10] serves as the backbone LLM for our agents.

4.2 Edge AI Applications & Datasets

Based on data modality, model architecture, and device heterogeneity, we select three representative edge AI applications. 1) **Image Recognition** enables real-time analysis of *visual data*. We use a *CNN*-based ResNet-152 and evaluate it on a *Raspberry Pi* using ImageNet. 2) **Speech Sentiment Analysis (SSA)** on *audio data* involves analyzing speech patterns to determine the emotional tone of spoken content. We use the *CNN+Transformer*-based Whisper model and evaluate it on a *Jetson Nano* using the TESS dataset. 3) **Vision-Language Understanding (VLU)** leverages *Vision-Language Models* (VLMs) to interpret and describe general images. We use the *Transformer*-based DeepSeek-VL-1.3b and evaluate it on *a mobile mini PC* (12 GB GPU) using the MMMU dataset.



(a) Task accuracy (b) Real-time performance

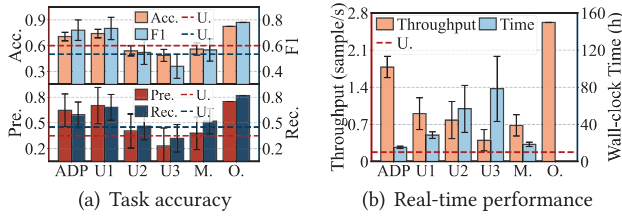
Figure 3: Evaluation results of IR (ADP - `AgentDeploy`, M. - MapCoder, O. - the original model, Acc. - classification accuracy, Pre. - precision, Rec. - recall rate, F1 - F1 score, U. - user requirement).

4.3 Metrics & Baselines

We consider four types of metrics: 1) **Resource Consumptions** include *RAM*, *GPU DRAM*, and *model size*. 2) **Task Accuracy** include *classification accuracy*, *precision*, *recall rate*, and *F1 score*. 3) **Real-time Performance** involves *throughput* and *tokens per second* (TPS). 4) The **wall-clock time** from the moment a user specifies her requirements to the final successful model deployment. We consider three types of baselines: 1) **Original Models (O.)**. 2) **MapCoder** [11], a SOTA multi-agent coding system. 3) **Human baselines**. We invite 15 developers and evenly divide them into three groups based on their self-assessed development experience and expertise. **User Group 1 (U1)** are experts with extensive experience in edge AI. **User Group 2 (U2)** are semi-experts who only have extensive experience in training and optimizing AI models but with basic knowledge about mobile devices and deployment. **User Group 3 (U3)** are non-experts who have limited knowledge of AI models and mobile devices.

4.4 Evaluation Results

Image Recognition (IR). We repeat the evaluation 20 times and obtain a set of different optimized models. From Fig. 3, **we observe that:** 1) The model deployed by `AgentDeploy` achieves a higher accuracy (> 13.8%). 2) `AgentDeploy` exhibits more consistent performance with smaller variances, whereas human baselines exhibit substantial fluctuations, especially in wall-clock time. 3) `AgentDeploy` significantly reduces the time required for executing the O&A process. Specifically, `AgentDeploy` takes only 3.3 hours in total, with 2.8 hours spent on hyperparameter tuning, 0.3 hours on model adaptation, and 0.2 hours on information retrieval. In contrast, even experts like U1 took substantial time (around 6 hours) on hyperparameter tuning. 4) The `AgentDeploy`-deployed model can sometimes achieve higher accuracy with



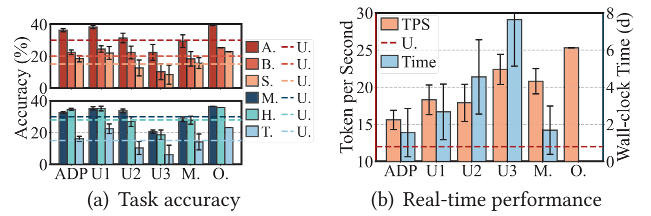
(a) Task accuracy (b) Real-time performance

Figure 4: Evaluation results of SSA.

higher throughput. *These results highlight the exceptional efficiency of* `AgentDeploy` *in finding optimal hyperparameters and generating performant optimized models for deployment.*

Speech Sentiment Analysis (SSA). This task is more challenging than IR due to a larger model size and more complex operators. Fig. 4 shows the overall performance. **We have four key findings**: 1) Only `AgentDeploy` satisfies all requirements, while the baselines fall short in distinct aspects. For example, to achieve high task accuracy, U1 sacrifices more disk space to store model parameters. Besides, to save GPU memory, U1 adopts a smaller batch size, leading to a much lower throughput (49% less than ours). 2) Despite considerable effort, U2 and U3 fail to meet most requirements due to the model's high complexity and their lack of expertise in model optimization. In contrast, `AgentDeploy` even adopts a pruning method tailored for Transformer-based models. 3) Though MapCoder completes the O&A process in a comparable amount of time to `AgentDeploy`, it delivers lower performance even than semi-experts with more resource consumption. This is because MapCoder is a general coding agent without domain knowledge in mobile AI and thus cannot effectively handle specific O&A tasks. 4) `AgentDeploy` takes significantly less time (17 hours) for O&A, while U2 and U3 take about 60 hours and 80 hours. The reasons are twofold. During model optimization, `AgentDeploy` efficiently reuses previous hyperparameters to narrow down search spaces and accelerate optimization, while U2 and U3 rely on brute-force search. During adaptation, U2 and U3 repeatedly fail to deploy the model due to mismatched operators and Jetson Nano's complex inference environment. As a result, they run the model on CPUs, leading to much lower inference speed and throughput. In contrast, `AgentDeploy` partitions the model using our OAMP module and selectively switches between different engines via our CEIP module for efficient inference. *These results further showcase that* `AgentDeploy` *can efficiently perform model optimization even for complex tasks while handling heterogeneous mobile AI environments.*

Vision-Langauge Understanding. Fig. 5 shows the overall performance. **We find that**: 1) `AgentDeploy`'s ability to reduce the time required for O&A becomes more evident with larger models. It can achieve up to an 80% reduction compared to humans. U2 and U3 even spend more than 4 days finishing the task due to their unfamiliarity with VLMs. 2)



(a) Task accuracy (b) Real-time performance

Figure 5: Evaluation results of VLU across different Q&A topics (A. - art, B. - business, S. - science, M. - medicine, H. - humanity, T. - technology).

The accuracy of `AgentDeploy` is slightly lower than that of U1. A deeper investigation reveals that U1 downloads several VLU datasets and fine-tunes the optimized model to enhance its performance. Thus, we can also add tailored prompts for `AgentDeploy` to proactively adopt more advanced techniques to further optimize the model. *These results exhibit the outstanding efficiency of* `AgentDeploy`*, even when confronted with large models, stemming from our four fully automated and collaborative agents with enhanced domain expertise.*

5 Discussions

In LLM agent systems, a common challenge is the generation of incorrect intermediate results. These errors can propagate and lead to incorrect final outputs, significantly impacting the reliability of the agent. To address this, `AgentDeploy` incorporates Reflexion [18] that enables agents to reflect on failed attempts and iteratively improve. *Moreover, to further enhance the robustness of* `AgentDeploy` *in helping agents detect and recover from mistakes during multi-step reasoning, we plan to orthogonally integrate more advanced techniques (e.g., ReAct [22] and PAL [9]) as part of our future work.*

6 Conclusion

`AgentDeploy` is a multi-agent automatic model optimization and adaptation (O&A) system for heterogeneous edge AI applications. Armed with four specialized agents to address three distinct technical challenges, `AgentDeploy` significantly enhances efficiency with enhanced model performance. Experiments demonstrate `AgentDeploy`'s high generalizability in adapting various AI models to heterogeneous mobile devices. We believe `AgentDeploy` can unlock the potential to drive the practical deployment of open-source AI models across diverse and distributed mobile devices in real-world everyday applications.

Acknowledgments

We sincerely thank all anonymous reviewers for their valuable suggestions. This work is supported by Hong Kong GRF under Grant No. 15206123 and No. 15211924. Yuanqing Zheng is the Corresponding Author.

References

[1] 2021. ONNX Runtime. https://onnxruntime.ai/.
[2] 2021. TensorRT. https://github.com/NVIDIA/TensorRT.
[3] 2022. LangChain. https://www.langchain.com/.
[4] 2023. OpenVINO. https://github.com/openvinotoolkit/openvino.
[5] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, et al. 2019. Optuna: A next-generation hyperparameter optimization framework. In *ACM KDD*.
[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, et al. 2020. Language models are few-shot learners. *NeurIPS* (2020).
[7] Tianqi Chen, Thierry Moreau, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *USENIX OSDI*.
[8] Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. 2026. V-Droid: Advancing Mobile GUI Agent Through Generative Verifiers. (2026).
[9] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *ICML*. 10764–10799.
[10] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).
[11] Md Ashraful Islam et al. 2024. MapCoder: Multi-Agent Code Generation for Competitive Problem Solving. *arXiv preprint arXiv:2405.11403* (2024).
[12] Tian Jin, Gheorghe-Teodor Bercea, et al. 2020. Compiling onnx neural network models using mlir. *arXiv preprint arXiv:2008.08272* (2020).
[13] Adam Paszke, Sam Gross, Francisco Massa, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *NeurIPS* (2019).
[14] Leming Shen, Qiang Yang, Kaiyan Cui, Yuanqing Zheng, Xiao-Yong Wei, Jianwei Liu, and Jinsong Han. 2024. Fedconv: A learning-on-model paradigm for heterogeneous federated clients. In *ACM MobiSys*. 398–411.
[15] Leming Shen, Qiang Yang, Xinyu Huang, Zijing Ma, and Yuanqing Zheng. 2025. GPIoT: Tailoring Small Language Models for IoT Program Synthesis and Development. In *ACM SenSys*. 199–212.
[16] Leming Shen, Qiang Yang, Yuanqing Zheng, and Mo Li. 2025. Autoiot: Llm-driven automated natural language programming for aiot applications. In *ACM MobiCom*. 468–482.
[17] Leming Shen and Yuanqing Zheng. 2024. Iotcoder: A copilot for iot application development. In *ACM MobiCom*. 1647–1649.
[18] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems* 36 (2023), 8634–8652.
[19] Jason Wei, Xuezhi Wang, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS* 35 (2022), 24824–24837.
[20] Huatao Xu, Liying Han, Qirui Yang, Mo Li, and Mani Srivastava. 2024. Penetrative AI: Making LLMs Comprehend the Physical World. In *ACL*.
[21] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *NeurIPS* 36 (2023), 11809–11822.
[22] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).